Verwendung neuronaler Netze zur Handverfolgung in OptiTrack

Using Neural Networks for Hand Tracking in OptiTrack

Universität Bremen

Institute for Computer Graphics and Virtual Reality

Denis Golubev
(Matrikelnummer: 2927151)

12. Januar 2020

Erstgutachter: Prof. Dr. Gabriel Zachmann
Zweitgutachter: Prof. Dr.-Ing. Udo Frese
Betreuer: M.Sc. Janis Roßkamp

ERKLÄRUNG

Ich versichere, die Masterarbeit oder den von mir zu verantwortenden Teil einer Gruppenarbeit*) ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

*) Bei einer Gruppenarbeit muss die individuelle Leistung deutlich abgrenzbar und bewertbar sein und den Anforderungen der Absätze 1 und 2 §13 der PO - Allgemeiner Teil - entsprechen.

Bremen, den _____

_____
(Unterschrift)

Inhaltsverzeichnis

1. Einleitung

Bei der Interaktion mit einer virtuellen Umgebung und bei Effekten in Filmen wird häufig auf echte Bewegungen zurückgegriffen. Eine Möglichkeit ist dabei die Verwendung von Motion-Capture-Verfahren zur Nachverfolgung von Nutzern oder Schauspielern. Insbesondere die Rekonstruktion einer Handkonfiguration ist dabei schwierig, da die Hand viele komplexe Bewegungen durchführen kann. Beim Motion-Capture muss zunächst zwischen optischen und nicht-optischen Systemen unterschieden werden. Ein Beispiel für Letztere ist die CyberGlove (CyberGlove Systems Inc., 2019), die eine Hand sowie die Finger basierend auf Winkelsensoren an den Gelenken verfolgen kann. Bei optischen Systemen existieren des Weiteren Verfahren mit oder ohne Marker, die an dem zu verfolgenden Objekt angebracht werden müssen.

Die Autoren von (Mueller u. a., 2019) haben ein markerloses Motion-Capture-Verfahren für eine Rekonstruktion der Interaktion zweier Hände in Echtzeit entwickelt. Des Weiteren kann die Form der Hände bestimmt werden. Dies wurde durch nur eine Tiefenkamera realisiert. Ein neuronales Netzwerk nutzt diese Tiefeninformationen, um die Hände voneinander zu trennen. Die Rekonstruktion der Handkonfiguration wird durch die Anpassung des virtuellen Handmodells basierend auf der Oberfläche an die Punktwolke der Aufnahme durchgeführt.

In (Zhang u. a., 2019) wurde ein anderes markerloses Motion-Capture-Verfahren für die gleichzeitige Erkennung von Handkonfigurationen und deformierbaren Gegenständen in Echtzeit vorgestellt. Dabei werden die Bilder zweier Tiefenkameras aus entgegengesetzten Richtungen kombiniert und durch ein neuronales Netzwerk in Hand und Gegenstand getrennt. Ein weiteres neuronales Netzwerk wird für die Rekonstruktion der eigentlichen Handpose genutzt.

Eine alternative Herangehensweise wurde in (Schröder u. a., 2014) beschrieben. Dabei wird die Kinect-Tiefenkamera verwendet und das virtuelle Handmodell direkt an die Punktwolke über inverse Kinematik angepasst. Durch zuvor aufgenommene gültige Handkonfigurationen wird die Anzahl an möglichen Parametern reduziert.

Optische, markerbasierte Motion-Capture-Systeme können nach (Pintaric u. Kaufmann, 2007) aktive und passive Marker zur Nachverfolgung verwenden. Aktive Marker benötigen eine Stromversorgung und sind teurer in der Anschaffung und Instandhaltung. Für passive Marker, wie mit einem retroreflektiven Material beschichtete Kugeln, müssen die Kameras mit einer Lichtquelle ausgestattet werden.

In (Alexanderson u. a., 2017) wurde ein auf passiven Markern basierendes Verfahren zum Labeling von Hand-Markern in Echtzeit vorgestellt. Es wird ein mit Markern bestückter Handschuh verwendet. Dabei werden Daten durch das OptiTrack-System aufgenommen und daraus Hypothesen für mögliche Labelings generiert. Aus einer Serie an Frames werden dann die Labelings ausgewählt, die eine möglichst flüssige Bewegung liefern. Zudem wurde untersucht, wie viele Marker bessere Labelings nach dieser Methode liefern.

In (Han u. a., 2018a) wurde ebenfalls ein auf OptiTrack basierendes Verfahren zum Labeling der Hand-Marker entwickelt. Die Autoren verwenden einen Handschuh mit 19 passiven Markern. Die Marker werden mit einer bestimmten optischen Achse orthographisch auf ein Tiefenbild projiziert und durch ein Convolutional Neural Network verarbeitet. Dieses Netzwerk liefert wiederum 19 Markerpositionen, dessen Reihenfolge den jeweiligen Labels entspricht. Die Ausgabe des Netzwerks wird dann wieder den Eingabedaten zugeordnet, um das Label des jeweiligen Markers zu bestimmen. Dieses Verfahren liefert nach den Autoren von (Han u. a., 2018a) ein besseres Ergebnis als (Alexanderson u. a., 2017) in Bezug auf den Anteil der Frames mit richtig zugeordneten Markern. Des

Weiteren kann das System aus (Han u. a., 2018a) die Handkonfiguration über inverse Kinematik aus diesen gelabelten Markern rekonstruieren.

Das Ziel dieser Masterarbeit ist die Entwicklung eines Systems, welches Marker-Positionen aus OptiTrack verarbeiten und daraus eine Handkonfiguration zur Darstellung einer virtuellen Repräsentation der echten Hand rekonstruieren kann. Ein möglicher darauf aufbauender Anwendungsfall wäre die Interaktion mit einer virtuellen Umgebung durch diese Repräsentation oder die Animation von Charakteren bei der Spieleentwicklung. Dies ist jedoch nicht Teil dieser Masterarbeit.

Die Arbeitsgruppe besitzt bereits ein OptiTrack-System und das Verfahren aus (Han u. a., 2018a) lieferte laut den Autoren sehr gute Ergebnisse zur Echtzeit. Des Weiteren ist OptiTrack sehr genau und kann Markerpositionen mit einer hohen Datenrate bereitstellen (siehe Abschnitt 2.1).

Bei der Verwendung von markerlosen Motion-Capture-Systemen mit nur einer oder zwei Tiefenkameras, entstehen schnell Verdeckungen bei Handbewegungen oder durch den Körper des Nutzers. OptiTrack erlaubt es mehr als zwei Kameras zu verwenden und somit deutlich mehr Blickrichtungen zu betrachten. Des Weiteren befassen sich diese Verfahren größtenteils nur mit Händen oder anderen spezifischen Objekten. Auch hierbei ist eine Verwendung von zum Beispiel OptiTrack im Hinblick auf eine spätere Erweiterung des Systems mit zum Beispiel anderen Objekten oder einer Nachverfolgung des gesamten Körpers von Vorteil.

Vor diesem Hintergrund fiel die Entscheidung darauf, das in (Han u. a., 2018a) beschriebene Verfahren zunächst zu reimplementieren und dann den Effekt verschiedener Erweiterungen sowie einer anderen Netzwerk-Architektur zu untersuchen. Auf eine automatische Kalibrierung der Handmodelle wird im Gegensatz zu (Han u. a., 2018a) verzichtet.

Im Rahmen der Arbeit wird nach der Reimplementierung das Auswahlverfahren für die zufälligen optischen Achsen aus (Han u. a., 2018a) angepasst, indem alternative Bewertungskriterien betrachtet werden. Des Weiteren wird eine Verwendung von mehreren Tiefenbildern aus verschiedenen, festen Blickrichtungen in einem Frame untersucht. Die dabei resultierenden Labelings werden verglichen und das beste Labeling ausgewählt. Im Rahmen dessen wird auch ein Durchschnitt dieser Labelings betrachtet.

Daraufhin wird ein neues Netzwerk mit der in (Han u. a., 2018a) beschrieben Architektur gelernt und die Auswirkung anderer Verlustfunktionen und Optimierungsalgorithmen untersucht. Im Rahmen dessen wird die Verwendung eines Netzwerks basierend auf der ResNet-50-Architektur aus (He u. a., 2016) betrachtet, da dieses für das hier vorliegende Regressionsproblem nach (Lathuilière u. a., 2019) einen Lösungsansatz darstellt.

Darüber hinaus wird die Rekonstruktion der Handkonfiguration angepasst und die globale Position sowie Rotation getrennt von den Fingerpositionen betrachtet. Darauf basierend werden die Fingerkonfigurationen unabhängig voneinander bestimmt.

Zur Darstellung von Aufnahmen oder Marker-Daten zur Echtzeit wird eine einfache graphische Oberfläche entwickelt. Die Markerpositionen können dabei über NatNet von OptiTrack oder über VRPN von anderen Quellen empfangen werden. Es wird ein kleines Tool zur Weiterleitung von Marker-Daten von NatNet über VRPN entwickelt, da NatNet nur auf Windows verfügbar ist.

Um echte Daten aufzunehmen und das System testen zu können, wird ein Handschuh basierend auf Abbildungen aus (Han u. a., 2018a) entwickelt. Danach wird eine Aufnahme mit einfachen und komplexen Bewegungen durchgeführt. Auf dieser Aufnahme wird der Labeling-Schritt durchgeführt, um ein erstes Labeling zu bekommen, welches dann manuell geprüft und korrigiert wird. Diese nun richtig gelabelte Aufnahme wird beim Vergleich der verschiedenen Verfahren und Netzwerke verwendet.

In Kapitel 2 wird zunächst auf die notwendigen Grundlagen eingegangen. Dies umfasst auch eine Zusammenfassung von (Han u. a., 2018a) in Abschnitt 2.14, worauf diese Arbeit größtenteils basiert. Des Weiteren wird kurz auf die Funktionsweise von optischen, auf passiven Markern basierenden Motion-Capture-Systemen in Abschnitt 2.1 eingegangen.

Die Beschreibung der Implementierung ist in Kapitel 3 zu finden. In Abschnitt 3.1 wird zunächst

die grobe Architektur des Systems vorgestellt. Darauf folgt eine Beschreibung der Reimplementierung der Labeling-Komponente von (Han u. a., 2018a) in den Abschnitten 3.4 bis 3.7 beschrieben. In den Abschnitten 3.8 und 3.9 wird auf die Erweiterung dieses Labeling-Verfahrens eingegangen. Der Abschnitt 3.10 beschreibt die Rekonstruktion von Handkonfigurationen nach (Han u. a., 2018a) sowie in dieser Arbeit entwickelte Erweiterungen. Das Training von neuen neuronalen Netzwerken wird in Abschnitt 3.11 erläutert. Die Aufnahme von echten Daten zur Auswertung des Systems ist in Abschnitt 3.12 zu finden. Abschnitt 3.13 beschreibt die Netzwerkunterstützung zum Empfang von Markerpositionen vom OptiTrack-System.

Das 4. Kapitel beschreibt die Ergebnisse dieser Arbeit. Zunächst wird in Abschnitt 4.1 auf den Ablauf der Auswertung eingegangen. Dann wird als Erstes die Reimplementierung in Abschnitt 4.2 betrachtet. Daraufhin werden die Effekte von Anpassungen am Labeling-Verfahren in Abschnitt 4.3 und von anderen Neuronalen Netzwerken in Abschnitt 4.4 untersucht. Das Kapitel schließt mit der Betrachtung der Änderungen am Rekonstruktionsschritt in Abschnitt 4.5 ab.

In Kapitel 5 werden die Ergebnisse nochmal zusammengefasst und Schlussfolgerungen daraus gezogen. Im darauffolgenden 6. Kapitel wird kurz auf zukünftige Forschungsansätze basierend auf dieser Arbeit eingegangen.

2. Grundlagen

2.1. OptiTrack

OptiTrack (NaturalPoint, Inc., 2019, General FAQs) ist ein optisches Motion-Capture-System, welches mehrere zeitlich synchronisierte Infrarot-Kameras zur Nachverfolgung kugelförmiger passiver Marker verwendet. Als Beleuchtung werden Infrarot-LEDs eingesetzt. Dabei wird von jeder Kamera ein zweidimensionales Bild aufgenommen. Darauf werden dann zunächst die 2D-Positionen der Marker bestimmt und deren Überlappungen auf den verschiedenen Bildern betrachtet. Darüber lassen sich dann die 3D-Positionen der Marker berechnen.

Die passiven Marker sind in OptiTrack mit einem retroreflektiven Material beschichtet (Natural Point, Inc., 2018a). Nach (Wikipedia, 2019d) wird das Licht von solchen Materialien größtenteils an die Lichtquelle zurückreflektiert. Dies kommt zum Beispiel bei Straßenschildern zum Einsatz.

Wie in (Natural Point, Inc., 2017b) beschrieben, kann den rekonstruierten Markern entweder manuell oder automatisch ein generiertes Label zugeordnet werden. Das jeweilige Label wird dem selben Marker wieder in nachfolgenden Frames zugewiesen. Dabei können jedoch Lücken entstehen und eventuell neue Labels für denselben Marker generiert werden.

Das OptiTrack-System kann nach (NaturalPoint, Inc., 2019, General FAQs) bei einer guten Systemkonfiguration eine Genauigkeit der Markerpositionen im Sub-Millimeter-Bereich erreichen. Die Kameras unterstützen dabei eine Datenrate von bis zu 360 Hz.

2.1.1. Funktionsweise

Die Funktionsweise von einem auf Infrarot-Kameras und passiven Markern basierenden Motion-Capture-System wird in (Pintaric u. Kaufmann, 2007) beschrieben. Die Kameras sind dabei auch mit Infrarot-LEDs ausgestattet und nehmen alle Bildpixel gleichzeitig auf. Dabei wird die Bildaufnahme aller verbundenen Kameras synchronisiert. Die kugelförmigen Marker sind mit einem retroreflektiven Material beschichtet.

In den Aufnahmen wird in (Pintaric u. Kaufmann, 2007) dann über eine Kombination aus Bildverarbeitungsalgorithmen für jeden sichtbaren Marker die zweidimensionale Position des Marker-Zentrums bestimmt. Dabei reicht häufig eine Segmentierung der Bilddaten zwischen Hintergrund und Marker sowie eine nachfolgende Betrachtung dieser Regionen aus. Überlappen sich Marker, wird eine Hough-Transformation durchgeführt, um die Zentren der Kreise und somit die Zentren der Marker zu bestimmen.

Über die bekannten Kamerapositionen und die Marker-Zentren lassen sich nach (Pintaric u. Kaufmann, 2007) Geraden in dem Raum generieren. Dann wird nach einer optimalen Zuordnung zwischen jeweils einer Gerade jeder Kamera gesucht. Mit Hilfe dieser Zuordnungen können dann die dreidimensionalen Positionen der jeweiligen Marker bestimmt werden.

2.1.2. Dateiformat der Tracking-Aufnahmen

Ein mögliches Dateiformat, um Tracking-Aufnahmen zu sichern, ist das Track Row Column (TRC) Format (Natural Point, Inc., 2017a). Es ist textbasiert und enthält verschiedene Metadaten in den ersten fünf Zeilen. Darauf folgen zeilenweise Frames von Markerpositionen. In allen Zeilen wird das Tabulator-Zeichen zur Trennung verwendet. Nicht sichtbare Marker werden als leere Spalten

dargestellt. In diesem Format stehen auch die in (Han u. a., 2018a) mitgelieferten Trainingsdaten zur Verfügung. In Abbildung 2.1 ist ein Auszug aus solch einer Datei zu sehen.

	A	B	C	D	E	F	G	H	I	J	K
1	PathFileType	4	(X/Y/Z)	C:/export-path/recording.trc							
2	DataRate	CameraRate	NumFrames	NumMarkers	Units	OrigDataRate	OrigDataStartFrame	OrigNumFrames			
3	60	60	1927	19	cm	60	0	3412			
4	Frame#	Time	thumb_base1			thumb_base2			thumb_middle		
5	X0	Y0	Z0	X1	Y1	Z1	X2	Y2	Z2		
6	0	0	-60.5546	103.457	265.362	-59.559	103.773	263.092	-57.6961	102.978	260.227
7	1	0.0166	-59.0476	103.548	265.252	-58.1462	103.86	262.984	-56.22	103.062	260.107

Abbildung 2.1.: Ein stark gekürzter Auszug einer im Rahmen dieser Arbeit aufgenommenen Tracking-Datei im TRC-Format, angezeigt mit LibreOffice Calc.

2.2. Orthographische Projektion

Eine der möglichen Transformationen, um dreidimensionale Daten auf einer Ebene darzustellen, ist die orthographische Projektion. Dabei wird mit Hilfe zur Projektionsebene paralleler Geraden projiziert (Hughes u. a., 2014, S.315f.).

Zur Bestimmung einer solchen Transformation, die Punkte aus dem Welt-Koordinatensystem in das Kamera-Koordinatensystem überführt, ist zunächst eine Kamera-Spezifikation notwendig. Diese setzt sich aus der Blickrichtung, die durch ihre optische Achse gegeben ist, und der Position der Kamera zusammen. Die Kameraposition ist bei dieser Arbeit nicht von Relevanz, da später nur die Blickrichtung eine Rolle spielt und die optische Achse durch den Ursprung verläuft.

Die Blickrichtung wird durch den Vektor $look$ beschrieben und die Position als P bezeichnet. Zusammen mit dem Vektor v_{up}, der die Richtung Oben von der Kamera aus gesehen, bezeichnet, können die weiteren, für die Transformation notwendigen, Vektoren u (nach rechts von der Kamera aus), v (nach oben in der Welt), w (entgegengesetzt der Kamera-Blickrichtung) berechnet werden (Hughes u. a., 2014, S.303f.). Im Folgenden ist deren Berechnung nach (Scratchapixel, 2016) zu finden:

$$\begin{aligned} w &= \frac{-look}{||look||} \\ u &= v_{up} \times w \\ v &= w \times u \end{aligned} \tag{2.1}$$

Die Transformation N ohne Translation lautet dann:

$$N = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \tag{2.2}$$

2.3. Splatting

Die Darstellung von dreidimensionalen Datenpunkten als ein zweidimensionales Bild wird als Volumen-Rendering bezeichnet. Die Verwendung von Datenpunkten anstatt konkreten Oberflächen zur Repräsentation von Objekten hat Vorteile beim Rendering von zum Beispiel medizinischen Bilddaten. Dabei existieren verschiedene Darstellungsverfahren. Es wird unterschieden, ob vom Bild aus Strahlen in den Raum, wie zum Beispiel beim Raycasting, geschickt werden oder ob die dreidimensionalen Daten auf das Bild abgebildet werden (Hughes u. a., 2014, S.349,S.391)(Westover, 1990, S.1).

Im letzteren Verfahren muss beim Rendering ermittelt werden, inwiefern jeder Datenpunkt zum Bild beiträgt. Dieser Beitrag wird nachfolgend als Fußabdruck beziehungsweise *footprint* bezeichnet. Splatting ist dann die Verteilung des Fußabdrucks mithilfe einer Footprint-Funktion $f(x)$. Dabei besitzt der Fußabdruck einen Radius, in dem er das Bild beeinflusst. Für jeden Pixel innerhalb dieses Radius liefert $f(x)$ ein Gewicht basierend auf dem Abstand dieses Pixels zur Position des Datenpunkts in Bildkoordinaten. Mit diesem Gewicht wird der zugehörige Pixelwert des Datenpunkts auf das Bild angewendet (Westover, 1990, S.1ff.).

2.3.1. Rendering von Körpern mithilfe von Punkten

Ein mögliches Verfahren zum Rendering von dreimensionalen Körpern ist die Repräsentation dieser mithilfe von Punkten. Dabei trägt jeder Punkt zu jedem Pixel auf dem Bild basierend auf dem Abstand des Punktes in Bildkoordinaten zu dem jeweiligen Pixelzentrum bei. Diese Distanz wird dabei mit einer Gaußfunktion gefiltert. Dadurch kann die Textur des Körpers dargestellt werden (Levoy u. Whitted, 1985, S.1, S.5).

2.4. Kovarianzmatrix

Der Erwartungswert $\mathbb{E}[f(x)]$ einer Funktion $f(x)$ ist nach (Goodfellow u. a., 2016, S.58f.) der Durchschnitt, den die Funktion f annimmt, wenn x aus der Wahrscheinlichkeitsverteilung $P(x)$ gezogen wird. Die Varianz $Var(f(x))$ gibt dann an, inwiefern sich der Wert der Funktion $f(x)$ von dem Erwartungswert unterscheidet, wenn verschiedene x aus $P(x)$ gezogen werden. Im Folgenden ist die Definition der Varianz zu finden (Goodfellow u. a., 2016, S.59):

$$Var(f(x)) = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right] \tag{2.3}$$

Die Kovarianz gibt nach (Goodfellow u. a., 2016, S.59) an, wie sich zwei Funktionswerte $f(x)$ und $g(y)$ zueinander verhalten. Eine hohe positive Kovarianz weist darauf hin, dass beide Funktionen gleichzeitig hohe Werte annehmen. Ist die Kovarianz stark negativ, nimmt $f(x)$ einen hohen Wert ein, wenn $g(x)$ klein ist, und umgekehrt. Die Kovarianz ist wie folgt definiert:

$$Cov(f(x), g(x)) = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])\right] \tag{2.4}$$

Die *Kovarianzmatrix* eines Zufallsvektors $\mathbf{x} \in \mathbb{R}^n$ beschreibt das Verhalten seiner Elemente zueinander und ist folgendermaßen definiert (Goodfellow u. a., 2016, S.59f.):

$$Cov(\mathbf{x}) = \begin{bmatrix} Cov(\mathbf{x}_1, \mathbf{x}_1) = Var(\mathbf{x}_1) & Cov(\mathbf{x}_1, \mathbf{x}_2) & \dots & Cov(\mathbf{x}_1, \mathbf{x}_n) \\ Cov(\mathbf{x}_2, \mathbf{x}_1) & Var(\mathbf{x}_2) & \dots & Cov(\mathbf{x}_2, \mathbf{x}_n) \\ \dots & \dots & \dots & \dots \\ Cov(\mathbf{x}_n, \mathbf{x}_1) & Cov(\mathbf{x}_n, \mathbf{x}_2) & \dots & Var(\mathbf{x}_n) \end{bmatrix} \tag{2.5}$$

2.4.1. Eigenwerte und -vektoren

Matrizen können zerlegt werden, um mehr Informationen aus ihnen zu bekommen. Ein konkretes Verfahren ist die Eigenwertzerlegung in Eigenvektoren und Eigenwerte. Ein Eigenvektor v einer quadratischen Matrix $\mathbf{A}$ zeichnet sich dadurch aus, dass die Multiplikation dessen mit eben dieser Matrix nur eine Skalierung ist. Es gilt also:

$$A * v = \lambda * v \tag{2.6}$$

Dabei ist λ der zu diesem Eigenvektor gehörende Eigenwert. Eine Eigenwertzerlegung kann in komplexen Eigenwerten und -vektoren resultieren, jedoch nicht für symmetrische Matrizen mit reellen

Einträgen. Eine Matrix $\mathbf{A}$ ist symmetrisch, wenn $\mathbf{A} = \mathbf{A}^\top$ gilt. Die Zerlegung hat dann die Form $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$, wobei $\mathbf{Q}$ eine orthogonale Matrix ist, dessen Spalten die Eigenvektoren darstellen. $\mathbf{\Lambda}$ ist eine diagonale Matrix, dessen Einträge die zu den jeweiligen Eigenvektoren gehörenden Eigenwerte sind. Somit ist die Anzahl der Eigenwerte und -vektoren durch die Größe der Matrix gegeben. Die zerlegte Matrix kann dann als eine Skalierung des Raums in die Richtung der jeweiligen Eigenvektoren um ihren Eigenwert betrachtet werden (Goodfellow u. a., 2016, S.39ff.). Eine Kovarianzmatrix ist symmetrisch, da die folgenden Aussagen gelten:

$$
\begin{aligned}
Cov(f(x), g(x)) &= \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])\right] && (2.7)\\
&= \mathbb{E}\left[(g(y) - \mathbb{E}[g(y)])(f(x) - \mathbb{E}[f(x)])\right] && (2.8)\\
&= Cov(g(x), f(x)) && (2.9)\\
Cov(\mathbf{x}) &= \begin{bmatrix} Cov(\mathbf{x}_1, \mathbf{x}_1) = Var(\mathbf{x}_1) & Cov(\mathbf{x}_1, \mathbf{x}_2) & \dots & Cov(\mathbf{x}_1, \mathbf{x}_n) \\ Cov(\mathbf{x}_2, \mathbf{x}_1) & Var(\mathbf{x}_2) & \dots & Cov(\mathbf{x}_2, \mathbf{x}_n) \\ \dots & \dots & \dots & \dots \\ Cov(\mathbf{x}_n, \mathbf{x}_1) & Cov(\mathbf{x}_n, \mathbf{x}_2) & \dots & Var(\mathbf{x}_n) \end{bmatrix} && (2.10)\\
&= \begin{bmatrix} Cov(\mathbf{x}_1, \mathbf{x}_1) = Var(\mathbf{x}_1) & Cov(\mathbf{x}_2, \mathbf{x}_1) & \dots & Cov(\mathbf{x}_n, \mathbf{x}_1) \\ Cov(\mathbf{x}_1, \mathbf{x}_2) & Var(\mathbf{x}_2) & \dots & Cov(\mathbf{x}_n, \mathbf{x}_2) \\ \dots & \dots & \dots & \dots \\ Cov(\mathbf{x}_1, \mathbf{x}_n) & Cov(\mathbf{x}_2, \mathbf{x}_n) & \dots & Var(\mathbf{x}_n) \end{bmatrix} && (2.11)\\
&= Cov(\mathbf{x})^\top && (2.12)
\end{aligned}
$$

2.4.2. Stichprobenmittelwert und -varianz

Der Erwartungswert einer Funktion kann über eine Menge von Stichproben geschätzt werden. Eine mögliche Schätzung ist der Stichprobenmittelwert, der für m Stichproben $x_1, x_2, \dots, x_m$ wie folgt definiert ist (Goodfellow u. a., 2016, S.122):

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x_i \tag{2.13}$$

Die Varianz einer Funktion kann unter anderem durch die Stichprobenvarianz geschätzt werden. Dabei wird die Variante ohne Bias bevorzugt und ist wie folgt definiert (Spruyt, 2014; Goodfellow u. a., 2016, S.123f.):

$$\sigma^2 = \frac{1}{m-1} \sum_{m}^{i=1} (x_i - \mu)^2 \tag{2.14}$$

2.4.3. Stichprobenkovarianz

Die Stichprobenkovarianz ist nach (Hoffbeck u. Landgrebe, 1996, S.3f.) ein übliche Schätzung der Kovarianz basierend auf einer Menge von Stichproben. Sie kann zwischen den Variablen x und y nach (Spruyt, 2014) folgendermaßen bestimmt werden:

$$\frac{1}{m-1} \sum_{i=1}^{m} (x_i - \mu)(y_i - \mu) \tag{2.15}$$

2.5. Eulersche Winkel

Eine komplexe Rotation kann aus drei einfachen Rotationen zusammengesetzt werden. Die drei dabei verwendeten Winkel werden als *eulersche Winkel* bezeichnet. Es existiert keine eindeutige

Definition für eine Rotation um eulersche Winkel, in der Computergrafik wird jedoch am häufigsten ein Produkt aus den folgenden drei Rotationen verwendet:

$$R_{xy}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

$$R_{yz}(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} \tag{2.17}$$

$$R_{zx}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{2.18}$$

Das Subskript bezeichnet dabei die Ebene auf der rotiert wird. R_{xy}, R_{yz} und R_{zx} sind also Rotationen um die Z-, die X- sowie die Y-Achse. Die Rotationsmatrix R setzt sich folgendermaßen zusammen:

$$R = R_{yz}(\psi) * R_{zx}(\theta) * R_{xy}(\phi) \tag{2.19}$$

Dabei wird also zunächst um ϕ auf der XY-Ebene, also um die Z-Achse, rotiert. Danach folgt eine Rotation von θ auf der ZX-Ebene, also um die Y-Achse. Zum Schluss wird um ψ auf der YZ-Ebene, also um die X-Achse, rotiert. Die Winkel können dabei als *roll*, *pitch* und *yaw* bezeichnet werden (Hughes u. a., 2014, S.266ff.).

2.6. Tensor

Als ein Tensor wird ein Array mit einer variablen Anzahl an Dimensionen bezeichnet. Darunter fallen auch Skalare, Vektoren und Matrizen (Goodfellow u. a., 2016, S.29ff.).

2.7. Stochastic Gradient Descent

Zum Trainieren eines Netzwerks kann der Optimierungsalgorithmus Stochastic Gradient Descent (SGD) verwendet werden (Goodfellow u. a., 2016; Lathuilière u. a., 2019; Han u. a., 2018a, S.6). In den folgenden Abschnitten wird dieser Algorithmus vorgestellt.

2.7.1. Zielfunktion

Eine zu optimierende Funktion wird als Zielfunktion bezeichnet. Eine zu minimierende Zielfunktion ist eine Verlustfunktion[1] (Goodfellow u. a., 2016, S.79ff.).

2.7.2. Gradient Descent

Bei dem Gradient-Descent-Verfahren wird nach Minima einer zu optimierenden Funktion $f(x)$ gesucht, indem sich Schrittweise entgegen des Vorzeichens der Ableitung $f'(x)$ der Funktion bewegt wird. Bei einer Funktion $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ mit mehrdimensionalen Eingaben wird die Bewegung analog in Richtung des negativen Gradienten durchgeführt. In einem Schritt wird das nächste x' mit einer Lernrate ϵ folgendermaßen bestimmt (Goodfellow u. a., 2016, S.79ff.):

$$x' = x - \epsilon \nabla_x f(x) \tag{2.20}$$

[1] Im Englischen wird eine Verlustfunktion als *cost function*, *loss function* oder *error function* bezeichnet.

2.7.3. Stochastic Gradient Descent

Bei der Bestimmung des Gradienten der Verlustfunktion muss häufig das Netzwerk mit jedem Trainingsdatum ausgeführt werden. Dies kann je nach Menge der Trainingsdaten und Komplexität des Netzwerks eine lang andauernde Operation sein. In der Praxis kann eine kleine Menge an Trainingsdaten zufällig ausgewählt werden und der Gradient basierend darauf ermittelt werden. Diese Teilmenge wird als *Minibatch* bezeichnet. Zum Einen konvergieren viele Optimierungsalgorithmen mit einer kleineren Zahl an Berechnungen, wenn schnell ein ungefährer Gradient berechnet wird. Zum Anderen kann es vorkommen, dass verschiedene Trainingsdaten ähnlich zur Gradientberechnung beitragen und dessen Auswertung somit redundant ist (Goodfellow u. a., 2016, S.270ff.).

Ein Gradient Descent mit Minibatches wird als ein Stochastic Gradient Descent bezeichnet (Goodfellow u. a., 2016, S.286).

2.7.4. Lernrate

Die Lernrate ist einer der wichtigsten Hyperparameter bei der Optimierung eines Netzwerks mit SGD. Wenn der Trainingsfehler unter verschiedenen Lernraten betrachtet wird, ergibt sich eine U-förmige Kurve. Bei Gradient Descent führt eine zu große Lernrate dazu, dass sich der Fehler vergrößert. Eine zu kleine Lernrate führt dazu, dass das Lernen langsamer ist und eventuell bei einem hohen Trainingsfehler stecken bleibt. Die Lernrate muss also korrekt und nicht zu klein oder zu groß gewählt werden. Dabei muss jedoch beachtet werden, dass sich bei gleicher Lernrate der Testfehler anders als der Trainingsfehler verhält (Goodfellow u. a., 2016, S.417f.).

2.7.5. Moment

Beim Stochastic Gradient Descent mit Moment tragen in jeder Iteration Gradienten aus bisherigen Iterationen mit einem festgelegten Anteil zum aktuellen Gradienten bei. Die Bewegung in die Richtung der bisherigen Gradienten wird dadurch fortgesetzt. Der Lernvorgang soll damit bei kleinen oder rauschenden Gradienten beschleunigt werden (Goodfellow u. a., 2016, S.288ff.).

Nesterov-Variante

Bei der Nesterov-Variante wird der aktuelle Gradient erst berechnet, nachdem ein Anteil der bisherigen Gradienten die Parameter des Netzwerks bereits beeinflusst hat (Goodfellow u. a., 2016, S.291f.).

2.8. Convolutional Neural Network

Bei einem neuronalen Netz, das mindestens eine Faltungsoperation (*convolution*) in einer der Ebenen besitzt, handelt es sich um ein *Convolutional Neural Network*. Eine Faltungsoperation kombiniert eine Funktion f mit einer Gewichtungsfunktion w und liefert eine durch w gewichtete Summe von Werten der ursprünglichen Funktion für einen bestimmten Parameterbereich. Dies lässt sich auch auf zweidimensionale Daten wie Bilder übertragen. In diesem Fall liefert die Faltung die gewichtete Summe aller im Zielbereich betrachteter Pixel. Diese Gewichtsfunktion wird dabei häufig als *kernel* bezeichnet (Goodfellow u. a., 2016, S.321ff.). Netzwerkebenen, die diese Operation implementieren, werden nachfolgend als *Conv2D* bezeichnet. LuaTorch (Collobert u. a., 2019a) und TensorFlow (Abadi u. a., 2015) führen bei solchen Ebenen zusätzlich noch einen Bias-Term ein, der zur Summe hinzu addiert wird.

In den folgenden Abschnitten werden die verschiedenen Ebenentypen beschrieben, die in (Han u. a., 2018a) verwendet werden. Des Weiteren wird kurz auf Verlustfunktionen, die Generierung

von initialen Gewichten und alternative Optimierungsalgorithmen eingegangen. Darauf folgt eine kurze Einführung in das ResNet-50-Netzwerk.

2.8.1. Dimensionen der Netzwerkeingaben

In modernen Deep-Learning-Frameworks, wie zum Beispiel TensorFlow (Abadi u. a., 2015), werden die Eingaben in Batches verarbeitet. Aus diesem Grund werden die Eingaben als ein dreidimensionaler Tensor repräsentiert. In TensorFlow beziehungsweise der übergeordneten Schnittstelle Keras[2] (Chollet u. a., 2019) werden die zwei Formate *channel-first* und *channel-last* unterstützt, wobei letzteres standardmäßig konfiguriert ist. Das erste Format enthält in der ersten Dimension die Anzahl von Eingaben im Batch (N), in der zweiten die Anzahl der betrachteten Features (C) wie zum Beispiel die Farbkanäle in einem Bild und in den folgenden zwei Dimensionen die Höhe (H) und Breite (W) der Eingabe. Deswegen wird das erste Format kurz als *NCHW* bezeichnet. Im *channel-last*-Format liegt die Anzahl der Features in der letzten Dimension vor, also im *NHWC*-Format (Li u. a. (2016); Abadi u. a. (2015), Dokumentation zu `tf.keras.layers.Conv2D`; Goodfellow u. a. (2016), S.96).

2.8.2. Batch Normalization

Bei Netzwerken mit vielen Ebenen ist die Anpassung der Parameter unter Verwendung des ermittelten Gradienten problematisch. Obwohl der Gradient beschreibt welche Parameter angepasst werden müssen, ist es in der Praxis häufig so, dass Parameter aller Ebenen gleichzeitig verändert werden. Dies kann jedoch zu unerwarteten Änderungen führen. Batch Normalization ist der Versuch dieses Problem zu beheben und somit das Lernen von Netzwerken zu vereinfachen. Dabei werden Elemente des Gradienten, die zu starke Veränderungen hervorrufen, gleich Null gesetzt. Die Veränderungen werden dabei auf Batches betrachtet (Goodfellow u. a., 2016, S.309ff.).

2.8.3. ReLU

In einem neuronalen Netzwerk wird auf der Ausgabe eines Neurons eine Aktivierungsfunktion angewendet. Traditionelle Beispiele dafür sind $f(x) = tanh(x)$ oder $f(x) = (1 + e^{-x})^{-1}$. Neuronen mit der Aktivierungsfunktion $f(x) = max(0, x)$ werden als Rectified Linear Units (ReLUs) bezeichnet. Solche Neuronen lassen sich einfach optimieren. Neuronale Netzwerke mit ReLUs lernen schneller als solche mit der *tanh*-Aktivierung und ermöglichen die Verwendung tieferer Netzwerke (Krizhevsky u. a., 2012, S.3) (Goodfellow u. a., 2016, S.187).

2.8.4. Pooling und Max-Pooling

Eine Pooling-Ebene in einem Convolutional Neural Network fasst mehrere naheliegende Ausgaben der vorhergehenden Ebene an einer Stelle zusammen. Durch Pooling wird erreicht, dass nachfolgende Ebenen nicht auf kleine Translationen im Input reagieren. Wenn Pooling über mehrere Convolutions mit separaten Kerneln betrieben wird, kann das Netzwerk lernen, von welche Transformationen im Input der Output unabhängig sein soll. Max-Pooling wählt dabei den maximalen Wert einer quadratischen Fläche im Input aus (Goodfellow u. a., 2016, S.330).

2.8.5. FC / Dense

Als FC in (Han u. a., 2018a) und Dense in (Abadi u. a., 2015) werden vollständig verbundene Ebenen bezeichnet. Nach (Goodfellow u. a., 2016, S.321ff.) beeinflussen dabei alle Werte der vorhergehenden

[2] In Abschnitt 2.19.4 auf Seite 27 ist der Zusammenhang zwischen TensorFlow und Keras kurz beschrieben.

Ebene die nachfolgende Ebene und die Operation kann als Matrixmultiplikation mit den Gewichten verstanden werden.

2.8.6. Verlustfunktionen bei der Optimierung von Netzwerken

Bei der Optimierung von Netzwerken wird eine Verlustfunktion benötigt. Im Folgenden werden drei solcher Funktionen vorgestellt. Hierbei wird die Notation von (Goodfellow u. a., 2016, S.104f.) übernommen und y bezeichnet die betrachtete Ausgabe eine Netzwerks, $\hat{y}$ die gewünschte Ausgabe sowie m die Anzahl der Elemente.

Eine solche Verlustfunktion ist der mittlere quadratische Fehler (*Mean Squared Error*, *MSE*) und ist wie folgt definiert (Goodfellow u. a., 2016, S.105):

$$MSE = \frac{1}{m} \sum_{i=1} (\hat{y}_i - y_i)^2 \tag{2.21}$$

Es existieren nach (Lathuilière u. a., 2019, S.6) noch andere gute Verlustfunktionen. Zu diesen gehören der mittlere absolute Fehler (*mean absolute error*, *MAE*) und der Huber-Verlust. Diese sind in TensorFlow (Abadi u. a., 2015) implementiert.

2.8.7. Glorot-Zufallsgenerierung

Beim dem in (Glorot u. Bengio, 2010, S.3) beschriebenen Verfahren werden die Gewichte in einem Intervall basierend auf der Anzahl von Gewichten in der vorhergehenden Ebene generiert. Diese Art der Zufallsgenerierung liefert nach (Simonyan u. Zisserman, 2015) für das darin beschriebene Netzwerk bessere Ergebnisse und die Netzwerke können ohne andere Vorverarbeitungsschritte direkt gelernt werden. Es ist in TensorFlow (Abadi u. a., 2015) das Standardverfahren für die Generierung von Gewichten bei Conv2D-Ebenen.

2.8.8. Optimierungsalgorithmen

Eine Erweiterung des SGD-Algorithmus ist AdaDelta, wobei die Lernrate für konkrete Gewichte über Zeit basierend auf deren aktuellen und bisherigen Gradienten automatisch verringert wird. Ein höherer Gradient sorgt dabei für ein schnelleres Absinken der Lernrate. Adam ist eine andere Erweiterung, die die bisherigen Gradienten analog zum SGD-Moment (siehe 2.7.5 auf Seite 14) verwendet (Lathuilière u. a., 2019, S.4).

2.8.9. ResNet-50

In (He u. a., 2016) wurde festgestellt, dass Netzwerke mit vielen Ebenen Probleme haben, die Identitätsfunktion zu lernen. Aus diesem Grund wurden in diesem Paper Abkürzungen über mehrere Netzwerkebenen eingeführt. Diese bilden die Identitätsfunktion ab, indem zu der eigentlichen Ausgabe des Blocks die ursprüngliche Eingabe addiert wird. ResNet-50 beschreibt eine konkrete darauf basierende Architektur. ResNet-50 besteht aus 16 Blöcken von jeweils drei Conv2D-Ebenen, die über die beschriebene Abkürzung verfügen.

2.9. Ausgewählte Grundlagen der Graphentheorie

In den folgenden Abschnitten wird auf Grundlagen der Graphentheorie in Bezug auf bipartite Graphen, das Matching-Problem, den Netzwerkfluss sowie auf das gewichtete bipartite Matching eingegangen. Darauf folgt die Darstellung des Zuordnungsproblems als ein Minimum-Cost-Flow Problem.

2.9.1. (Bipartite) Graphen und Matchings

Ein Graph $G = (V, E)$ besteht aus einer Menge von Knoten V (*vertices*) und einer Menge von Knoten-Paaren beziehungsweise Kanten E (*edges*). Die Kanten können entweder gerichtet oder ungerichtet sein. Ein Graph ist ungerichtet, wenn für alle Kanten gilt, dass $(i, j) \in E \longrightarrow (j, i) \in E$. Mit Graphen können verschiedene Zusammenhänge dargestellt werden. Zusätzlich ist eine Zuordnung von Kosten zu den jeweiligen Kanten möglich.

Ein Graph G ist bipartit, wenn sich dessen Knotenmenge in zwei disjunkte Teilmengen L und R teilen lässt, sodass alle Kanten jeweils einen Knoten in L und einen in R besitzen.

Ein Matching in einem Graphen ist eine Teilmenge von Kanten $E' \subset E$, wobei sich keine der Kanten in E' auf den gleichen Knoten beziehen (Skiena, 2012, S.145f., S.191, S.218, S.498).

Wird ein Matching in einem Graphen gesucht, dass die Kosten der Kanten minimiert, handelt es sich um ein gewichtetes Matching-Problem (*weighted matching problem*). Ist der betrachtete Graph dabei bipartit und gilt $|L| = |R|$, wird das Problem als ein Zuordnungsproblem (*assignment problem*, *weighted bipartite matching problem*) bezeichnet (Ahuja u. a., 1994, S.33,S.36)(Skiena, 2012, S.498f.).

2.9.2. Netzwerkfluss

Der Netzwerkfluss wird auf einem gerichteten Graphen betrachtet, dessen n Kanten $(i, j) \in E$ eine Kapazität c_{ij} besitzen. In dem Graphen muss dabei jeweils ein Quellknoten s (*source node*) und ein Zielknoten t (*sink node*) vorhanden sein. Das Netzwerkfluss-Problem beschäftigt sich mit der Frage, wie viele Elemente vom Quellknoten zum Zielknoten übertragen werden beziehungsweise fließen können, ohne die Kapazitätsbeschränkungen zu überschreiten. Nachfolgend beschreibt x_{ij} die Anzahl der Elemente, die über die Kante (i, j) fließen. Es gilt damit folgender Zusammenhang:

$$\forall (i, j) \in E.\ 0 \leq x_{ij} \leq c_{ij} \tag{2.22}$$

Die Kanten können zusätzlich zur Kapazität auch um Übertragungskosten d_{ij} pro Element erweitert werden. Des Weiteren lässt sich die Anzahl an zu übertragenden Elementen zwischen Quell- und Zielknoten auf f festlegen. Es muss also folgende Gleichung gelten:

$$\sum_{i=1}^{n} x_{it} = f \tag{2.23}$$

Das Minimum-Cost-Flow Problem beschreibt die Suche nach einer Netzwerkflusszuordnung, die die folgende Summe minimiert:

$$\sum_{(i,j)\in E}^{n} d_{ij} * x_{ij} \tag{2.24}$$

Dabei müssen die entsprechenden Kapazitätsbeschränkungen beachtet werden (Ahuja u. a., 1994, S.2f) (Skiena, 2012, S.509f.). Die Kanten des Graphen können darüber hinaus mit einer Untergrenze u_{ij} erweitert werden, die beschreibt, wie viele Elemente über diese Kante mindestens fließen müssen (Ahuja u. a., 2014, S.5).

In (Han u. a., 2018a) wird zur Lösung des Minimum-Cost-Flow Problems lineare Programmierung verwendet. Alternativ dazu bietet die LEMON-Bibliothek (Egerváry Research Group on Combinatorial Optimization (EGRES), 2014) ein anderes Lösungsverfahren an (siehe Abschnitt 3.7 auf Seite 42).

2.9.3. Darstellung des Zuordnungsproblems als Minimum-Cost-Flow Problem

Das Zuordnungsproblem lässt sich als ein Minimum-Cost-Flow Problem darstellen. Dazu wird ein Fluss $f_{ij} = 1$ zwischen jedem Knoten $i \in L$ und $j \in R$ festgelegt. Die Kapazität c_{ij} jeder Kante $(i, j) \in E$ wird auf 1 begrenzt (Ahuja u. a., 2014, S.7). Es sind also mehrere Quell- und Zielknoten in einem solchen Netzwerkfluss-Graphen vorhanden.

Um dies nun auf jeweils einen Quell- und einen Zielknoten zu reduzieren, lässt sich der Graph durch Hinzufügen eines übergeordneten Quell- und eines übergeordneten Zielknotens ergänzen, die diese ursprünglichen Knoten beliefern oder von denen beliefert werden. Der Gesamtfluss des angepassten Graphen ist somit $f = \sum_{(i,j)\in E} f_{ij}$ und die Teilflüsse werden entfernt. Die Kosten dieser neuen Verbindungen werden auf 0 und die Kapazitäten auf 1 gesetzt. Dadurch wird erreicht, dass sich die Gesamtkosten nicht durch neuen Kanten verändern (Wikipedia, 2019a; Skiena, 2012, S.510).

2.10. Direkte und inverse Kinematik

Starre Körper (*rigid bodies*) können über Gelenke verbunden werden (Siciliano u. Khatib, 2008, S.9f.). Wie auch in (Siciliano u. Khatib, 2008, S.18f.) wird im Folgenden angenommen, dass die Gelenke keine Abstände im Kontaktpunkt aufweisen.

Es existieren verschiedene Gelenkarten, davon sind das Scharnier- und Kugelgelenk[3] in dieser Arbeit von Relevanz. Ein Kugelgelenk lässt sich dabei als eine Serie von Scharniergelenken mit unterschiedlichen Gelenkachsen darstellen (Siciliano u. Khatib, 2008, S.20ff).

Eine der möglichen Repräsentationen ist das in (Siciliano u. Khatib, 2008, S.23f.) beschriebene Verfahren mit vier Parametern a, α, d und θ, durch das sich Scharniergelenke und prismatische Gelenke mit abstrakten Verbindungsstücken (*links*) abbilden lassen. Dabei ist a die Länge der Verbindung. α gibt die Rotation um die Achse entlang dieser Verbindung an. d gibt die Strecke an, um die das Gelenk senkrecht vom Ende des Verbindungsstücks verschoben ist. Der letzte Parameter θ beschreibt den Gelenkwinkel. Daraus resultiert die folgende Transformation (Siciliano u. Khatib, 2008, S.25):

$$T = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & a \\ \sin\theta\cos\alpha & \cos\theta\cos\alpha & -\sin\alpha & -\sin(\alpha)d \\ \sin\theta\sin\alpha & \cos\theta\sin\alpha & \cos\alpha & \cos(\alpha)d \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.25}$$

Um die Position nachfolgender Verbindungen und Gelenke in einem System zu ermitteln, wird die Transformation in der Formel 2.25 mit den Parametern der jeweiligen Gelenke verkettet. Darüber lassen sich andere Körper in Abhängigkeit von den jeweiligen Gelenken positionieren. Jedem Gelenk kann also ein eigenes Koordinatensystem zugeordnet werden. Wird das letzte Verbindungsstück betrachtet, wird das zugrunde liegende Problem der Bestimmung der Position als das *Problem der direkten Kinematik* bezeichnet (Siciliano u. Khatib, 2008, S.26).

Ist die Position des letzten Verbindungsstücks bekannt, aber nicht die Parameter der jeweiligen Gelenke, wird von dem *Problem der inversen Kinematik* gesprochen. Je nach Komplexität des zugrundeliegenden Systems existieren verschiedene Lösungsansätze. Bevorzugt werden Lösungen mit einer geschlossenen Form, da diese schneller sind. Ein Nachteil dabei ist, dass diese Ansätze nicht auf andere Systeme übertragen werden können. Numerische Methoden sind hingegen nicht vom System abhängig, sind jedoch langsamer und können nicht immer alle möglichen Lösungen

[3] Im Englischen werden das Scharniergelenk als *revolute joint* oder *hinge joint* und das Kugelgelenk als *spherical joint* bezeichnet.

finden. Eine iterative Herangehensweise ist die Formulierung des Problems als ein Optimierungsproblem, wobei von initialen Parameterwerten ausgehend nach einer Lösung gesucht wird (Siciliano u. Khatib, 2008, S.28f.). Der Optimierungsalgorithmus Levenberg-Marquardt kann hierfür verwendet werden (Han u. a., 2018a, S.4).

2.11. Modellierung der Hand

Die menschliche Hand besteht aus verschiedenen Gelenken. Die hier relevanten Gelenke werden in der Abbildung 2.2 dargestellt. Dabei werden die Finger nach (Lee u. Kunii, 1995) nummeriert.

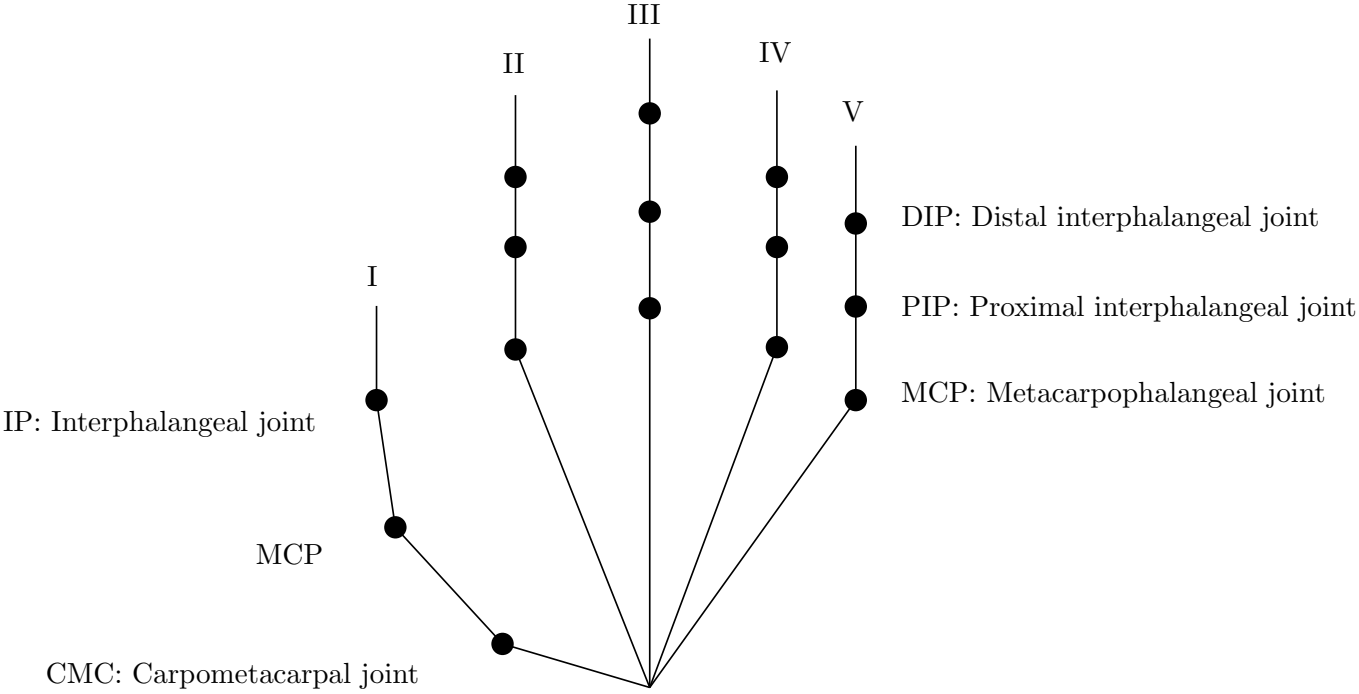

Abbildung 2.2.: Eine abstrakte Darstellung einer Hand mit medizinischen Bezeichnungen für die Gelenke aus (Albrecht u. a., 2003), die auch bei den Trainingsdaten aus (Han u. a., 2018a) verwendet werden. Die Finger werden vom Daumen aus mit I bis V nach (Lee u. Kunii, 1995) nummeriert.

Es gibt verschieden Möglichkeiten die Hand vereinfacht zu modellieren. Dabei wird an verschiedenen Stellen auf Freiheitsgrade (*degrees of freedom*, *DOFs*) verzichtet. (Rijpkema u. Girard, 1991) verwenden dabei für die Finger II bis V vier Freiheitsgrade, wobei der Finger an den Gelenken MCP, PIP sowie DIP gebeugt werden kann und im MCP-Gelenk dazu noch eine Seitwärtsbewegung möglich ist. Im Daumen haben die Gelenke CMC und MCP jeweils einen zweiten DOF. Die Konfiguration der Finger hat also 21 Freiheitsgrade. In diesem Paper wird zudem noch für II bis V eine geschlossene Lösung für die Bestimmung der Position der Fingerspitzen vorgestellt.

(Sanso u. Thalmann, 1994) verwenden analog zu (Rijpkema u. Girard, 1991) jeweils ein DOF in den DIP- und PIP-Gelenken der Finger II-V und dem IP-Gelenk im Daumen. Die MCP-Gelenke von II-V haben analog dazu zwei Freiheitsgrade. Der Daumen in diesem Modell kann um eine durch den Zeigefinger gebildete Achse im CMC-Gelenk rotieren. Dieses Modell besitzt ebenfalls 21 Freiheitsgrade.

Das Modell aus (Schröder u. a., 2014) ist analog zu (Rijpkema u. Girard, 1991) aufgebaut, das CMC-Gelenk des Daumens unterstützt jedoch nur eine Seitwärtsbewegung. Es wird für die Rekonstruktion von Handkonfigurationen beim Motion-Tracking verwendet. Zusammen mit der globalen

Position und Rotation besitzt dieses Modell 26 Freiheitsgrade. Als Basis für das Handmodell wird dabei die Handwurzel verwendet.

Im Kontrast dazu wird in (Albrecht u. a., 2003) ein komplexeres Handmodell mit 3 DOF in den MCP-Gelenken von II bis V und im CMC des Daumen. Des Weiteren wurde jeweils zwei DOF für die Finger IV und V für eine Bewegung an der Handwurzel hinzugefügt. Dieses Modell soll eine natürlichere Darstellung von Handbewegungen erreichen, besitzt jedoch eine deutliche höhere Anzahl an Freiheitsgraden. Ein anderes komplexeres Handmodell ist in (Lee u. Kunii, 1995) zu finden. Dabei werden für die Gelenke der Finger II-V analog zu (Rijpkema u. Girard, 1991) vier DOFs und im MCP- sowie CMC-Gelenk des Daumens zwei DOFs verwendet. Als Basis wird hier die Handwurzel verwendet, die zusätzlich noch 6 Freiheitsgrade für eine globale Position und Rotation zum Modell beiträgt. Dieses Modell hat also 27 Freiheitsgrade. Andere Modelle, wie das aus (Aristidou u. Lasenby, 2010), haben Freiheitsgrade für die Bewegung des Mittelfingers in dem Gelenk an der Handwurzel.

Da die Bewegungen der Gelenke bei einer echten Hand begrenzt sind, werden unter anderem in (Lin u. a., 2000) Beschränkungen für Gelenke eingeführt, die in der Tabelle 2.1 für die Finger II-V dargestellt sind. Dabei wurde unter anderem die Seitwärtsbewegung im MCP-Gelenk des Mittelfingers verboten, wodurch die DOFs für das Handmodell in diesem Paper reduziert werden konnten.

Gelenk	**Bewegung**	**Intervall**
MCP (II-V)	Beugung	$[0°, 90°]$
MCP (II, IV, V)	Seitwärtsbewegung	$[0°, 90°]$
MCP (III)	Seitwärtsbewegung	0
PIP (II-V)	Beugung	$[0°, 110°]$
DIP (II-V)	Beugung	$[0°, 90°]$

Tabelle 2.1.: Die Intervalle für Gelenkbeschränkungen aus (Lin u. a., 2000).

2.12. Linear Blend Skinning

Nach (Kavan, 2014) wird bei der Computeranimation die Verformung eines Objekts durch zum Beispiel Knochen eines virtuellen Skeletts als *Skinning* bezeichnet. Ein solches Objekt kann die Haut eines animierten Charakters sein. Der Begriff umfasst darüber hinaus andere Objekte, die der Verformung zugrunde liegen können, aber hier jedoch nicht relevant sind. Ein bekannter Algorithmus dafür ist das *Linear Blend Skinning*, wobei das zu verformende Objekt über Gewichte mit verschiedenen Knochen assoziiert ist (Kavan, 2014).

2.13. Quadratisches Mittel

Das quadratische Mittel (*RMS*) einer Menge von Werten x der Größe[4] m ist die Wurzel eines Stichprobenmittelwerts, bei dem die Werte zuvor quadriert worden sind, und ist folgendermaßen definiert (Wikipedia, 2019c):

$$\sqrt{\frac{\sum_{i=1}^{m} x_i^2}{m}} \tag{2.26}$$

[4]Bei dem Bezeichner der Größe wurde sich an den in Abschnitt 2.4.2 auf Seite 12 orientiert.

2.14. Deep Labels

Die Autoren von (Han u. a., 2018a) haben ein Verfahren zum Labeling von Markern, die zum Beispiel über OptiTrack erfasst werden können, entwickelt. Dabei wird ein Handschuh mit 19 passiven Markern verwendet. Das Zentrum ihres Systems bildet ein Convolutional Neural Network. Das Problem dabei war, dass 3D-Convolutions für eine Echtzeitverarbeitung laut den Autoren nicht performant genug sind. Anstatt die Markerpositionen im Raum direkt zu betrachten, projiziert deren System diese zunächst auf ein Bild und behält dabei die Tiefeninformationen bei. Darauf können dann die in der Bildverarbeitung üblichen zweidimensionalen Convolutions verwendet werden. Das daraus resultierende Bild wird nachfolgend als Tiefenbild bezeichnet. Im Rahmen von (Han u. a., 2018a) wurden zum Lernen eines solchen Netzwerks synthetische Trainingsdaten mit verschiedenen Handkonfigurationen generiert.

Die Projektion ist orthographisch und die optische Achse der zugehörigen Kamera wird auf das Zentrum der Punktwolke von Markern gerichtet. Dabei wird mit der Kamera uniform so gezoomt, dass sich alle Marker im Bild mit einem Sicherheitsabstand zum Rand von jeweils 10% befinden. Die Tiefenwerte der Marker werden daraufhin aus der durch die optische Achse gegebenen Blickrichtung in das Intervall $[0.1, 1]$ normalisiert. Diese normalisierte Tiefeninformation des jeweiligen Markers wird dann bei seiner Projektion als Pixelwert verwendet.

Jeder Marker wird dabei über Splatting (siehe Abschnitt 2.3 auf Seite 10) auf das resultierende 52×52 große Tiefenbild projiziert.

Die Auswahl der optischen Achse der Kamera bei der Echtzeitverarbeitung basiert auf zehn zufälligen Achsen. Dabei wird für jede Achse ein Tiefenbild projiziert. Die daraus resultierenden zehn Tiefenbilder werden dann entsprechend der räumlichen Verteilung der projizierten Marker darauf bewertet. Dieses Maß wird in (Han u. a., 2018a) als *spatial spread* bezeichnet. Diese Verteilung wird dabei basierend auf den zweidimensionalen Koordinaten der Marker auf dem Tiefenbild untersucht. Dabei wird zunächst eine Kovarianzmatrix (siehe Abschnitt 2.4 auf Seite 11) dieser Marker-Koordinaten berechnet, um den Zusammenhang zwischen den X- und Y-Koordinaten zu untersuchen. Danach werden die Eigenwerte dieser Matrix bestimmt. In (Han u. a., 2018a) wird dann das Tiefenbild mit den höchsten Eigenwerten ausgewählt. Beim Training wird stattdessen eine optische Achse zufällig ausgewählt.

Das daraus resultierende Tiefenbild wird dann über ein Convolutional Neural Network verarbeitet. Der Aufbau dieses Netzwerks ist in Tabelle 2.2 zu finden.

Die Ausgabe des Netzwerks ist ein Tensor mit 19 Markerpositionen im Format 19×3. Aus der Reihenfolge der Marker in der Ausgabe lässt sich auf die Position des Markers auf der Hand und somit auf das jeweilige Label schließen. Nun müssen jedoch die jeweiligen Markern der Ausgabe den Marker aus der Eingabe zugeordnet werden. Das Problem dieser Zuordnung wird als ein gewichtetes bipartite Matching-Problem zwischen den n_x ursprünglichen Markern x_i und den n_y vom Netzwerk vorhergesagten Markern y_j dargestellt, mit $1 \leq i \leq n_x$ und $1 \leq j \leq n_y$. Das gesuchte Matching $\mathcal{M}$ muss dabei folgende Summe minimieren:

$$\sum_{j}^{n_j} ||y_j - x_{\mathcal{M}(j)}||_2 \tag{2.27}$$

Dieses Matching-Problem wird als Minimum-Cost-Flow Problem mit den Kantenkosten basierend auf der paarweisen Distanz der Marker betrachtet und über Linear Programming gelöst.

Bei der Rekonstruktion der Handkonfiguration aus diesen gelabelten Markern wird ein Handmodell mit 26 Freiheitsgraden verwendet. Dabei sind 20 Freiheitsgrade durch die möglichen Fingerkonfigurationen und sechs DOFs durch die globale Position und Rotation gegeben. Das Handmodell besitzt dabei eigene Marker, die durch Linear Blend Skinning an die jeweilige Handkonfiguration

Ebenentyp	Filtergröße
Conv2D	$64 \times 3 \times 3$
BatchNormalization	-
ReLU	-
Conv2D	$64 \times 3 \times 3$
BatchNormalization	-
ReLU	-
Maxpool	2×2
Conv2D	$128 \times 3 \times 3$
BatchNormalization	-
ReLU	-
Conv2D	$128 \times 3 \times 3$
BatchNormalization	-
ReLU	-
Conv2D	$128 \times 3 \times 3$
BatchNormalization	-
ReLU	-
Maxpool	2×2
Reshape	-
FC	2048×10368
ReLU	-
FC	2048×57
Reshape	-

Tabelle 2.2.: Der Aufbau des neuronalen Netzwerks aus (Han u. a., 2018a). Die Ein- und Ausgabedimensionen sind in der Tabelle 3.2 auf Seite 40 im Abschnitt 3.6.3 zu finden.

angepasst werden. In (Han u. a., 2018a) wird die folgende Formel dazu verwendet:

$$LBS(\theta, m_i, w) = \sum_j w_{ji} * T_j(\theta) * \left(T_j^{rest}\right)^{-1} * m_i \tag{2.28}$$

Dabei beschreibt i den Index des aktuell betrachteten Markerpaares, θ die Parameter des Handmodells und LBS die Funktion, die das Linear Blend Skinning für einen Marker der Hand m_i abhängig vom Handmodell durchführt. x_i beschreibt den jeweiligen Marker des aktuell betrachteten Frames. Die Menge w enthält die Gewichte für jedes Gelenk-Marker-Paar. T_j beschreibt die aus der aktuellen Handkonfiguration ermittelte Transformation für das betrachtete Gelenk und T_j^{rest} die Transformation dieses Gelenks in Ruheposition.

Zur Rekonstruktion wird die Summe der paarweisen Differenzen zwischen den n zugeordneten Markern x_i des aktuellen Frames und den n Markern m_i des Handmodells minimiert. Es wird folgender Fehlerterm E_{IK} verwendet:

$$E_{IK} = \sum_{i=1}^{n} ||LBS(\theta, m_i, w) - x_i||_2 \tag{2.29}$$

Als Optimierungsalgorithmus wird Levenberg-Marquardt verwendet. Die Parameter des Handmodells werden zu Beginn mit der Ruheposition und bei aufeinander folgenden Frames durch die Parameter des vorherigen Frames initialisiert. Es wurden fünf verschieden große Handmodelle entwickelt. Als Maß für die ermittelte Konfiguration wird der RMS-Fehler der Distanz auf den jeweiligen Markerpaaren betrachtet. überschreitet dieser Fehler 8 mm, wird das aktuelle Frame verworfen.

Es wurden verschiedene Aufnahmen von echten Daten aus OptiTrack betrachtet. Dies umfasste die Bewegung von einer Hand, von zwei Händen, sowie die Hinzunahme von anderen mit Markern bestückten Objekten, wie einen Stift oder einen Controller.

Damit das Netzwerk trotz zusätzlichen Objekten ein gutes Ergebnis liefert, wurden beim Lernen zudem Trainingsdaten mit fehlenden oder zusätzlich Markern generiert.

Das Labeling-Verfahren erreichte laut (Han u. a., 2018a) sehr gute Ergebnisse. Bei der Betrachtung von zuvor beschriebenen aufgenommenen Daten, wurden bei einer Hand alle Frames richtig gelabelt. Bei der Variante mit zwei Händen wurden die Marker in 99.24% aller Frames richtig zugeordnet. Bei der Stift- und Controller-Variante sind es 93.73% beziehungsweise 98.26%. Auf synthetischen Testdaten wurden ähnliche Ergebnisse erreicht.

Das beste verwendete Netzwerk sowie die synthetischen Trainingsdaten wurden von den Autoren veröffentlicht. Die in (Han u. a., 2018a) generierten Daten mit fehlenden oder zusätzlichen Markern werden im Gegensatz dazu jedoch nicht mitgeliefert.

2.14.1. Optimierung der Markerpositionen des Handmodells

Um das Handmodell an den Nutzer anzupassen, werden die Positionen der Marker des Handmodells optimiert. Als Grundlage dafür werden verschiedene Handpositionen, wie eine Berührung des Daumens mit dem Zeigefinger, betrachtet. Dadurch wird erreicht, dass der Kontakt und Kontaktpunkt bei solchen Positionen beim virtuellen Handmodell der echten Hand entspricht.

2.15. JSON-Format

Das JSON-Format ist ein einfaches textuelles Format zum Datenaustausch. In JSON können verschiedene Daten wie Objekte, Arrays, Zeichenketten oder Zahlen dargestellt werden. Objekte und Arrays können wiederum Daten aller unterstützen Typen enthalten. Dadurch können komplexe Daten dargestellt werden (Ecma International, 2017).

Mehrere Programmiersprachen, wie Python (Python Software Foundation, 2019a) oder Go (The Go Authors, 2019), haben in ihrer Standardbibliothek entsprechende Schnittstellen zur Verarbeitung von JSON-Daten.

2.16. PGM und PLY

Das PGM-Format ist ein einfaches textbasiertes Bildformat für Graustufen-Bilder. Die Spezifikation ist unter (Poskanzer, 2016) zu finden, zusätzlich dazu wurde bei der Implementierung (Wikipedia, 2020) herangezogen. In dieses Format lassen sich ohne großen Implementierungsaufwand Bilder exportieren.

Das PLY-Format dient dazu dreidimensionale Szenen und Objekte darzustellen. Es ist wie das PGM-Format textbasiert (Wikipedia, 2019b). Auch in dieses Format ist der Export sehr einfach zu implementieren.

2.17. Singulärwertzerlegung

Zusätzlich zur Eigenwertzerlegung einer Matrix $\mathbf{A}$ existiert die sogenannte Singulärwertzerlegung $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$ in drei Matrizen. Dabei sind $\mathbf{U}$ und $\mathbf{V}$ orthogonal. Die Matrix $\mathbf{D}$ ist diagonal und die Werte werden als Singulärwerte bezeichnet (Goodfellow u. a., 2016, S.42).

2.18. Ermitteln einer Transformation zwischen zwei Punktwolken

In (Arun u. a., 1987) wird ein Verfahren beschrieben, dass die Translation T und Rotation R zwischen zwei gleich großen Punktwolken p und p' bestimmt. Die Größe dieser Punktwolken wird nachfolgend als N bezeichnet. Es werden genau die T und R gesucht, die die Differenz zwischen der durch R sowie T transformierten ersten Punktwolke und der zweiten Punktwolke minimieren. Als Abstandsmaß wird die Summe der paarweise quadratischen Distanzen verwendet.

Nach (Arun u. a., 1987) kann die Translation getrennt betrachtet werden. Dazu werden beide Punktwolken zunächst um den Ursprung zentriert. Dies zentrierten Punktwolken werden nachfolgend als q und q' bezeichnet. Dann muss nur noch die folgende Summe minimiert werden:

$$\sum_{i=1}^{N} ||q'_i - Rq_i||^2 \tag{2.30}$$

Dazu wird eine zusätzliche Matrix H aus den zentrierten Punktwolken folgendermaßen bestimmt:

$$H = \sum_{i=1}^{N} q_i * q_i'^{\top} \tag{2.31}$$

Aus der Singulärwertzerlegung $H = UDV^{\top}$ dieser Matrix kann nun die Rotation $R = VU^{\top}$ berechnet werden, die die Summe in Formel 2.30 minimiert. Falls die Determinante $det(R)$ dieser Matrix negativ ist, trifft der sogenannte Reflektionsfall auf. Ist einer der Singulärwerte von U gleich Null, muss die Rotation stattdessen aus $R' = V'U^{\top}$ ermittelt werden. Dabei ist V' die Matrix V mit umgekehrten Vorzeichen bei den Einträgen der dritten Spalte. Sind alle Singulärwerte ungleich Null, ist zu viel Rauschen vorhanden beziehungsweise die Punktwolken unterscheiden sich zu stark. Die Translation kann dann mit der Rotation und den Zentren c_p und $c_{p'}$ der ursprünglichen Punktwolken durch $T = c_{p'} - Rc_p$ bestimmt werden (Arun u. a., 1987).

2.19. Abhängigkeiten der Implementierung

2.19.1. C++

C++ ist eine plattformübergreifende Programmiersprache, die von verschiedenen Compilern unterstützt wird und vielfach dokumentiert ist (Free Software Foundation, 2019; Microsoft, 2019; cppreference.com, 2019).

CMake

CMake (Kitware, Inc. and Contributors, 2019) ist ein Werkzeug, um aus einer Beschreibung des aktuellen Builds Skripte zum Kompilieren des Projekts oder IDE-Projekte zu generieren.

Das PIMPL-Idiom

Häufig müssen für private Attribute in Klassendeklarationen in Header-Dateien zusätzliche Header inkludiert werden. Diese zusätzlichen Header sind dabei aber nur für die Implementierung und nicht den endgültigen Nutzer relevant. Dies führt dazu, dass wenn diese Klassen und Methoden in einer unabhängigen Quellcode-Datei verwendet werden, sehr viele Header unnötig inkludiert und somit verarbeitet werden müssen.

Um diesem Problem vorzubeugen, existiert das sogenannte PIMPL-Idiom. PIMPL steht dabei für „pointer to implementation“. Dabei werden alle Attribute der ursprünglichen Klasse in eine

neue Klasse in der zur Klassendeklaration gehörenden Quellcode-Datei ausgelagert. Die neue Klasse enthält als einziges Attribut einen Zeiger auf diese neue Klasse. Dabei wird die neue Klasse in der Header-Datei deklariert, aber nicht definiert. Die ursprüngliche Klasse leitet dann alle Methodenaufrufe auf die neue Klasse weiter.

Die privaten Attribute sind dann in der Quellcode-Datei versteckt und die dazugehörigen Header werden nicht mehr vom Nutzer der Header-Datei indirekt inkludiert (Meyers, 2015, S.147ff.).

Eigen

Eigen ist eine C++-Bibliothek, die Unterstützung für Matrizen und Vektoren sowie Operationen darauf anbietet. Dies umfasst auch die Eigen- und Singulärwertzerlegung (Jacob u. Guennebaud, 2018a,b). Auf diese Zerlegungen wird in den Abschnitten 2.4.1 auf Seite 11 und 2.17 auf Seite 23 eingegangen. Die Einbindung von Eigen in CMake wurde analog zu Google Test nach (Google, 2019b, README.md) durchgeführt.

Irrlicht

Irrlicht ist eine C++-Bibliothek für die Spiele-Entwicklung und bietet die Möglichkeit dreidimensionale Szenen einfach anzuzeigen. Dabei werden verschiedene Formate für 3D-Modelle unterstützt (Gebhardt, 2016). Es musste für eine Windows-Kompatibilität ein eigene CMake-Build-Beschreibung entwickelt werden, diese basiert auf der originalen Build-Beschreibung aus (Gebhardt, 2016). Dabei wurde (Arias, 2018) als Dokumentation hinzugezogen.

NatNet

NatNet ist eine C++-Bibliothek zur Netzwerkkommunikation mit der „OptiTrack Motive“-Software. Dabei wird eine Übertragung von Markerpositionen unterstützt (NaturalPoint, Inc., 2019).

VRPN

Virtual Reality Peripheral Network (VRPN) ist eine C++-Bibliothek zur Netzwerkkommunikation zwischen verschieden Virtual Reality (VR) Geräten. Dadurch können zum Beispiel verschiedene Computersysteme, an die jeweils unterschiedliche Geräte angeschlossen sind, miteinander über das Netzwerk verbunden werden und deren Daten dann auf einem anderen System verarbeitet werden. Die Kommunikation läuft dabei über Nachrichten, die durch VRPN codierte sind, ab. Die Basisklasse für die Kommunikation ist dabei `vrpn_BaseClass` (VRPN-Entwickler, 2019).

Von dieser VRPN-Basisklasse abgeleitete Klassen stellen die geerbte Methode `mainloop` bereit. Diese muss in jedem Fall in der Mainloop des Programms, also kontinuierlich, ausgeführt werden (VRPN-Entwickler, 2019, Getting Started). Tutorials zu VRPN sind in (The VR Geeks Association, 2010, 2011) zu finden.

LEMON

LEMON (Egerváry Research Group on Combinatorial Optimization (EGRES), 2014) ist eine Bibliothek, die Datenstrukturen und Algorithmen zur Arbeit mit Graphen bereitstellt. Dies umfasst auch mehrere Solver des Min-Cost-Flow Problems.

Boost

Die Boost-Bibliothek (Boost, 2018) bietet einen breiten Rahmen an verschiedenen Teilbibliotheken an. Dies umfasst unter anderem die Verarbeitung von Zeichenketten, den Umgang mit dem Dateisystem und die Verarbeitung von Kommandozeilenoptionen.

Google Test und Google Benchmark

Google Test (Google, 2019b) ist eine Bibliothek zur Entwicklung von unter anderem Unit-Tests. Google Benchmark (Google, 2019a) erweitert diese um Tools zur Entwicklung von Benchmarks.

Zur Einbindung von Google Test in das Projekt wurde CMake, wie es in (Google, 2019b, README.md) beschrieben ist, verwendet.

2.19.2. Python

Python ist eine Skriptsprache mit einem dynamischen Typsystem, dessen Quellcode durch einen Interpreter ausgeführt wird. Pakete beziehungsweise Abhängigkeiten werden üblicherweise global installiert. Das Modul *venv* bietet die Möglichkeit in einem Verzeichnis, zum Beispiel in einem Unterverzeichnis des aktuellen Projekts, eine Python-Umgebung zu erzeugen. Diese Python-Umgebung enthält Verknüpfungen auf den gewählten Interpreter und bietet die Möglichkeit Abhängigkeiten lokal in diesem Verzeichnis zu installieren. Besitzen Abhängigkeiten ausführbare Bestandteile, wie zum Beispiel JupyterLab, werden in der Umgebung ebenfalls Verknüpfungen darauf angelegt und können dann genutzt werden. Der Python-Interpreter lässt sich in andere Programme einbetten (Python Software Foundation, 2019b).

NumPy

NumPy ist eine Python-Bibliothek, die mehrdimensionale Arrays wie Matrizen oder Vektoren und verschiedene Operationen darauf anbietet. Diese Arrays werden im Folgenden als Numpy-Arrays bezeichnet. Dabei werden viele Operationen aus Python in unter anderem C ausgelagert. Dadurch sind diese schneller als eine reine Python-Implementierung. Hinzu bietet NumPy eine C-Schnittstelle, um auf Numpy-Arrays von C oder C++ aus zuzugreifen (The SciPy community, 2019).

Matplotlib

Matplotlib ist eine Python-Bibliothek, die verschiedene Tools zum Anzeigen von zwei- oder dreidimensionalen Graphen bereitstellt. Dabei können diese Graphen in zum Beispiel JupyterLab-Notizbücher eingebettet werden (Hunter u. a., 2019). Beim Erzeugen von Graphen mit Fehlerbalken wurde (Stack Overflow, 2015) hinzugezogen.

JupyterLab

JupyterLab ist eine in Python entwickelte Anwendung, die die Möglichkeit bietet, interaktiv Python-Skripte in sogenannten Notizbüchern auszuführen. Dabei werden (Zwischen-)Ergebnisse direkt ausgegeben und zusätzlich zu den Eingaben gespeichert. Dies bietet besonders bei der Entwicklung von Prototypen Vorteile. Des Weiteren können von Matplotlib erstellte Grafiken direkt in den Notizbüchern eingebettet angezeigt werden (Project Jupyter, 2018). Es wird die Matplotlib-Erweiterung (Matplotlib Contributors, 2020) für JupyterLab benutzt.

2.19.3. Lua

Lua ist eine Skriptsprache die zum Einbetten in bestehende Systeme, die zum Beispiel in C oder C++ implementiert sind, entwickelt wurde. Die Sprache besitzt die üblichen Grunddatentypen wie Zahlen oder Zeichenketten. Listen und Wörterbücher (*dictionaries*, *maps*) werden unter sogenannten Tabellen zusammengefasst. Funktionsparameter und -rückgabewerte werden zwischen Lua und C/C++ über einen Stack übertragen. Das heißt, dass bei einem Funktionsaufruf zunächst der Funktionsname und dann die Parameter auf den Stack gelegt werden müssen. Bei eingebauten

Funktionen, wie dem Hinzufügen von Elementen zu einer Tabelle, wird auf den Funktionsnamen verzichtet (Ierusalimschy u. a., 2012).

Neben der offiziellen Implementierung (Ierusalimschy u. a., 2019) existiert eine alternative Laufzeitumgebung LuaJIT (Pall, 2018). Diese bietet die gleiche C/C++-Schnittstelle zum Einbetten, unterstützt jedoch nur Lua bis einschließlich Version 5.1.

LuaTorch

LuaTorch (Collobert u. a., 2019a,b) ist eine Lua-Bibliothek, die unter anderem verschiedene Datenstrukturen und Algorithmen für die Entwicklung von neuronalen Netzwerken bereitstellt. Dabei wird eine Ausführung dieser auf der GPU unterstützt.

cudnn.torch

Die Bibliothek cudnn.torch (Chintala, 2017) bietet eine Anbindung an die cuDNN-Bibliothek von NVIDIA an, um LuaTorch-Netzwerke über die GPU laufen zu lassen.

2.19.4. TensorFlow

TensorFlow ist eine Python-Bibliothek, die analog zu LuaTorch verschiedene Datenstrukturen und Algorithmen zur Arbeit mit neuronalen Netzwerken bereitstellt. Diese können sowohl auf der CPU als auch auf der GPU verwendet werden. Dabei ist die übergeordnete Programmierschnittstelle Keras integriert (Chollet u. a., 2019; Abadi u. a., 2015). Im Folgenden wird kurz auf das Erzeugen von Netzwerken mit Keras eingegangen.

Erzeugen von Netzwerken mithilfe der Keras-Schnittstelle

TensorFlow bietet einen abstrakten Tensor an, der in der Keras-Schnittstelle die Ein- und Ausgabe von Netzwerk-Ebenen ohne konkrete Daten repräsentieren kann. Dabei können die Ebenen mit der abstrakten Ausgabe einer anderen Ebene aufgerufen werden. Dieser Aufruf liefert wiederum eine solche abstrakte Ausgabe. Dadurch können mehrere Ebenen verkettet und im Endeffekt ein Graph aus Netzwerkebenen aufgebaut werden. Unter der Spezifizierung der ursprünglichen Eingabe und der endgültigen Ausgabe kann dann ein Netzwerk erzeugt werden, das in Keras als *Model* bezeichnet wird (Chollet u. a., 2019; Abadi u. a., 2015).

3. Implementierung

3.1. Systemarchitektur

3.1.1. Kompilierung der Implementierung

Die Implementierung wird in mehreren Stufen kompiliert. Vor dem Start des Build-Vorgangs muss das Python-Skript *prepare_hand_tracking.py* ausgeführt werden. Dies bereitet eine Python-Umgebung mit den passenden Abhängigkeiten und die für die eigentliche Implementierung benötigten Bibliotheken vor. Als eigentliches Build-System wird dann CMake verwendet, da es plattform- und IDE-unabhängig ist.

3.1.2. Graphische Anzeige

Um Pipeline-Ergebnisse anzuzeigen, wurde mit Irrlicht eine graphische Oberfläche entwickelt. Dabei wird die Anzeige von Eingabe-Markern, Gelenken und den Markern des Handmodells durch Sphären realisiert. Eingabe-Marker sind dabei rot und Gelenke grau gefärbt. Die Marker des Handmodells haben ebenfalls eine graue Färbung, sind jedoch im Vergleich zu den Gelenken herunterskaliert. Beim Start lässt sich einstellen, dass die Pipeline-Ergebnisse bei der Anzeige zentriert werden, was die Betrachtung erleichtert.

Diese Oberfläche ermöglicht es, Markerdaten über VRPN oder NatNet zu empfangen. Dies ist in Abschnitt 3.13 auf Seite 62 beschrieben.

3.1.3. Logging

An verschiedenen Stellen in der Implementierung werden Log-Meldungen ausgegeben. Dazu wird ein einfaches Makro verwendet. Dieses Makro erwartet neben der Nachricht den Typ der Meldung, also ob es sich um einen fatalen Fehler, nicht fatalen Fehler, eine Warnung, eine einfache Information oder eine Debug-Meldung handelt. Soll die Ausführung an bestimmten Bereichen verfolgt werden, existiert noch ein weiterer Typ *trace*. Der Typ der Meldung kann auch als Dringlichkeitsstufe verstanden werden. Die Nachricht muss sich dabei an einen STL-*ostream* anhängen lassen und einzelne Elemente können mit dem <<-Operator verknüpft werden. Dadurch werden bereits bekannte Formatierungsanweisungen in der Nachricht unterstützt.

Das Makro expandiert zu einer Bedingungs-Anweisung, in der geprüft wird, ob die Dringlichkeitsstufe der Nachricht hoch genug ist, um angezeigt zu werden. Der Nachrichten-Ausdruck wird an einen `std::ostringstream` angehängt. Es werden die Präprozessordefinitionen `__LINE__` und `__FILE__` verwendet, um die Zeile und Datei zu ermitteln, in der das Makro aufgerufen wird (siehe Free Software Foundation, 2020).

Die Ideen ein Makro zum Loggen zu verwenden, die Formatierungsanweisungen in der Nachricht und die verschiedenen Dringlichkeitsstufen basieren auf (Boost, 2018, Log-Modul). Im Gegensatz dazu ist der Formatierungsausdruck jedoch ein Argument des Makros und das Makro liefert kein Objekt, auf den ein <<-Operator angewendet werden kann. Auf eine direkte Verwendung dieser Bibliothek wurde verzichtet, da die Anpassung der Log-Meldungen für einen einfachen Anwendungszweck komplexer als die Neuentwicklung war.

3.1.4. Debug-Output in der Verarbeitungspipeline

Zu der Labeling-Pipeline lassen sich verschiedene Debug-Adapter hinzukonfigurieren. In diesem Fall werden Informationen zu verschiedenen Teilschritten wie der Projektion oder der Netzwerkausgabe an diesen Adapter weitergereicht. Die Debug-Ausgaben können durch diese Adapter entweder in die Konsole, eine Verzeichnisstruktur mit verschiedenen Dateien für die unterschiedlichen Schritte oder in eine einzelne JSON-Datei ausgegeben werden. Letzteres ist nützlich zur Betrachtung der Ergebnisse über JupyterLab-Notebooks. Dies ist in *debug_output.h* und *debug_output.cpp* sowie *matching_debug_output.h* und *matching_debug_output.cpp* implementiert.

3.1.5. Konfiguration der Verarbeitungspipeline

Beim Erzeugen einer Pipeline können verschiedene Parameter konfiguriert werden. Dies umfasst unter anderem die Strategie zum Generieren fester optischer Achsen (siehe Abschnitt 3.8 auf Seite 44) und den Rand bei Projektionen. Der Wechsel zwischen ursprünglichem und angepasstem Verfahren zur Generierung optischer Achsen muss jedoch direkt im Quellcode der Pipeline geschehen.

3.1.6. Verarbeitungspipeline

Die in Abschnitt 2.14 auf Seite 21 beschriebene und im Rahmen dieser Arbeit reimplementierte Pipeline zum Labeling von Markern und zur Rekonstruktion der Handkonfiguration befindet im Teilprojekt *deep_labeling*. Die Implementierung des Labelings wird dabei in den Abschnitten 3.4 (Auswahl der optischen Achse), 3.5 (Projektion), 3.6 (Verwenden von Neuronalen Netzwerken) und 3.7 (Lösen des Matching-Problems) beschrieben. Die dabei entwickelten Erweiterungen sind in den Abschnitten 3.8 und 3.9 zu finden.

3.1.7. TensorFlow-Version

Im Rahmen der Implementierung wurde TensorFlow in der Version 1.13.1 mit CUDA (NVIDIA, 2018) in der Version 10.0 und cuDNN (NVIDIA, 2019) in der Version 7.6.5 verwendet.

3.1.8. Sonstiges

Verwendung von JupyterLab-Notebooks

Im Rahmen dieser Masterarbeit wurden verschiedene JupyterLab-Notebooks verwendet, um Ergebnisse darzustellen, neue Netzwerke zu lernen und Prototypen zu entwickeln.

`nodiscard`-Attribut

Einige Strukturen und Funktionen sind mit dem C++-Attribut `nodiscard` gekennzeichnet. Wird die Struktur oder der Rückgabewert verworfen, ohne mindestens einmal weiterverwendet worden zu sein, gibt der Compiler eine Warnung aus. Da dieses Attribut nur ab C++17 verfügbar ist, wird stattdessen ein Makro verwendet, was durch das Attribut ersetzt wird. Unterstützt der Compiler nur ältere C++-Versionen, wird das Makro ohne Wert definiert und die Aufrufe werden somit durch *nichts* ersetzt (cppreference.com, 2019; Microsoft, 2019).

Anzeige von Projektionen

Nach der Projektion lässt sich das System anhalten, um die aktuellen Projektionen anzuzeigen. Dies ist nur beim ursprünglichen Verfahren für optische Achsen möglich. Dafür muss dieses entsprechend konfiguriert sein. Die Anzeige wird durch Matplotlib über Python durchgeführt. Auf die

Kommunikation zwischen C++ und Python wird in Abschnitt 3.6.4 auf Seite 40 genauer eingegangen.

Verwendung des PIMPL-Idioms

An verschiedenen Stellen wird im Projekt das in Abschnitt 2.19.1 auf Seite 24 beschriebene PIMPL-Idiom verwendet. Ein Beispiel dafür sind die Pipeline für das Labeling der Marker, die andere Header des Projekts für die privaten Attribute und die Implementierung benötigt, und einige Klassen, die für Attribute Python-Header importieren. Letztere enthalten viele Definitionen im globalen C++-Namespace und durch die Verwendung von PIMPL wird verhindert, dass diese Definitionen beim Nutzer auftauchen.

Mathematische Konstanten unter Windows

Unter Windows tritt das Problem auf, dass die mathematische Konstante π (`M_PI` in C++) Standardmäßig nicht in dem Math-Headern definiert ist. Teilweise ist das Problem durch eine manuelle Definition dieser Konstanten und teilweise durch das Hinzufügen von `_USE_MATH_DEFINES` vor dem Import des Math-Headers nach (Stack Overflow, 2011) gelöst worden.

Warnungen in Headern von Bibliotheken

Um Warnungen beim Kompilieren von Headern aus Abhängigkeiten zu verhindern, wurden diese Header nach (Kitware, Inc. and Contributors, 2020, Issue #17904) als System-Header gekennzeichnet.

Unit-Testing

Einzelne Komponenten wurden durch Unit-Testing mit Google Test (Google, 2019b) geprüft. Einzelne Funktionen wurden dabei auf ihre Eigenschaften hin mit RapidCheck (Eriksson, 2019) getestet. Beim Testen der JSON-Exporte wurde das erste Beispiel aus Kapitel 13 von (Internet Engineering Task Force (IETF), 2014) verwendet.

3.2. Import von Tracking-Aufnahmen im TRC-Format

Tracking-Aufnahmen werden in der Implementierung mit mehreren Klassen und Strukturen repräsentiert. Dabei ist der Aufbau analog zur TRC-Datei. Eine übergeordnete Klasse *tracking_data* enthält eine Instanz einer Metadaten-Struktur und eine Liste der vorhanden Frames. Einzelne Frames enthalten den Index, den Aufnahmezeitpunkt und die sichtbaren Marker in diesem Frame. Jeder Marker besitzt einen Index sowie die Koordinaten. In Abbildung 3.1 ist ein kurzer Überblick über den Zusammenhang zwischen diesen Klassen und Strukturen zu finden.

Beim Ladevorgang wird die TRC-Datei zeilenweise verarbeitet. Daraus wird eine Instanz der zuvor beschriebenen Klasse *tracking_data* aufgebaut. Der Marker-Index stellt den Spalten-Index des jeweiligen Markers dar. Die Indizes sind aufsteigend, können jedoch lückenhaft sein, wenn nicht alle Marker in dem aktuellen Frame sichtbar sind.

Aus OptiTrack Motive exportierte Aufnahmedaten sowie die bereitgestellten Trainingsdaten weisen doppelte Frame-Indizes auf. Zudem stimmt die eigentliche Anzahl an Frames teilweise nicht mit der angegebenen Anzahl in den Metadaten überein (siehe Abschnitt 3.11.2 auf Seite 55). In diesen Fällen werden beim Import Warnungen ausgegeben. Die Frame-Indizes entsprechen also nicht in jedem Fall der Position in der Frame-Liste in *tracking_data*. Das führt dazu, dass der Zugriff auf einen Frame über dessen Index problematisch ist, da dieser nicht direkt zum Indizieren der Liste von Frames verwendet werden kann. Dies wird durch eine binäre Suche über die Liste gelöst.

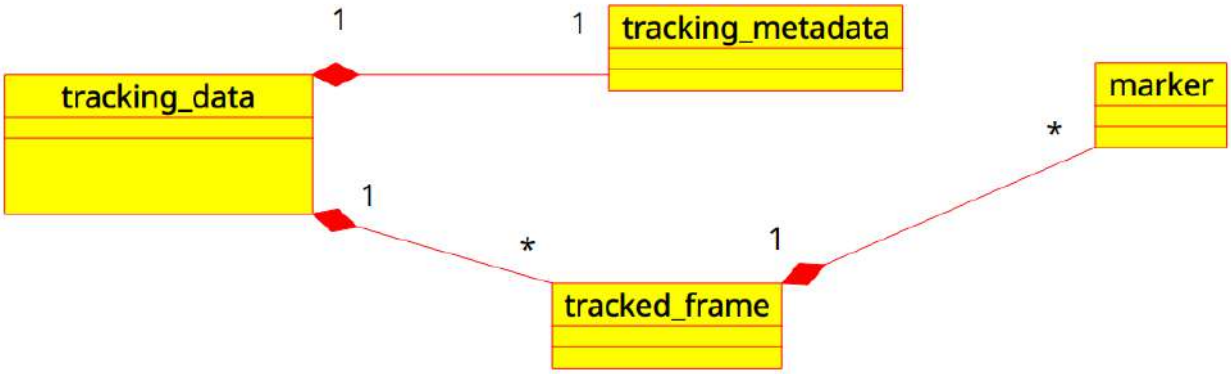

Abbildung 3.1.: Ein grober Überblick über die Repräsentierung von Tracking-Aufnahmen in der Implementierung.

Die Achsenkonfiguration beim Ladevorgang ist konfigurierbar, falls Daten aus unterschiedlichen Quellen zum Beispiel die Z-Achse für die Tiefe anstatt Höhe verwenden. Auch die Richtung der Achsen lässt sich anpassen, um zwischen rechts- und linkshändigen Koordinatensystemen zu wechseln.

Die hier beschriebene Implementierung ist in den Teilprojekten *tracking_data* und *io* zu finden. Eine Variante ist auch in Python vorhanden und wird für verschiedene Tools, wie die Vorschau von TRC-Dateien, verwendet.

3.3. Export von (Live-)Aufnahmen

Tracking-Daten, die über NatNet oder VRPN empfangen werden, können zur Laufzeit exportiert werden. Als Dateiformat wurde hierbei JSON ausgewählt, da es sich dabei um ein textuelles Format handelt und es im Falle von zum Beispiel Programmabstürzen problemlos manuell repariert werden kann. Des Weiteren gibt es dafür eine Schnittstelle in der Python-Standardbibliothek (Python Software Foundation, 2019a) und die Daten können einfach verarbeitet werden.

Der Export-Mechanismus ist zweiteilig. Der erste Teil ist die Klasse *json_writer*, die JSON-Dateien schreiben kann und auch an anderen Stellen in der Implementierung verwendet wird. Der *json_writer* ist hierbei einfach gehalten und unterstützt als Wurzelelement nur ein Objekt. In dieses Objekt können aber Instanzen der in JSON vorhandenen Datentypen geschrieben werden. Der zweite Teil ist die Klasse *tracking_data_writer*, die Tracking-Daten über den *json_writer* exportiert. Analog zum TRC-Format enthält die resultierende Datei Metadaten und die Frames mit sichtbaren Markern.

Die Implementierung hierzu ist in dem Teilprojekt *io* zu finden. Ein dazugehöriges Python-Skript *convert.py* im Teilprojekt *convert_tracking_output* erlaubt es, die Tracking-Daten im JSON-Format in das TRC-Format zu konvertieren.

Markerpositionen in verschiedenen Teilschritten der Pipeline sowie Projektionen können ebenfalls exportiert werden. Erstere werden dabei im PLY-Format gespeichert, welches für 3D-Objekte und -Szenen verwendet wird und in zum Beispiel Blender[1] importiert werden kann. Projektionen werden in das textuelle Graustufen-Bildformat PGM exportiert und können durch verschiedene Bildprogramme geöffnet werden.

[1] `https://www.blender.org/`

3.4. Bestimmen der optischen Achse der orthographischen Kamera

3.4.1. Zufallsgenerierung

Eine optische Achse wird zufällig erzeugt, indem drei Werte gleichverteilt im Intervall $[-1, 1]$ generiert werden. Der daraus entstehende Vektor wird dann normalisiert, damit er eine gültige Achse darstellt. Zum Generieren einer zufälligen Rotation wurde das Intervall so angepasst, dass alle Rotationen zwischen $-360°$ und $360°$ um jede der Achsen des Koordinatensystems möglich sind, und es wird auf eine Normalisierung verzichtet.

3.4.2. Auswahl einer optischen Achse

Wie in Abschnitt 2.14 auf Seite 21 bereits beschrieben, werden außerhalb des Netzwerktrainings zehn verschiedene optische Achsen zufällig generiert. Mit diesen Achsen werden dann jeweils Tiefenbilder, wie in Abschnitt 3.5 auf Seite 34 erläutert, projiziert.

Für jedes dieser Tiefenbilder wird ein sogenannter *spatial spread* berechnet. Dieses Maß ist nach (Han u. a., 2018a) durch das Maximum der Eigenwerte einer Kovarianzmatrix über die Bildkoordinaten der Marker gegeben. Da in (Han u. a., 2018a) explizit zweidimensionale Bildkoordinaten angesprochen wurden, wird dieses Maß nachfolgend auch als Flächen-Verteilung bezeichnet. Die Bildkoordinaten werden in Abschnitt 3.5 auf Seite 34 beschrieben. Die betrachteten Stichproben sind also die sichtbaren Marker und der zugehörige Zufallsvektor $\mathbf{x}$ setzt sich aus dessen zweidimensionalen Bildkoordinaten zusammen (Han u. a., 2018a, S.2). Die Kovarianzmatrix wird hier also folgendermaßen berechnet:

$$Cov(\mathbf{x}) = \begin{bmatrix} Var(\mathbf{x}_x) & Cov(\mathbf{x}_x, \mathbf{x}_y) \\ Cov(\mathbf{x}_y, \mathbf{x}_x) & Var(\mathbf{x}_y) \end{bmatrix} \tag{3.1}$$

Dabei werden nur die Diagonale und die Elemente oberhalb davon berechnet, um die Rechenoperationen zu sparen. Dies wird durch die Symmetrie von Kovarianzmatrizen ermöglicht (siehe Abschnitt 2.4.1 auf Seite 11).

Bei der Berechnung der Kovarianzmatrix wird nach (Spruyt, 2014) als Erwartungswert der Stichprobenmittelwert (*sample mean*) eingesetzt. Als Schätzung für die Varianz wird die Stichprobenvarianz ohne Bias (*unbiased sample variance*) verwendet, da solche Schätzungen bevorzugt werden (Spruyt, 2014; Goodfellow u. a., 2016, S.123ff.). Die Definitionen für diese Schätzverfahren sind in Abschnitt 2.4.2 auf Seite 12 zu finden. Für die Kovarianz wird die Stichprobenkovarianz verwendet, die nach (Spruyt, 2014; Hoffbeck u. Landgrebe, 1996) ein übliches Annäherungsverfahren dafür ist. Die Definition dazu ist in Abschnitt 2.4.3 auf Seite 12 zu finden.

Zunächst wird der Stichprobenmittelwert mithilfe einer Iteration über alle Stichproben, das heißt Bildpositionen, bestimmt. In einem zweiten Schritt wird erneut über die Stichproben iteriert und die Summen für die Varianzen und die Kovarianz anhand des Stichprobenmittelwertes berechnet. Jede dieser Summen wird dann entsprechend der Stichprobenvarianz und -kovarianz durch den Faktor $m - 1$ dividiert. Aus den Ergebnissen dieser Berechnungen kann dann anhand der Formel 3.1 die Kovarianzmatrix gebildet werden. Zum Berechnen der Eigenwertzerlegung wird dann die Eigen-Bibliothek (Jacob u. Guennebaud, 2018a) verwendet.

Geometrisch gesehen ist der Stichprobenmittelwert hier das Zentrum der Punktwolke und des Tiefenbildes. Die Kovarianzmatrix gibt also an, wie sich die beiden Dimensionen der Koordinaten zueinander verhalten. Die Eigenvektoren und -werte dieser Matrix geben dann an, in welche Richtung die Abweichung höher ist (siehe Abschnitt 2.4.1 auf Seite 11 und (Spruyt, 2014)).

Da die zwei resultierenden Eigenvektoren orthogonal und deren Eigenwerte reell sind sowie die zugehörende Kovarianzmatrix eine Skalierung in deren Richtungen darstellt (siehe Abschnitt 2.4.1 auf Seite 11), werden die Eigenwerte als Kantenlänge eines durch die Eigenvektoren gebildeten

Rechtecks betrachtet. Die Fläche dieses Rechtecks, also die Multiplikation beider Eigenwerte, wird dabei als Schätzung für das Maß der Flächen-Verteilung verwendet.

Dadurch werden Tiefenbilder bevorzugt, die eine Flächenverteilung in beide Dimensionen besitzen. Würden Tiefenbilder nur nach dem Maximum einer Dimension ausgewählt werden, könnten Bilder (siehe Abbildung 3.2) mit einer gestreckten Verteilung in eine Richtung ausgewählt werden. Dadurch würden sich viele Überdeckungen entwickeln, was die Verarbeitung mit dem Netzwerk erschwert.

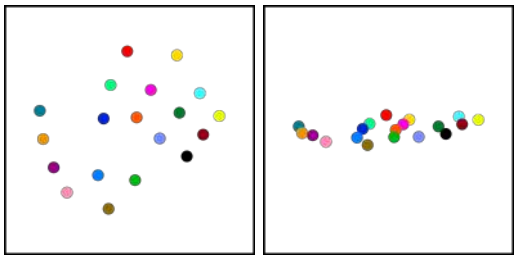

Abbildung 3.2.: Eine Punktwolke projiziert mit unterschiedlichen optischen Achsen (siehe Abschnitt 3.5 auf der nächsten Seite). Links ist ein gute und rechts eine schlechte Wahl der Achse zu sehen. Wenn das Maximum aller Eigenwerte betrachtet wird, würden beide Bilder das gleiche Maß für die Flächenverteilung besitzen. Der Übersichtlichkeit halber wurde anstatt von Tiefenwerten eine zufällige Farbe verwendet, um die Überlappung zu verdeutlichen.

Die optische Achse, die nach diesem Maß das beste Tiefenbild lieferte, wird ausgewählt und im nächsten Schritt als Eingabe für das Netzwerk verwendet.

3.4.3. Anpassungen am Auswahlverfahren

Aufgrund von schlechten Ergebnissen wird eine Auswahl der Tiefenbilder mit der kleinsten Flächen-Verteilung, also das Minimum des Eigenwert-Produkts, betrachtet. Dies lieferte widersprüchlich ein deutlich besseres Ergebnis und wird bei der Auswertung in Abschnitt 4.2.1 auf Seite 66 verwendet. Des Weiteren wurde das Maximum der Summe der beiden Eigenwerte betrachtet.

Das Problem an dem bestehenden Verfahren aus (Han u. a., 2018a) ist, dass nur die Abweichung zum Zentrum des Bildes betrachtet wird und für zum Beispiel das Tiefenbild in Abbildung 3.3 eine gute Bewertung liefert, welches aber viele Überdeckungen aufweist.

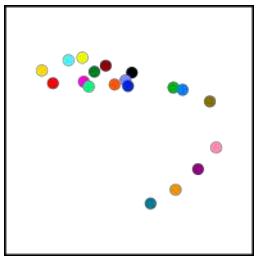

Abbildung 3.3.: Eine Punktwolke mit abgespreiztem Daumen projiziert mit einer optischen Achse, die eine gute Flächenverteilung besitzt, aber trotzdem viele Überlappungen aufweist.

Eine alternative Herangehensweise ist die Verwendung der Summe der Distanzen $d_{i,j}$ mit $i, j \in \{1, 2, \ldots, m\}$ zwischen den Bildkoordinaten b_i aller Marker, also die Auswahl eines Tiefenbildes, dessen Marker möglichst weit voneinander entfernt sind. Das Maß setzt sich also folgendermaßen

zusammen:

$$\sum_{i=1}^{m} \sum_{j=1;i\neq j}^{m} ||b_i - b_j||_2 \tag{3.2}$$

Analog dazu wird auch ein „Überdeckungsmaß“ betrachtet, wobei ebenfalls die Distanzen zwischen allen Bildkoordinaten berechnet wurden. Dabei werden aber nur Distanzen, die unter einem bestimmten Schwellenwert ε liegen, also eine Überdeckung zwischen Markern mit einem Radius von ε darstellen, betrachtet. Teil des Maßes ist dann die Stärke solcher Überdeckungen. Es wird das Quadrat der Überdeckungsdistanz verwendet, damit stärkere Überdeckungen schneller ein schlechteres Maß liefern. Das Maß setzt sich also folgendermaßen zusammen:

$$\sum_{i=1}^{m} \sum_{j=1;i\neq j}^{m} f_{Ueberdeckung}(||b_i - b_j||_2) \tag{3.3}$$

Dabei ist $f_{Ueberdeckung}$ wie folgt definiert:

$$f_{Ueberdeckung}(dist) = \begin{cases} (\epsilon - dist)^2 & \text{wenn } dist < \epsilon \\ 0 & \text{sonst} \end{cases} \tag{3.4}$$

Der Schwellenwert wird analog zum Splatting-Radius mit $\varepsilon = \sqrt{3.5}$ festgelegt (siehe Abschnitt 3.5.1 auf der nächsten Seite). Es wird untersucht, ob ein „Sicherheitsabstand“, also $\varepsilon = \sqrt{3.5} + 1$, die Ergebnisse verbessert.

3.5. Projektion von Tiefenbildern

Vor der Projektion wird die Marker-Punktwolke um den Ursprung zentriert, indem das Zentrum dieser Punktwolke berechnet und dann von jeder Marker-Position subtrahiert wird.

Basierend auf der Blickrichtung und der daraus resultierenden optischen Achse der orthographischen Kamera wird zunächst die Transformation, wie in dem Abschnitt 2.2 auf Seite 10 und (Scratchapixel, 2016) beschrieben, bestimmt.

Durch das Zentrieren der Punktwolke kann, wie schon in dem Abschnitt 2.2 erwähnt, dabei die Translation vernachlässigt werden. Die optische Achse ist in diesem Fall der *look*- beziehungsweise w-Vektor. Die Richtung des *look*-Vektors wird umgekehrt, um die Projektion an ein linkshändiges Koordinatensystem anzupassen. Falls die optische Achse entlang oder entgegengesetzt der Y-Achse verläuft, werden die Rotationsmatrizen manuell berechnet. Des Weiteren erlaubt es die Verwendung anderer Rotationen ohne größere Anpassungen des Quellcodes (siehe Abschnitt 3.4 auf Seite 32).

Da die Markerdaten in einem Z-Up-Koordinatensystem vorliegen, N' jedoch von einem Y-Up-Koordinatensystem ausgeht, werden zunächst die beiden Achsen vertauscht. Nach der Transformation der Punktwolke mit N' entsprechen die X- und Y-Achse den jeweiligen Achsen des resultierenden Tiefenbildes und die Z-Achse der Tiefe, die später zur Berechnung der Pixelwerte verwendet wird. Die transformierte Punktwolke wird nachfolgend *pc* genannt.

Die Maße der Punktwolke auf den Bildachsen werden bestimmt und nachfolgend als *pc_min* für die minimalen und *pc_max* für die maximalen Werte bezeichnet. Dies wird analog für die Tiefe (Z-Achse) durchgeführt. Die Tiefenmaße werden nachfolgend *depth_min* beziehungsweise *depth_max* genannt. Die Höhe (*width*) und Breite (*height*) werden folgendermaßen bestimmt:

$$\text{width} = |\text{pc\_max}_x - \text{pc\_min}_x| \tag{3.5}$$

$$\text{height} = |\text{pc\_max}_y - \text{pc\_min}_y| \tag{3.6}$$

Wie bereits in Abschnitt 2.14 auf Seite 21 beschrieben, muss auf dem Tiefenbild um die projizierte Punktwolke von jeder Seite ein Rand in Höhe von zehn Prozent ($m = 10$) der Bildbreite beziehungsweise -höhe frei von projizierten Markern gehalten werden. Dabei wird von einem quadratischen Bild mit der Kantenlänge $a = 52$ ausgegangen. Die Breite des Randes (*margin_offset*) und die Kantenlänge ohne diesen Rand (*target_size*) werden folgendermaßen berechnet:

$$\text{margin\_offset} = \frac{a}{100} * m \tag{3.7}$$

$$\text{target\_size} = a - 2 * \text{margin\_offset} \tag{3.8}$$

Ob die Breite oder Höhe der Punktwolke größer ist, bestimmt die Berechnung des Skalierungsfaktors *factor*. Die Skalierung der Punktwolke wird mit diesem Faktor gleichmäßig auf beiden Achsen durchgeführt, um die in Abschnitt 2.14 beschriebene uniforme Verkleinerung beziehungsweise Vergrößerung zu realisieren. Im Folgenden ist die Berechnung des Faktors zu sehen:

$$\text{factor} = \begin{cases} \frac{\text{target\_size}}{\text{width}} & \text{falls width} > \text{height} \\ \frac{\text{target\_size}}{\text{height}} & \text{sonst} \end{cases} \tag{3.9}$$

Der Abstand zum freien Rand *offset* wird so bestimmt, dass die projizierte Punktwolke nach der Translation mit *offset* vertikal oder horizontal auf dem Tiefenbild zentriert wird:

$$\text{offset} = \begin{cases} \begin{bmatrix} 0 \\ \frac{\text{target\_size}}{2} - \text{height} * \frac{\text{factor}}{2} \end{bmatrix} & \text{falls width} > \text{height} \\ \begin{bmatrix} \frac{\text{target\_size}}{2} - \text{width} * \frac{\text{factor}}{2} \\ 0 \end{bmatrix} & \text{sonst} \end{cases} \tag{3.10}$$

Um die Marker nun korrekt im Koordinatensystem des Tiefenbildes zu positionieren, wird die Punktwolke zunächst so verschoben, dass alle Punkte im ersten Quadranten des Koordinatensystems liegen. Daraufhin wird die Punktwolke skaliert, sodass diese auf das Tiefenbild passt. Mit einer Translation um *offset* und *margin_offset* wird die Punktwolke dann auf dem Tiefenbild zentriert. Es wird also die folgende Transformation T auf den Bildkoordinaten durchgeführt:

$$T = \begin{bmatrix} 1 & 0 & \text{offset}_x + \text{margin\_offset} \\ 0 & 1 & \text{offset}_y + \text{margin\_offset} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \text{factor} & 0 & 0 \\ 0 & \text{factor} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -\text{pc\_min}_x \\ 0 & 1 & -\text{pc\_min}_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.11}$$

$$= \begin{bmatrix} \text{factor} & 0 & \text{offset}_x + \text{margin\_offset} - \text{factor} * \text{pc\_min}_x \\ 0 & \text{factor} & \text{offset}_y + \text{margin\_offset} - \text{factor} * \text{pc\_min}_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.12}$$

Die Implementierung zu diesem und zum nächsten Abschnitt ist in dem Teilprojekt *deep_labeling* in den Dateien *projection.h* und *projection.cpp* zu finden.

3.5.1. Splatting

Da die Tiefenbild-Beispiele in (Han u. a., 2018a) für den selben Marker die gleiche Intensität besitzen, wird beim Splatting die folgende Footprint-Funktion $f(x)$ mit einem festen Schwellenwert ε verwendet:

$$f(x) = \begin{cases} 1 & \text{falls } ||x|| \leq \varepsilon \\ 0 & \text{sonst} \end{cases} \tag{3.13}$$

Der Schwellenwert wurde nach einer Kommunikation mit den Autoren von (Han u. a., 2018a)[2] folgendermaßen festgesetzt:

$$\varepsilon = \sqrt{3.5} \tag{3.14}$$

Für jeden mit T transformierten Marker werden zunächst die Bildkoordinaten b_i durch das Runden von dessen Position bestimmt. Diese Rundungsoperation wurde auch nach der oben genannten Kommunikation mit den Autoren von (Han u. a., 2018a) eingeführt. Aufgrund dessen wurde auch die Verwendung von Pixelzentren bei der nachfolgenden Fußabdruckberechnung eingestellt.

Die Verwendung von Pixelzentren basierte auf dem Punkt-Rendering in Abschnitt 2.3.1 auf Seite 11 und war der Versuch, den Fußabdruck auf dem Pixelgitter um die Bildkoordinate des jeweiligen Markers zu zentrieren (siehe Abbildung 3.4). Beim Testen wurden keine nennenswerten Veränderungen festgestellt.

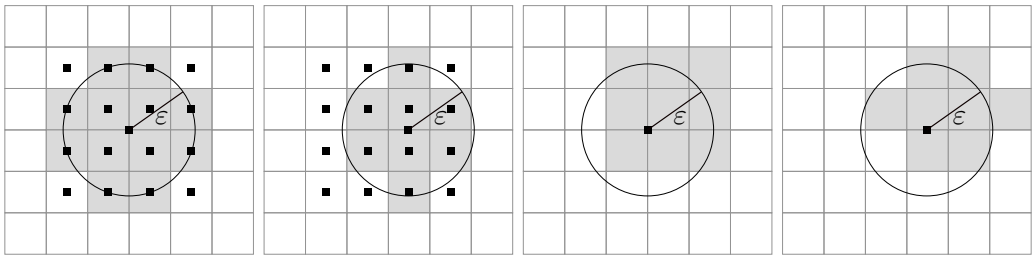

Abbildung 3.4.: Der Vergleich zwischen dem Splatting mit und ohne Verwendung von Pixelzentren. In *a)* und *b)* ist zu sehen, dass die gesetzten Pixel um die Bildkoordinate zentriert sind. *c)* und *d)* stellen das Splatting mit den selben Bildkoordinaten wie in *a)* beziehungsweise *b)* dar.

Beim Splatting wird um die Bildkoordinaten in einem Suchradius r jeder Pixel mit f darauf geprüft, ob dieser Pixel noch ein Teil des Fußabdrucks ist. Dabei ist der Suchradius auf Grundlage von ε so gewählt, dass alle Pixel innerhalb des Schwellenwertes beachtet werden, und setzt sich folgendermaßen zusammen:

$$r = \lceil \varepsilon \rceil + 1 \tag{3.15}$$

Der Pixelwert v für alle Pixel mit den Koordinaten $p_i \in \{ p \in \mathbb{Z} \mid \ ||p - b_i|| \leq r \}$ im Suchradius wird basierend auf dem Tiefenwert des jeweiligen Markers sowie *depth_min* und *depth_max* bestimmt. Dabei wird der Tiefenwert d zuvor in das gewünschte Intervall $[0.1, 1]$ skaliert. Darüber hinaus wird die Footprint-Funktion verwendet, um das Splatting umzusetzen, wie es im Folgenden zu sehen ist:

$$v_s = (\frac{d - \text{depth\_min}}{|\text{depth\_max} - \text{depth\_min}|}) * 0.9 + 0.1 \tag{3.16}$$

$$v = f(p_i - b_i) * v_s \tag{3.17}$$

Wenn der Tiefenwert umgekehrt werden soll, wird stattdessen folgende Berechnung durchgeführt (siehe Abschnitt 3.9 auf Seite 46):

$$v' = f(p_i - b_i) * ((1 - \frac{d - \text{depth\_min}}{|\text{depth\_max} - \text{depth\_min}|}) * 0.9 + 0.1) \tag{3.18}$$

[2]Die Autoren antworteten auf eine E-Mail am 23. Juni 2019.

Beim Setzen der jeweiligen Pixelwerte wird darauf geachtet, bisherige Splats nur zu überschreiben, wenn der neue Pixelwert kleiner als der bestehende ist, also der Marker näher an der Kamera ist. Des Weiteren wird der bestehende Pixelwert nicht überschrieben, wenn der Pixelwert Null ist, da in diesem Fall der aktuelle Pixel außerhalb des Splats aber noch innerhalb des Suchradius ist und somit bestehende Splats fälschlicherweise überschreiben würde.

Darüber hinaus werden die normalisierten Marker-Koordinaten n_i basierend auf der Projektion bestimmt. Dies wurde ebenfalls nach der oben genannten Kommunikation mit den Autoren umgesetzt und ersetzte eine Anpassung der Netzwerkausgabe, wie es in Abschnitt 3.7 auf Seite 42 kurz erläutert wird. Dazu werden die Bildkoordinaten durch die Kantenlänge $a = 52$ des Bildes skaliert und als Z-Koordinate wird der normalisierte Pixelwert ohne Footprint-Funktion verwendet, wie es im Folgenden zu sehen ist:

$$n_i = \begin{bmatrix} \frac{p_{i_x}}{a} \\ \frac{p_{i_y}}{a} \\ v_s \end{bmatrix} \tag{3.19}$$

3.6. Verarbeiten von Tiefenbildern mit dem neuronalen Netzwerk

Zunächst wird das in (Han u. a., 2018a,b) beigelegte neuronale Netzwerk verwendet. Dieses liegt im LuaTorch-Format vor (Collobert u. a., 2019a; Han u. a., 2018b, Issue #1). Diese Programmierbibliothek wird nicht mehr weiterentwickelt (Collobert u. a., 2019b, README) und stellt keine Installationsanweisungen für Windows bereit (Collobert u. a., 2019a, Getting started with Torch). Obwohl im Sourcecode Skripte für Windows vorlagen, schlug die Installation an mehreren Stellen fehl. Des Weiteren ist es nicht möglich, das Netzwerk ohne CUDA-Hardware zu nutzen[3], wodurch Prototyping auf zum Beispiel Notebooks ohne dedizierte Grafikkarten verhindert wird. Aus diesen Grund fiel die Entscheidung darauf, eine andere Programmierbibliothek für das Deep Learning in dieser Masterarbeit zu nutzen.

Hierbei fiel TensorFlow positiv auf, da es sich einfach installieren lässt, von Unternehmen wie Google und Intel verwendet wird, frei erhältlich und auch auf der CPU ohne Anpassungen lauffähig ist (Abadi u. a., 2015).

In den beiden nächsten Abschnitten wird zunächst auf die Installation von LuaTorch unter einer Linux-Distribution und auf die Verwendung von LuaTorch vom C++-Code eingegangen. Die darauf folgenden Abschnitte beschreiben die Migration des Netzwerks ins TensorFlow-Format und die Einbindung der TensorFlow Python-Schnittstelle im C++-Code.

3.6.1. LuaTorch unter ArchLinux

Unter Linux werden vorkonfigurierte Paketskripte Dritter aus der Arch User Repository verwendet. Da diese nicht offiziell von den LuaTorch-Autoren bereitgestellt werden, ist die Verwendung dieser außerhalb der Migration problematisch. Eine Alternative wäre hier die Verwendung der offiziellen Installationsskripte, auf dessen Anwendung aufgrund der Erfahrungen unter Windows verzichtet wurde.

Es wurden neben der LuaTorch-Abhängigkeit LuaJIT (Pall, 2018), das eine Laufzeitumgebung für Lua ist, die folgenden Abhängigkeiten installiert:

Die verwendeten Paketskripte sind im Anhang in der Tabelle B.2 im Abschnitt B.2 auf Seite 90 des Anhangs zu finden.

[3] Beim Laden des beigelegten Netzwerks in LuaTorch wird ersichtlich, dass Netzwerk-Ebenen, die CUDA-spezifisch sind, verwendet wurden.

[4] Abgerufen am 28. Dezember 2019.

Abhängigkeit	Webadresse[4]
torch7	`https://github.com/torch/torch7/`
torch7-cudnn-r7	`https://github.com/soumith/cudnn.torch`, Git-Branch „R7“
torch7-cutorch	`https://github.com/torch/cutorch`
torch7-cwrap	`https://github.com/torch/cwrap`
torch7-nn	`https://github.com/torch/nn`
torch7-paths	`https://github.com/torch/paths`

Tabelle 3.1.: Eine Liste von Abhängigkeiten, die für das in (Han u. a., 2018a) mitgelieferte Netz benötigt werden.

3.6.2. Kommunikation zwischen C++ und Lua / LuaJIT

Sowohl LuaJIT als auch LuaTorch stellen Header-Dateien zur Einbindung in C oder C++ bereit (Ierusalimschy u. a., 2012; Collobert u. a., 2019a). Zunächst wird in C++ die Lua-Laufzeitumgebung mit `luaL_newstate` initialisiert und die Standardbibliothek mit `luaL_openlibs` importiert. Ein für die Kommunikation entwickeltes Lua-Skript *interface.lua* aus dem Teilprojekt *torch_interface* wird dann über `luaL_dofile` in der Laufzeitumgebung ausgeführt.

Dieses Skript lädt das Netzwerk mithilfe von LuaTorch und definiert eine globale Funktion `cnn_prediction`. Diese erwartet eine Tabelle von Pixeldaten von 32 Tiefenbildern. Die Pixeldaten müssen dafür eindimensional im row-major Format vorliegen. Dies erleichtert die Konvertierung der Tiefenbilder auf der C++-Seite. LuaTorch stellt eine Tensor-Implementierung bereit, mit der die Dimensionen einfach angepasst werden können. Das Netzwerk aus (Han u. a., 2018a) erwartet als Eingabe einen vierdimensionalen Tensor im Format $32 \times 1 \times 52 \times 52$, also *channel-first* mit einer Minibatch-Größe von 32 (siehe Abschnitt 2.8.1 auf Seite 15). Dies steht im Kontrast zu der im Paper beschrieben Minibatch-Größe von 256 zum Lernen von Netzwerken (siehe Han u. a., 2018a, S.6).

Dann wird die `cuda`-Methode des aus Tiefenbildern bestehenden Tensors aufgerufen, um diesen auf die GPU zu übertragen. Für die Inferenz wird die `forward`-Methode des Netzwerks verwendet, die einen CUDA-Tensor im Format $32 \times 19 \times 3$. Das Ergebnis wird dann mit der `float`-Methode in einen CPU-Tensor konvertiert und somit wieder in den RAM übertragen. Dieser Tensor wird dann von der Funktion zurückgeliefert.

Auf der C++-Seite wird diese Funktion aufgerufen, wenn ein Tiefenbild verarbeitet werden soll. Beim Übertragen wird die benötigte Lua-Tabelle aus den Tiefenbildern, die als 52×52-Matrizen vorliegen, erzeugt. Dabei wird darauf geachtet, dass wenn weniger als 32 Tiefenbilder verarbeitet werden sollen, leere Tiefenbilder für die fehlenden übergeben werden. Nach dem Aufruf der Funktion lässt sich Tensor über die C/C++-Schnittstelle von LuaTorch auslesen und in eine Liste von Vektoren konvertieren (Ierusalimschy u. a., 2012; Collobert u. a., 2017, 2019b,a).

Diese Implementierung wird im System nur zur Verifikation der TensorFlow verwendet und kann optional hinzugeschaltet werden. Der C++-Teil ist in *torch_interface.h* und *torch_interface.cpp* selben Teilprojekt *torch_interface* zu finden.

3.6.3. Konvertierung des Netzwerks ins TensorFlow-Format

Die Migration kann unabhängig vom Hauptprojekt durchgeführt werden und wird von einem Python-Skript, welches TensorFlow und Lua über die *lupa*-Bibliothek anspricht. Diese Bibliothek erlaubt es Zeichenketten als Lua-Code auszuführen und übernimmt die Konvertierung zwischen Lua- und Python-Grunddatentypen (Behnel, 2019). Dadurch lässt sich LuaTorch einfacher als über C++ ansprechen. Bei der Installation der *lupa*-Abhängigkeit muss die folgende Kommandozeile verwendet werden, um LuaJIT als Laufzeitumgebung zu nutzen (siehe Behnel, 2019, Issue

#104):

```
LUA=luajit pip install lupa --no-binary lupa
```

Bei der Migration wird analog zum C++-Lua-Interface zunächst das Netzwerk mithilfe von LuaTorch geladen. Aus diesem Netzwerk werden dann die jeweiligen Ebenen extrahiert. Die Parameter, wie Gewichte und Bias-Terme, dieser Ebenen liegen als Tensoren vor. Da in *lupa* die automatische Konvertierung nur für Grunddatentypen unterstützt wird, werden zunächst die Dimensionen extrahiert und der Tensor dann als eine eindimensionale Lua-Tabelle dargestellt. Diese Lua-Tabelle wird dann automatisch in eine Map in Python konvertiert. Auf der Python-Seite werden die Werte dieser Map in ein NumPy-Array konvertiert. Da in dem LuaTorch-Netzwerk die Gewichte im *channel-first*-Format vorliegen, TensorFlow diese standardmäßig aber als *channel-last* erwartet, werden die Achsen des Parameter-Tensors bei Convolution-Ebenen nach (Stack Overflow, 2017a) dementsprechend vertauscht (The SciPy community, 2019; Collobert u. a., 2019b, Wiki: Torch-for-Numpy-users).

Zum Aufbau des Netzwerks in TensorFlow wird die darin mitgelieferte High-Level-Programmierschnittstelle Keras verwendet. Diese ist auch unabhängig von TensorFlow verfügbar und unterstützt andere Back-Ends (Chollet u. a., 2019). In der Tabelle 3.2 sind die jeweiligen Klassen aus Keras mit ihren Ausgabeformaten und der Anzahl ihrer Parameter vermerkt.

Die Parameter der jeweiligen Ebenen in Keras können über die Methode `set_weights` gesetzt werden. Dabei wird eine Liste von Tensoren erwartet. Die Reihenfolge der Elemente dieser Liste konnte bei Convolution-Ebenen und vollständig verbundenen Ebenen über die Dimensionen des über `get_weights` zurückgelieferten Tensors ermittelt werden, da die Gewichte und Bias-Terme unterschiedliche Dimensionen besitzen. Bei Batch-Normalization-Ebenen wurde die Reihenfolge aus dem Quellcode von (Abadi u. a., 2015) ermittelt.

Bei Convolution-Ebenen wird eine Filtergröße von 3×3 und die in Deep Labels beschrieben Anzahl von Filtern der jeweiligen Ebene verwendet (Han u. a., 2018a,b).

Bei Batch-Normalization-Ebenen wird die relevante Achse (Parameter *axis*) auf 3 gesetzt, da *channel-last* verwendet wird. Die Gewichte und der Bias dieser Ebenen werden in TensorFlow respektive als Gamma und Beta bezeichnet. Des Weiteren werden der laufende Durchschnitt (*running mean*) und die laufende Varianz (*running variance*) des bestehenden Netzwerks übernommen. Analog zu (Han u. a., 2018a) wird ein Moment von 0.01 und ein Epsilon von $1e-5$ verwendet (Han u. a., 2018b; Chollet u. a., 2019; Abadi u. a., 2015). Dabei wird auf die *fused*-Variante verzichtet, da diese stärkere Differenzen zwischen den Ausgaben des LuaTorch- und TensorFlow-Netzwerks aufwies, als Batch-Normalization-Ebenen mit explizit ausgeschaltetem *fused*-Schalter.

Die anderen Parameter werden dem bestehenden Netzwerk entsprechend gesetzt. Analog dazu werden die Aktivierungsebene *ReLU* und die Pooling-Ebene *MaxPool* verwendet. Letztere wird in der zweidimensionalen Variante als *MaxPool2D* bezeichnet (Han u. a., 2018a,b; Chollet u. a., 2019; Abadi u. a., 2015).

Da bei den bisherigen Ebenen das *channel-last*-Format verwendet wird, die vollständig verbundene Ebene (*FC*) jedoch ohne Veränderung übernommen wird, übernimmt ein Permute-Layer mit dem Parameter $(3, 1, 2)$ das Verändern der Zwischenergebnisse in das *channel-first*-Format.

Vor der ersten vollständig verbundenen Ebene wird analog zu (Han u. a., 2018a) eine Reshape-Ebene verwendet, die die Zwischenergebnisse im Format $128 \times 9 \times 9$ in einen eindimensionalen Tensor mit 10368 Elementen umformt. Nach der zweiten FC-Ebene wird mit einer weiteren Reshape-Ebene der eindimensionale Tensor mit 57 Elementen in einen mit den Dimensionen 19×3 umgeformt.

Die erfolgreiche Migration wird durch Null-Eingaben, zufällige Eingaben und echte Tiefenbilder verifiziert. Die Ergebnisse unterscheiden sich nur in nicht mehr signifikanten Nachkommastellen. Dieser Unterschied lässt sich auf die Verwendung von Gleitkommazahlen bei den Berechnungen zurückführen. Diese haben nach (Goldberg, 1991, S.1) wegen der festen Bitgröße immer einen Rundungsfehler.

Index	Klasse	Filtergröße	Ausgabeformat	Parameteranzahl
0	`Conv2D`	$64 \times 3 \times 3$	$50 \times 50 \times 64$	640
1	`BatchNormalization`	-	$50 \times 50 \times 64$	256
2	`ReLU`	-	$50 \times 50 \times 64$	0
3	`Conv2D`	$64 \times 3 \times 3$	$48 \times 48 \times 64$	36928
4	`BatchNormalization`	-	$48 \times 48 \times 64$	256
5	`ReLU`	-	$48 \times 48 \times 64$	0
6	`MaxPool2D`	2×2	$24 \times 24 \times 64$	0
7	`Conv2D`	$128 \times 3 \times 3$	$22 \times 22 \times 128$	73856
8	`BatchNormalization`	-	$22 \times 22 \times 128$	512
9	`ReLU`	-	$22 \times 22 \times 128$	0
10	`Conv2D`	$128 \times 3 \times 3$	$20 \times 20 \times 128$	147584
11	`BatchNormalization`	-	$20 \times 20 \times 128$	512
12	`ReLU`	-	$20 \times 20 \times 128$	0
13	`Conv2D`	$128 \times 3 \times 3$	$18 \times 18 \times 128$	147584
14	`BatchNormalization`	-	$18 \times 18 \times 128$	512
15	`ReLU`	-	$18 \times 18 \times 128$	0
16	`MaxPool2D`	2×2	$9 \times 9 \times 128$	0
17	`Permute`	-	$128 \times 9 \times 9$	0
18	`Reshape`	-	10368	0
19	`Dense`	2048×10368	2048	21235712
20	`ReLU`	-	2048	0
21	`Dense`	2048×57	57	116793
22	`Reshape`	-	19×3	0

Tabelle 3.2.: Die verwendeten Ebenen mit den dazugehörigen Keras-Klassennamen im Python-Modul `tensorflow.keras.layers`, Ausgabeformat ohne Batchgröße sowie die Anzahl der Parameter. Die Gesamtzahl an Parametern beträgt 21761145 (Han u. a., 2018a; Chollet u. a., 2019).

TensorFlow beziehungsweise Keras erlaubt es, Netzwerke inklusive der Gewichte und sogar dem aktuellen Lernstatus zu exportieren. Das dadurch exportierte Netzwerk kann damit wie im nächsten Abschnitt beschrieben verwendet werden (Abadi u. a., 2015; Chollet u. a., 2019).

Die Implementierungen zu diesem Abschnitt sind in den Python-Skripten *convert_model.py* und *torch_model.py* im Teilprojekt *cnn-conversion* zu finden. Zur Einbindung von LuaJIT in die CMake-Build-Beschreibung wurde (Stack Overflow, 2019) hinzugezogen.

3.6.4. Verwendung des konvertierten Netzwerks

Die Verwendung der C++-Schnittstelle von TensorFlow setzt voraus, die Bibliothek selbst zu kompilieren und den eigenen Programmcode in der TensorFlow-Verzeichnisstruktur zu platzieren[5]. Die Verwendung von TensorFlow über die Python-Programmierschnittstelle ist bereits vom Migrationsprozess bekannt. Bei der Verwendung des Netzwerks müssen nur die Tiefenbilder und Vorhersagen übertragen werden. Andere Schnittstellen von TensorFlow werden nicht benötigt. Unter diesem Hintergrund ist es einfacher Python einzubinden und das bereits kompilierte TensorFlow darüber zu verwenden (Abadi u. a., 2015). Dies wird im Folgenden beschrieben.

Das Aufrufen der TensorFlow-Programmierschnittstelle zum Verwenden des konvertierten Netz-

[5]Das Tutorial dazu ist unter `https://github.com/tensorflow/docs/blob/master/site/en/r1/guide/extend/cc.md` zu finden.

werks wird analog zu der im Abschnitt 3.6.2 auf Seite 38 beschriebenen Kommunikation zwischen Lua und C++ durchgeführt. Es wird ebenfalls ein Teil in C++ und ein Teil in Python implementiert.

Die Python-Laufzeitumgebung erlaubt es, die Suchpfade für Module zur Laufzeit anzupassen (Python Software Foundation, 2019b, Py_SetPath). Bei der Initialisierung dieser Umgebung wird das *components*-Verzeichnis diesen Pfaden hinzugefügt, damit entsprechende Teilprojekte direkt als Modul in Python importiert werden können. Des Weiteren wird die C/C++-Schnittstelle von NumPy initialisiert. Bei dieser Schnittstelle muss darauf geachtet werden, dass standardmäßig die Definitionen spezifisch für die jeweilige Übersetzungseinheit sind. In einer Einheit wird die Definition `PY_ARRAY_UNIQUE_SYMBOL` auf den programmspezifischen Wert `hand_tracking_ARRAY_API` gesetzt. Dadurch wird die Programmierschnittstelle global unter diesem Wert sichtbar. Wird die Schnittstelle nun von anderen Einheiten genutzt, wird zusätzlich zu dieser Definition noch `NO_IMPORT_ARRAY` definiert. Dies verhindert, dass mehrere globale Variablen unter dem gleichen Namen von NumPy erzeugt werden (The SciPy community, 2019). Die zur Initialisierung gehörende Implementierung ist in dem Teilprojekt *python_interface* zu finden. Wenn eine Python-Umgebung der Version 3.7 oder höher verwendet wird, gibt es ohne die Definition `PY_SSIZE_T_CLEAN` Warnungen (siehe (Python Software Foundation, 2019b, Embedding Python in Another Application)).

Der in Python implementierte Teil ist im Teilprojekt *cnn* zu finden und definiert eine Klasse `CNN`, die das geladene Netzwerk darstellt, und eine Funktion `load`, die das Netzwerk aus einer Datei importieren kann. Die Klasse stellt eine Methode `predict` zum Verarbeiten der Tiefenbilder bereit. Diese werden als eine Liste von NumPy-Arrays mit den Dimensionen 52×52 im *row-major*-Format erwartet. Nach einer Prüfung der Datentypen und Dimensionen werden alle Tiefenbilder in einen Tensor der Form $N \times 52 \times 52 \times 1$ konvertiert. Dabei ist N die Anzahl der Tiefenbilder, die gleichzeitig verarbeitet werden sollen. Nach dem Aufruf der `predict`-Methode des TensorFlow-Netzwerks wird der zurückgelieferte Tensor, der als NumPy-Array vorliegt, zurückgegeben.

Falls eine NVIDIA-Grafikkarte der RTX-Serie verwendet wird, treten Probleme mit der Initialisierung der Convolution-Ebenen des Netzwerks auf. Zur Fehlerbehebung wird der Quellcode aus Abbildung 3.5 vor dem Laden des Netzwerks ausgeführt. Dadurch wird die Verwendung des GPU-Speichers so angepasst, dass dieser nicht sofort vollständig, sondern nur wenn benötigt, von TensorFlow alloziert wird (Abadi u. a., 2015, GitHub-Repository: Issue #24496 und *tensorflow/core/protobuf/config.proto*).

```
1 config = tf.ConfigProto()
2 config.gpu_options.allow_growth = True
3 tf.keras.backend.set_session(tf.Session(config=config))
```

Abbildung 3.5.: Anpassung des Verwendungsverhaltens vom GPU-Speicher durch TensorFlow zur Fehlerbehebung bei RTX-Grafikkarten (Abadi u. a., 2015, GitHub-Repository: Issue #24496).

In der C++-Implementierung übernimmt die Klasse `tensorflow\_interface` die Verarbeitung der Tiefenbilder. Beim Erzeugen einer Instanz dieser Klasse wird das oben genannte Teilprojekt in die Python-Laufzeitumgebung importiert und die `load`-Funktion genutzt um eine Instanz der `CNN`-Klasse mit dem geladenen Netzwerk zu erzeugen. Basierend darauf wird ein Symbol, welches auf die `predict`-Methode dieser Instanz zeigt, für die weitere Verwendung zwischengespeichert.

Wenn nun ein oder mehrere Tiefenbilder verarbeitet werden sollen, werden diese zunächst von einer *column-major*- in eine *row-major*-Repräsentation überführt. NumPy bietet eine Programmierschnittstelle für C/C++ an und erlaubt es NumPy-Arrays direkt im C++-Code zu erzeugen (The SciPy community, 2019). Diese werden in einer Liste verpackt an die `predict`-Methode übergeben. Der Rückgabewert, also die Vorhersage des Netzwerks, wird mithilfe eben dieser Programmier-

schnittstelle in eine Liste von Vektoren konvertiert. Die jeweiligen Vorhersagen werden dann im Programm weiter verarbeitet.

3.7. Lösung des Matching-Problems

Wie in Abschnitt 2.14 auf Seite 21 beschrieben, wird die Zuordnung der Labels zu den Markern als ein gewichtetes bipartite Matching-Problem zwischen den n_x ursprünglichen Markern x_i und den n_y vom Netzwerk vorhergesagten Markern y_j dargestellt, mit $1 \leq i \leq n_x$ und $1 \leq j \leq n_y$. Analog zu (Han u. a., 2018a) wird dieses in ein Minimum-Cost-Flow Problem umgeformt. Zunächst wird nur der Fall betrachtet, wenn die Anzahl der ursprünglichen Marker genau 19 beträgt, also $n_x = 19$.

Nach einer Kommunikation mit den Autoren von (Han u. a., 2018a)[6] werden für die ursprünglichen Marker die in Abschnitt 3.5.1 auf Seite 35 als normalisiert bezeichneten Marker verwendet. Dies ersetzt das im folgenden beschriebene Verfahren.

Da sich die ursprünglichen Marker in der Rotation und Skalierung von der Netzwerkausgabe durch die Projektion unterscheiden, wird zunächst die durch die optische Achse gegebene Rotation auf die zentrierten ursprünglichen Marker angewendet. Dadurch wird die gleiche Blickrichtung zur Netzwerkausgabe erreicht. Die Marker der Netzwerkausgabe werden daraufhin ebenfalls um den Ursprung zentriert, da diese bisher zwischen 0 und 1 liegen. Daraufhin wird die Achse ermittelt, auf der die Marker die größte Ausdehnung d_o haben. Auf der selben Achse wird dann die Ausdehnung d_i der rotierten und zentrierten ursprünglichen Marker ermittelt. Die Netzwerkausgabe wird dann mit dem Faktor $\frac{d_o}{d_i}$ uniform skaliert, um die Wertebereiche aneinander anzunähern.

Die Verwendung von normalisierten Markern nutzt im Gegensatz dazu bereits bekannte Transformationen und liefert in Bezug auf die Netzwerkausgabe richtig rotierte und skalierte Marker.

Zur Darstellung von Graphen in der Implementierung wird die LEMON-Bibliothek in der Version 1.3.1 verwendet. Diese Programmierbibliothek stellt Datenstrukturen zum Darstellen von Graphen und Netzwerken sowie Algorithmen zur Lösung von Problemen in diesen bereit (Egerváry Research Group on Combinatorial Optimization (EGRES), 2014).

Um nun das Problem als einen Graphen $G = (V, E)$ darzustellen, werden zunächst, wie in Abschnitt 2.9.3 auf Seite 18 beschrieben, ein Quellknoten s und ein Zielknoten t eingeführt. Jeder ursprüngliche Marker wird durch einen Knoten x'_i und jeder Marker der Netzwerkausgabe durch einen Knoten y'_j repräsentiert. Es werden Kanten von s zu allen x'_i und von allen y'_j zu t eingeführt. Diese werden nachfolgend als Hilfskanten H bezeichnet. Alle diese Kanten $(a, b) \in H$ wird dann analog zu Abschnitt 2.9.3 eine Kapazität $c_{ab} = 1$ und Übertragungskosten $d_{ab} = 0$ zugeordnet. Dadurch haben diese neuen Kanten keinen Einfluss auf die eigentliche Zuordnung der Marker.

Zwischen den Markermengen $x_1, \ldots, x_{n_x}$ und $y_1, \ldots, y_{n_y}$ werden daraufhin ebenfalls Kanten eingeführt. Eine Kapazität von 1 erreicht hierbei, dass Marker nicht mehrfach einander zugeordnet werden und Markerpaare nicht mehr als einmal im Matching vorkommen. Analog zu (Han u. a., 2018a) werden als Übertragungskosten die Distanzen zwischen dem jeweiligen Markerpaar verwendet. Als Gesamtfluss wird die Anzahl an gesuchten Markerpaaren festgelegt. Diese ist durch die Anzahl der Marker in der Netzwerkausgabe gegeben. Für alle Kanten $(a, b) \in H$ wird darüber hinaus eine Untergrenze von 1 verwendet. Dadurch wird erreicht, dass über alle Hilfskanten mindestens ein Element übertragen werden muss[7]. Alle anderen Kanten haben eine Untergrenze von 0, da nicht

[6] Die Autoren antworteten auf eine E-Mail am 23. Juni 2019.

[7] Dies wird bereits durch den Gesamtfluss zusammen mit der Anzahl der jeweiligen Hilfskanten zu den Markermengen erreicht und war ein Versuch dem Solver mehr Informationen zu liefern. Dies wurde jedoch nicht weiter untersucht.

jede Kante für den Fluss verwendet wird. Im Folgenden wird der gebildete Graph zusammengefasst:

$$\begin{aligned} G &= (V, E) \\ V &= \{s, t\} \cup L \cup R \\ L &= \{x'_1, \ldots, x'_{n_x}\} \\ R &= \{y'_1, \ldots, y'_{n_y}\} \\ E &= H \cup (L \times R) \\ H &= (\{s\} \times L) \cup (R \times \{t\}) \end{aligned} \tag{3.20}$$

Die Parameter des Minimum-Cost-Flow Problems werden dann wie folgt festgelegt:

$$\begin{aligned} \forall (c, d) \in E.\ c_{ab} &= 1 \\ \forall (a, b) \in H.\ d_{ab} &= 0 \\ \forall i \in \{1, \ldots, n_x\}.\ \forall j \in \{1, \ldots, n_y\}.\ d_{x'_i y'_j} &= ||y_j - x_i||_2 \\ \forall (a, b) \in H.\ u_{ab} &= 1 \\ \forall i \in \{1, \ldots, n_x\}.\ \forall j \in \{1, \ldots, n_y\}.\ u_{x'_i y'_j} &= 0 \\ f &= n_y \end{aligned} \tag{3.21}$$

In Abbildung 3.6 ist die zugehörige graphische Repräsentation zu finden. Die Lösung des Minimum-Cost-Flow Problems übertragen auf das ursprüngliche Zuordnungsproblem liefert genau das Matching $\mathcal{M}$, das die kleinsten Distanzen zwischen den Markerpaaren im Vergleich zu allen anderen Matchings besitzt. $\mathcal{M}$ kann als eine Funktion betrachtet werden, die zu einem Index eines Markers aus der Netzwerkausgabe den Index des zugeordneten ursprünglichen Markers dieses Matchings liefert. Durch die Wahl der Übertragungskosten auf den Kanten zwischen L und R wird also folgende Summe durch die entsprechende Wahl von $\mathcal{M}$ minimiert (siehe Abschnitt 2.9.2 auf Seite 17 und (Han u. a., 2018a)):

$$\sum_{j=1}^{n_y} ||y_j - x_{M(j)}||_2 \tag{3.22}$$

Wie auch in (Han u. a., 2018a) ist es möglich mehr als 19 sichtbare Marker zu besitzen. Dabei handelt es sich nach (Ahuja u. a., 2014, S.7) nicht mehr um das Zuordnungsproblem, da die Anzahl der Knoten in der Menge L nicht mehr mit der Anzahl in der Menge R übereinstimmt. Der Netzwerkfluss-Graph wird so angepasst, dass die Untergrenze der Hilfskanten zwischen s und x_i auf 0 gesetzt wird. Dies erlaubt die Verwendung einer Teilmenge dieser Hilfskanten aus $\{s\} \times L$, dessen Elementzahl durch den Gesamtfluss gegeben ist. Bei dem daraus resultierenden Matching wird also die Teilmenge der Menge normalisierter Marker zu den vorhergesagten Markern zugeordnet und analog die Summe aus der Formel 3.22 minimiert. Im Folgenden ist die Anpassung der Untergrenze des Minimum-Cost-Flow Problems zu sehen:

$$\begin{aligned} \forall (a, b) \in (\{s\} \times L).\ u_{ab} &= 0 \\ \forall (a, b) \in (R \times \{t\}).\ u_{ab} &= 1 \\ \forall i \in \{1, \ldots, n_x\}.\ \forall j \in \{1, \ldots, n_y\}.\ u_{x'_i y'_j} &= 0 \end{aligned} \tag{3.23}$$

Zur Lösung des Minimum-Cost-Flow Problems wird der in LEMON (Egerváry Research Group on Combinatorial Optimization (EGRES), 2014) implementierte Algorithmus Capacity Scaling verwendet. Dieser liefert dabei den optimalen Netzwerkfluss mit den minimalen Kantenkosten unter den Einschränkungen der Kapazität und Untergrenzen. Aus diesem Netzwerkfluss wird dann das

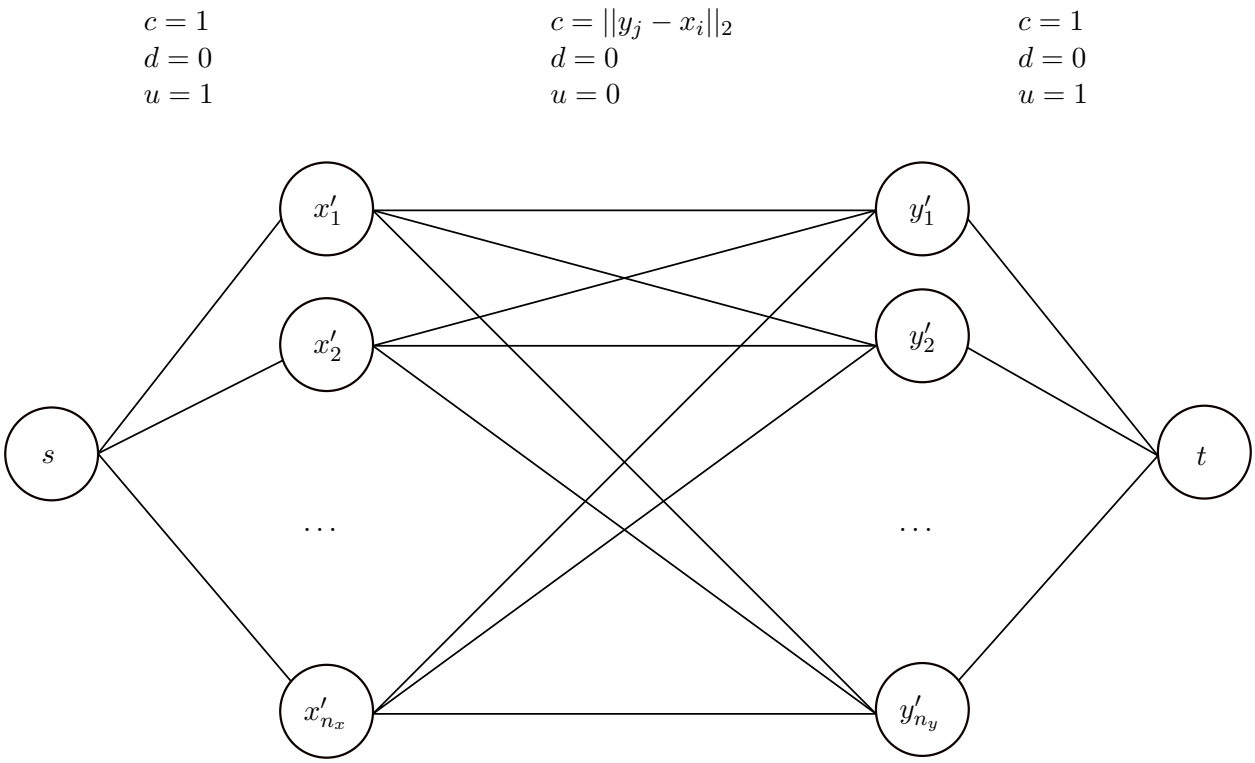

Abbildung 3.6.: Die graphische Darstellung des zum Minimum-Cost-Flow Problem gehörenden Graphen. Die Kosten der Kanten zwischen den Knoten aus L und R sind die Distanz zwischen den jeweiligen Markerpaaren.

zugehörige Matching ermittelt, indem der paarweise Fluss der Kanten zwischen den ursprünglichen und vorhergesagten Markern, also den Kanten aus $(L \times R)$, betrachtet wird. Ist dieser 1 für eine Kante sind die Knoten und somit auch die dazugehörigen Marker Teil des Matchings.

Die Implementierung zu diesem Abschnitt befindet sich in Quellcode-Dateien *marker_matching.h* und *marker_matching.cpp* sowie *marker_label.h* und *marker_label.cpp* des Teilprojekts *deep_labeling*. Die darin zentrale Funktion `solve_matching_problem` liefert zu den jeweiligen Markermengen eine geordnete Liste von Beschriftungen (*labels*). Diese geben an, wo sich der jeweilige normalisierte und somit auch ursprüngliche Marker auf der Hand befindet. Gibt es mehr als 19 ursprüngliche Marker, wird nicht zugeordneteten Markern eine Beschriftung *no_label* zugewiesen.

Auf eine alternative Lösung des Minimum-Cost-Flow Problems durch Linear Programming, wie es in (Han u. a., 2018a) umgesetzt ist, wurde verzichtet, da Algorithmen für das direkte Lösen dieses konkreten Problems bekannt sind und in zum Beispiel LEMON (Egerváry Research Group on Combinatorial Optimization (EGRES), 2014) implementiert sind.

3.8. Generierung fester Principal Axes

Um mehrere Blickrichtungen auf die Punktwolke konsistent abzubilden, werden feste optische Achsen mit verschiedenen Richtungen generiert. Dazu werden zunächst n Punkte $P = \{p_i \mid i \in \{1, .., n\};\ p_i \in \mathbb{R}^3\}$ gleichverteilt auf der Oberfläche einer Kugel generiert.

Ein dafür mögliches Verfahren ist in (Deserno, 2004) beschrieben. Dabei wird die Kugel in Scheiben mit einer Höhe von d_ϑ aufgeteilt. Auf diesen Scheiben werden dann Punkte mit einem Abstand von d_φ platziert. d_ϑ und d_φ werden so gewählt, dass $d_\vartheta * d_\varphi$ der durchschnittlichen Fläche pro Punkt auf der Oberfläche der betrachteten Kugel entspricht.

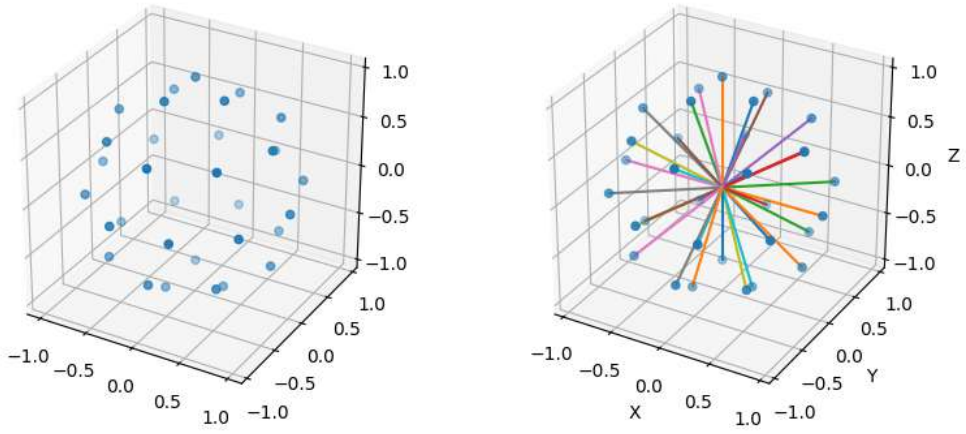

Abbildung 3.7.: Links: Platzierung von Punkten auf der Oberfläche einer Sphäre ($n = 32$). Rechts: Darstellung der daraus berechneten Principal Axes.

Für jeden Punkt $p \in P$ wird dann die Differenz zum Ursprung der Kugel bestimmt. Die normalisierte Differenz bildet eine Richtung für die Principal Axis ab. In Abbildung 3.7 ist eine Punkte-Platzierung sowie die darauf basierenden Principal Axes zu erkennen.

In den Abbildungen 3.8 und 3.9 ist die Implementierung zu finden.

```python
import numpy as np

def to_point(r, theta, phi):
    return [
        r * np.sin(theta) * np.cos(phi),
        r * np.sin(theta) * np.sin(phi),
        r * np.cos(theta),
    ]
```

Abbildung 3.8.: Umrechnung von Kugelkoordinaten in das kartesische Koordinatensystem. Dies ist eine Python-Implementierung des Pseudocodes aus (Deserno, 2004).

Die Generierung der Punkte auf einer Kugel ist in dem JupyterLab-Notebook *points_on_sphere.ipynb* im Teilprojekt *notebooks*, wie in Abbildungen 3.9 und 3.8 dargestellt, nach (Deserno, 2004) implementiert. Dieses Notebook enthält auch die Generierung der optischen Achsen aus diesen Punkten.

```python
def regular_placement(r, num_points):
    xs_regular = []
    ys_regular = []
    zs_regular = []

    n_count = 0
    a = 4 * np.pi * (r ** 2) / num_points
    d = np.sqrt(a)
    m_theta = np.round(np.pi / d)
    d_theta = np.pi / m_theta
    d_phi = a / d_theta

    for m in range(int(m_theta)):
        theta = np.pi * (np.float(m) + 0.5) / m_theta
        m_phi = np.round(2 * np.pi * np.sin(theta) / d_phi)
        for n in range(int(m_phi)):
            phi = 2 * np.pi * np.float(n) / m_phi

            point = to_point(r, theta, phi)
            xs_regular.append(point[0])
            ys_regular.append(point[1])
            zs_regular.append(point[2])

            n_count += 1

    return (xs_regular, ys_regular, zs_regular)
```

Abbildung 3.9.: Generieren von gleichverteilten Punkten auf der Kugeloberfläche. Dies ist eine Python-Implementierung des Pseudocodes aus (Deserno, 2004).

3.9. Labeln in Batches

Zusätzlich zur Auswahl einer optischen Achse analog zu Deep Labels werden Auswahlstrategien betrachtet, die mehrere Achsen erst nach dem Matching bewerten. Dies war notwendig, da die Reimplementierung von (Han u. a., 2018a), nicht die dort beschriebenen Ergebnisse erreichte und Probleme mit Markern einer linken Hand hatte (siehe Abschnitt 4.2.1 auf Seite 66).

Zunächst wird eine zuvor konfigurierte Anzahl an solchen Achsen generiert. Dabei werden drei verschiedene Verfahren betrachtet, wobei zwei zufallsbasiert sind. Ersteres generiert eine zufällige Blickrichtung und basierend darauf wird die Welt-Kamera-Transformation bestimmt. Zweiteres rotiert die zentrierte Punktwolke von Markerpositionen zufällig um alle drei Achsen und projeziert entlang der Tiefenachse.

Bei zufälligen Blickrichtungen kann es jedoch dazu kommen, dass ein Teil und insbesondere der Teil, wo das Netzwerk ein gutes Ergebnis liefert, nicht abgedeckt wird. Deswegen wird beim dritten Verfahren eine zuvor bestimmte Anordnung aus allen Blickrichtungen betrachtet, die in Abschnitt 3.8 beschrieben ist.

Jede dieser generierten Achsen liefert ein Tiefenbild und die normalisierten Markerpositionen. Alle Tiefenbilder werden an das CNN übergeben. TensorFlow erlaubt dabei die Verarbeitung verschiedener Eingaben mit einem einzigen Funktionsaufruf. Auf den dann vorliegenden Netzwerkausgaben und zugehörigen normalisieren Positionen werden die jeweiligen gewichteten bipartiten

Matching-Probleme gelöst.

Die für das beste Matching relevanten Kosten sind dabei die Summe aller Differenzen zwischen Netzwerkausgaben und den normalisierten Positionen. Basierend darauf werden die Zuordnungen bewertet und die mit den kleinsten Kosten wird zur Weiterverarbeitung ausgewählt.

Als ein alternatives Verfahren wird der Durchschnitt der Netzwerkausgaben für alle optischen Achsen als Eingabe ins Matching betrachtet. Dabei musste die durch die optische Achse bedingte Rotation der normalisierten Markerpositionen und somit auch der Netzwerkausgaben rückgängig gemacht werden. Da die jeweilige Rotation bereits bekannt ist, wird dazu einfach die Inverse der zugehörigen Rotationsmatrix verwendet.

Aufgrund der Probleme der ursprünglichen Implementierung mit der linken Hand (siehe 4.2.1 auf Seite 66), werden zusätzliche Tiefenbilder mit umgekehrter Tiefe zu den bereits generierten verwendet. Dies ist der Versuch die linke Hand als die rechte Hand darzustellen und sich somit die besseren Ergebnisse für die rechte Hand zu nutze zu machen.

Beim Labeling in Batches wurden 16, 32 und 64 optische Achsen pro Hand betrachtet. Dabei wurden diese Größen gewählt, da bei der Verarbeitung auf der GPU Arrays, dessen Größe eine Zweierpotenz ist, nach (Goodfellow u. a., 2016, S.272) bessere Performance aufweisen.

3.10. Rekonstruktion der Handkonfiguration über Inverse Kinematik

3.10.1. Handmodell

Das virtuelle Handmodell wird aus den im Abschnitt 2.10 auf Seite 18 beschriebenen Gelenken aufgebaut. Jeder Finger wird als eine Kette von Verbindungen und Gelenken repräsentiert. Jede Kette beginnt bei der Handwurzel, die eine globale Rotation und Position im Raum besitzt, wie in (Schröder u. a., 2014) und Abschnitt 2.11 auf Seite 19 beschrieben ist. Die Position wird als ein dreidimensionaler Vektor und die Rotation durch eulersche Winkel, wie in Abschnitt 2.5 auf Seite 12 beschrieben, dargestellt. Bei der Beschreibung des Handmodells wird nachfolgend von einem linkshändigen Koordinatensystem mit der Y-Achse nach oben gerichtet ausgegangen. In Abbildung 3.10 ist der Start für alle Gelenkketten zur Verdeutlichung auf einer echten Hand ungefähr markiert. Im Gegensatz zu (Han u. a., 2018a) wird nur die rechte Hand betrachtet.

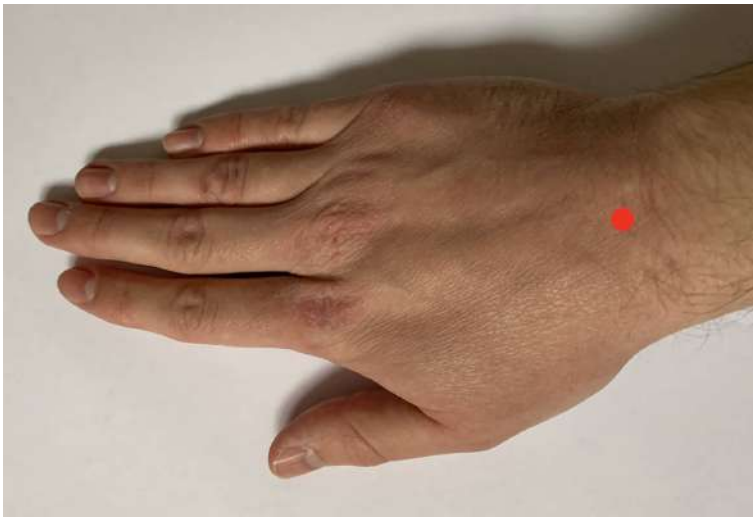

Abbildung 3.10.: Echte Hand mit markiertem Start aller Gelenkketten.

In der Kette werden die beiden Parameter a und d der jeweiligen Gelenke genutzt, um das jeweils nächste Gelenk zu positionieren. Die Positionierung in Bezug auf die letzte Achse wird durch eine

Verbindung mehrerer Gelenke umgesetzt. Das Gelenk was diese Verschiebung y erreicht, hat eine Rotation um $\alpha - 90°$[8] um die Gelenkachse und eine Verschiebung des Gelenks $d = y$. Das darauf folgende Gelenk muss diese Rotation wieder rückgängig machen, also $\alpha = 90$. Eine Verschiebung in der y-Achse wurde eingeführt, da sich die Knöchel der Finger sowie der Daumen in unterschiedlicher Höhe zur Handwurzel befinden.

In Abbildung 3.11 ist das verwendete Handmodell zu sehen. Es basiert vom Aufbau her auf dem Modell aus (Schröder u. a., 2014) mit einer Anpassung im CMC-Gelenk des Daumens. Ein Vergleich unterschiedlicher Handmodelle ist in 2.11 auf Seite 19 zu finden. Für die jeweiligen MCP-Gelenke der Finger II-V und das CMC-Gelenk des Daumens wird ein Kugelgelenk verwendet. Bei diesem Kugelgelenk wird die Rotation um die Gelenkachse jedoch nicht erlaubt. Dadurch bildet ein solches Gelenk im Handmodell die Seitwärtsbewegung und Beugung dieser Fingergelenke ab. Es lässt sich durch nur zwei Scharniergelenke mit einer festen Rotation um 90° ohne Verbindungslänge ($a = 0$) wie in Abschnitt 2.10 beschrieben darstellen. Analog zur zuvor beschriebenen Verschiebung muss diese Rotation im nächsten Gelenk wieder rückgängig gemacht werden. Alle anderen Gelenke sind einfache Scharniergelenke um ein Beugen der Finger zu realisieren.

Bei den Fingern wird die Verschiebung und die Seitwärtsbewegung der Kugelgelenke in einem Gelenk kombiniert. Beim Daumen wird die Rotation jedoch nicht vollständig rückgängig gemacht, um die Beugung in den Daumengelenken in 45° zur Hand umzusetzen, wie es in Abbildung 3.12 zu sehen ist. Dies stellt einen Unterschied zum Modell aus (Schröder u. a., 2014) dar.

Zur Vereinfachung der Darstellung wurde jeweils ein Gelenk für die Fingerspitzen eingeführt. Dies hat bei der Rekonstruktion der Handkonfiguration keinen Effekt.

Im implementierten Handmodell wird zwischen fixierten, freien und begrenzten Parametern unterschieden. Durch die Anpassung der freien und begrenzten Parameter, also die Bewegung der Hand- und Fingergelenke, können verschiedene Handpositionen umgesetzt werden. Die Tabelle 3.3 enthält die jeweiligen Gelenke und ihre Parameter aufgeschlüsselt nach ihrer Art, die das verwendete Handmodell bilden. Analog zu (Han u. a., 2018a) werden 26 anpassbare Parameter verwendet.

Die Parameter wurden teilweise direkt an der Hand gemessen (zum Beispiel die Höhe der Knöchel im Vergleich zur Handwurzel) und teilweise auf einem Bild der Hand (zum Beispiel die Knochenlänge zwischen den Gelenken). Der Skalierungsfaktor des Bildes wurde über die Distanz zwischen der Spitze des Mittelfingers und der Handwurzel ermittelt. Die Verkrümmung durch die Kameralinse wurde hierbei nicht beachtet.

Die Intervalle der begrenzten Parameter aus Tabelle 3.3 (Gelenklimits) für den Daumen wurden basierend auf der eigenen Hand geschätzt. Die Intervalle für die anderen Gelenke sind aus (Lin u. a., 2000) entnommen worden, jedoch ohne die Einschränkung der Seitwärtsbewegung im MCP-Gelenk des Mittelfingers (siehe Abschnitt 2.11 auf Seite 19 für die Intervalle). Aufgrund der Rotation durch vorhergehende Gelenke musste die Richtung der Begrenzung angepasst werden. Der Standardwert in der Ruheposition der Hand ist 0 für alle Gelenke bis auf die im Daumen. Im Folgenden sind die Standardwerte für die Daumenparameter zu finden:

$$\begin{aligned} \theta_1 &= -27.8° \\ \theta_2 &= -0.2\,\text{rad} \\ \theta_3 &= 0 \\ \theta_4 &= 0 \end{aligned} \tag{3.24}$$

Für die Rekonstruktion benötigt das Handmodell die Position der Marker auf dem Handschuh in der Standardkonfiguration. Dazu wurde mit dem Handschuh eine Ruheposition eingenommen. Diese ist ähnlich zu der Abbildung 3.10, jedoch wurde ein dünnes Buch zwischen Oberfläche und

[8] Eine Rotation in die entgegengesetzte Richtung ist auch möglich, in dem Fall muss d und das α des nächsten Gelenks ebenfalls angepasst werden.

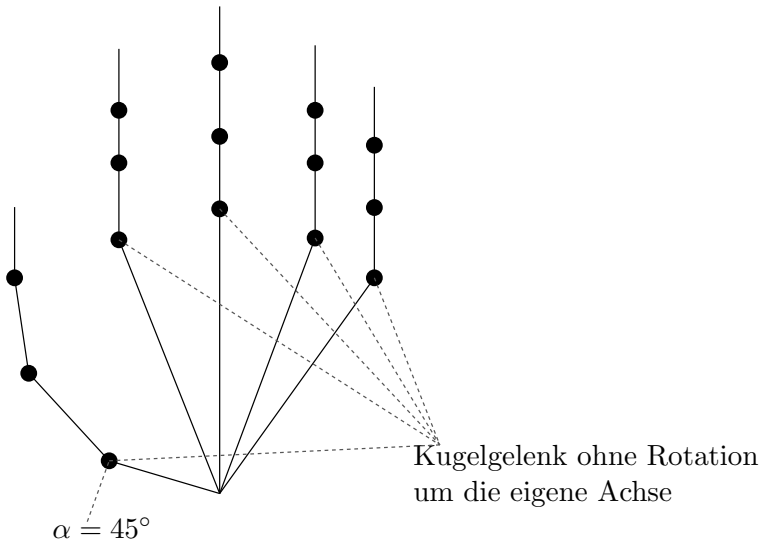

Abbildung 3.11.: Vereinfachte Visualisierung des Handmodells mit den virtuellen Gelenken die zur Repräsentation der echten Gelenke verwendet werden. Beim Daumen existiert beim CMC-Gelenk eine Rotation um 45° im Uhrzeigersinn, damit die Beugung in diesem und den nachfolgenden Gelenken etwa der echten Hand entspricht. Die PIP- und DIP-Gelenke bei den Fingern II bis V sowie MCP und IP beim Daumen sind einfache Scharniergelenke, um das Beugen der Finger und des Daumens zu realisieren. Die Gelenkbezeichnungen sind in Abbildung 2.11 auf Seite 19 zu finden.

Finger gelegt. Dies gleicht die leichte Rotation der Hand aus, wenn diese auf eine Fläche gelegt wird. Des Weiteren wurde darauf geachtet, dass die Finger keine Seitwärtsbewegung aufweisen. Dieses Frame wurde in Blender geöffnet, entsprechend der Standardkonfiguration positioniert und rotiert. Dadurch wurde versucht, eine möglichst gute Position der Marker auf der Hand zu finden. Beim Daumen mussten die Standardparameter der Hand manuell angepasst werden, da es bei der Aufnahme schwierig war den Daumen entsprechend der Standardkonfiguration zu positionieren.

Jedem Marker wird ein Gelenk zugeordnet, um bei Handkonfigurationen durch Linear Blend Skinning (*LBS*) die entsprechende Markerposition zu ermitteln (Han u. a., 2018a). Bei dieser Zuordnung wird als Gewicht beim LBS für das jeweilige Gelenk 1 und für alle anderen Gelenke 0 verwendet. Das heißt, dass immer nur ein Gelenk eine Markerposition beeinflusst. Es wird die in Abschnitt 2.14 auf Seite 21 beschriebene Formel verwendet.

Die Markerpositionen in der Ruheposition und Gelenkzuordnungen sind in der Tabelle 3.4 zu finden. Da sich das Gelenk mit dem Index 1 bei Anpassung von Parametern nicht bewegt, werden diesem Gelenk auch die Marker auf dem Handrücken zugeordnet. Dadurch muss kein Gelenk für die Handwurzel eingeführt werden und die globale Rotation sowie Position, die durch die freien Parameter gegeben sind, können direkt bei der Transformation der jeweils ersten Kettenglieder verwendet werden.

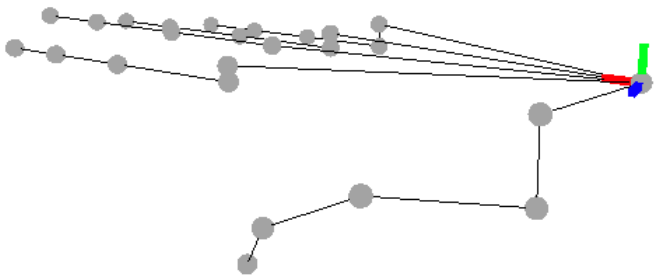

Abbildung 3.12.: Visualisierung des Handmodells in der Implementierung mit gebeugtem Daumen. Die Gelenke werden durch die grauen Sphären dargestellt.

Finger	**#**	**Gelenk**	a	α	d	θ	**Parameter (Hinweis)**
HW	-	-	-	-	-	-	6 (Rotation, Position)
	0	CMC	1.5496	0	3.8883	0	0 (XY-Verschiebung)
	1	CMC	0	$-90°$	-2.5	0	0 (Z-Verschiebung)
	2	CMC	0	0	0	$[0°, -45°]$	1 (seitwärts)
I	3	CMC	0	$-45°$	0	$[-0.2\,\text{rad}, 0.6\,\text{rad}]$	1 (Kippgelenk 1)
	4	MCP	4.5	0	0	$[0°, 30°]$	1 (kippen)
	5	IP	3.5	0	0	$[-10°, 80°]$	1 (kippen)
	6	-	3.2	0	0	0	0 (Daumenspitze)
	7	MCP	11.2264	0	3.7456	0	0 (XY-Verschiebung)
	8	MCP	0	$90°$	0.5	0	0 (Z-Verschiebung)
	9	MCP	0	0	0	$[15°, -15°]$	1 (seitwärts)
II	10	MCP	0	$-90°$	0	$[-90°, 0°]$	1 (kippen)
	11	PIP	4.3596	0	0	$[-110°, 0°]$	1 (kippen)
	12	DIP	2.687	0	0	$[-90°, 0°]$	1 (kippen)
	13	-	1.926	0	0	0	0 (Fingerspitze)
	14	MCP	11.4803	0	1.1472	0	0
	15	MCP	0	$90°$	0	0	0
	16	MCP	0	0	0	$[15°, -15°]$	1
III	17	MCP	0	$-90°$	0	$[-90°, 0°]$	1
	18	PIP	4.1267	0	0	$[-110°, 0°]$	1
	19	DIP	3.407	0	0	$[-90°, 0°]$	1
	20	-	2.306	0	0	0	0
	21	MCP	10.7185	0	-1.1216	0	0
	22	MCP	0	$90°$	0.5	0	0
	23	MCP	0	0	0	$[15°, -15°]$	1
IV	24	MCP	0	$-90°$	0	$[-90°, 0°]$	1
	25	PIP	3.7875	0	0	$[-110°, 0°]$	1
	26	DIP	3.196	0	0	$[-90°, 0°]$	1
	27	-	2.158	0	0	0	0
	28	MCP	9.9567	0	-3.1319	0	0
	29	MCP	0	$90°$	0.8	0	0
	30	MCP	0	0	0	$[15°, -15°]$	1
V	31	MCP	0	$-90°$	0	$[-90°, 0°]$	1
	32	PIP	3.0263	0	0	$[-110°, 0°]$	1
	33	DIP	2.37	0	0	$[-90°, 0°]$	1
	34	-	2.074	0	0	0	0
Gesamtzahl an anpassbaren Parametern							26

Tabelle 3.3.: Die Gelenkketten und Parameter für das Handmodell. Feste Parameter werden durch einen normalen Wert und begrenzte Parameter durch ein Intervall gekennzeichnet. Es gibt nur sechs freie Parameter, die die globale Position und Rotation der Hand angeben. Die Spalte „Gelenk“ enthält das Gelenk der Hand, das zum Gelenk in der Implementierung gehört. Im Handmodell wird ein linkshändiges Koordinatensystem mit der Y-Achse nach oben verwendet. Die Y-Verschiebung ist bei allen Fingern außer beim Daumen aufgrund des α-Parameters invertiert. Jedem Gelenk wurde eine eindeutige Identifikationsnummer in der Spalte „#“ zugewiesen. *HW* steht für die Handwurzel.

Markername	**#**	**X**	**Y**	**Z**	**Gelenkindex**
thumb_base1	1	2.271	-0.93413	5.0637	3
thumb_base2	2	3.9175	-0.41223	6.9042	3
thumb_middle	3	6.6417	-0.11661	8.8166	4
thumb_tip	4	9.6707	0.68882	10.788	5
index_knuckle	5	12.322	3.0976	3.4306	10
index_middle	6	16.355	2.3656	3.5248	11
index_tip	7	19.06	2.3056	3.484	12
middle_knuckle	8	12.386	3.2432	1.0328	17
middle_middle	9	16.753	2.7064	1.5208	18
middle_tip	10	20.091	2.2995	1.2094	19
ring_knuckle	11	12.18	3.0114	-1.3809	24
ring_middle	12	16.26	2.5355	-1.0043	25
ring_tip	13	18.547	2.2026	-1.3191	26
little_knuckle	14	11.297	1.892	-4.2074	31
little_middle	15	13.863	1.2495	-3.2834	32
little_tip	16	17.056	1.5654	-3.5861	33
hand_base_little	17	8.5672	3.7576	-1.1109	0
hand_base_wrist	18	5.1201	4.6741	-1.9046	0
hand_base_thumb	19	6.1431	5.2004	2.8206	0

Tabelle 3.4.: Positionen der zum Handmodell gehörenden Marker. Als Ursprung des Koordinatensystems wird der in Abbildung 3.10 markierte Punkt, also der Start aller Gelenkketten verwendet. Als Namen für die Marker werden die in Abschnitt 3.4 eingeführten Bezeichner verwendet.

3.10.2. Performance des LBS

Da die Gelenkketten voneinander unabhängig berechnet werden können, wird versucht die Berechnung der Markerpositionen beim LBS durch OpenMP (OpenMP Architecture Review Board, 2018) zu beschleunigen. Dabei wird `#pragma omp for` für die Schleife über die Gelenkketten und `#pragma omp critical` für die Summierung der jeweiligen gewichteten Markerpositionen verwendet. Letzterer Befehl ist nur notwendig, wenn ein Marker von mehreren Gelenken abhängt, und wird im Hinblick auf spätere Anpassungen beziehungsweise Erweiterungen hinzugefügt. Die Performance wurde in einem Benchmark mit Hilfe der Google-Benchmark-Bibliothek[9] gemessen (*inverse_kinematics_benchmark.cpp* in Teilprojekt *benchmarks*). Dabei wurde die Laufzeitperformance von unabhängigen Aufrufen der LBS-Methode betrachtet, da diese bei der Rekonstruktion ebenso durch den Solver aufgerufen wird. Wie in Abschnitt 4.2.3 auf Seite 68 zu sehen ist, hat diese Art der Parallelisierung einen negativen Effekt und wird nicht weiter verwendet.

3.10.3. Bewertung einer Rekonstruktion

In (Han u. a., 2018a) wird für beim Verwerfen von schlechten Frames das quadratische Mittel des Fehlers betrachtet. Die Summe der quadratischen Fehler wird dabei in dieser Implementierung vom Solver geliefert. Liegt dieses über $8mm$, wird das jeweilige Frame verworfen (siehe Abschnitt 2.14 auf Seite 21). Dieses Maß wird beim Betrachten der in den folgenden Abschnitten beschriebenen Verfahren hinzugezogen.

3.10.4. Einstufige Rekonstruktion

Als zu optimierenden Fehlerterm wird hier E_{IK} aus (Han u. a., 2018a) betrachtet. Die Definition ist in Abschnitt 2.14 auf Seite 21 zu finden.

Die Optimierung der Handparameter für das aktuelle Frame wird durch einen in der Eigen-Bibliothek (Jacob u. Guennebaud, 2018a,b) implementierten Levenberg-Marquardt-Solver analog zu (Han u. a., 2018a) durchgeführt. Dabei wurde als Dokumentation ein Beispiel zu diesem Solver aus (Jacob u. Guennebaud, 2018b) herangezogen. Die zu optimierende Funktion berechnet den zuvor beschriebenen Fehlerterm basierend auf den aktuellen Parametern und die dafür benötigte Ableitung wird durch die in Eigen (Jacob u. Guennebaud, 2018b) implementierte NumericalDiff-Klasse geschätzt. Für die initialen Parameter wird wie in (Han u. a., 2018a) die Ruheposition verwendet, die in Abschnitt 3.10.1 beschrieben und für den Daumen in Formel 3.24 zu finden ist. Der genutzte Solver erwartet einen Fehlervektor dessen Elemente den Summanden der zur optimierenden Summe entspricht (Jacob u. Guennebaud, 2018b).

Da in den vorhergehenden Schritten von einer Z-Achse nach oben ausgegangen wird, müssen die Markerpositionen vor der Rekonstruktion der Handkonfiguration entsprechend angepasst werden.

Überstreckung

Bei dem aktuell verwendeten Handmodell wurden beim Testen dieses Verfahrens festgestellt, dass die resultierenden Handkonfigurationen größtenteils ungültig waren, da sie starke Überstreckungen in den Gelenken aufwiesen. In (Schröder u. a., 2014) wird das Handmodell zur Behebung dieses Problems im Rahmen eines anderen Rekonstruktionsverfahrens durch Gelenklimits beschränkt. Aus diesem Grund wird der zuvor beschriebene Fehlerterm um einen Überstreckungsfehler E_{over} ergänzt. Zur Berechnung des Überstreckungsfehlers werden die aktuellen Parameterwerte θ betrachtet. Ist der Wert außerhalb des für den Parameter θ_i spezifizierten Intervalls $[a_i, b_i]$, wird die Differenz zum Start beziehungsweise Ende des Intervalls berechnet. Damit größere Überstreckungen den Fehler stärker steigen lassen, wird das Quadrat dieser Differenz verwendet. Analog dazu wird auch die

[9] Siehe (Google, 2019a).

Summe aller dieser Teilfehler quadriert. Der Überstreckungsfehler setzt sich also folgendermaßen zusammen:

$$
\begin{aligned}
E_{over} &= \left(\sum_{\theta_i} error(\theta_i, a_i, b_i) \right)^2 \\
error(x, a, b) &= \begin{cases} (a - x)^2 & \text{falls } x \leq a \\ (x - b)^2 & \text{falls } x \geq b \\ 0 & \text{sonst} \end{cases} \\
E'_{IK} &= E_{IK} + E_{over}
\end{aligned}
\tag{3.25}
$$

3.10.5. Zweistufige Rekonstruktion

Wie in Abschnitt 4.2.2 auf Seite 67 erläutert, schlägt die Rekonstruktion fehl, sobald sich die aufgenommene Hand nicht in der Nähe des Ursprungs befindet. Die Entscheidung fiel dann auf die Verwendung eines zweistufigen Verfahrens, wobei zunächst die globale Position sowie Rotation bestimmt und dann nur die 20 Parameter der Fingerkonfiguration optimiert werden. Somit werden die beschriebenen Probleme vorgebeugt. Dies ist möglich, da zwischen der globalen Transformation und der Fingerkonfiguration eine gewisse Unabhängigkeit besteht. Jedoch ist diese zweistufige Methode anfälliger gegenüber Rauschen in den Markern, durch die diese Transformation bestimmt wird. Dies liegt daran, dass bei der ursprünglichen Methode der Fehler aller Marker gleichzeitig betrachtet wird (siehe Abschnitt 3.10.4).

Dazu wird die in Abschnitt 2.18 auf Seite 24 beschriebene Methode zur Bestimmung der Transformation zwischen zwei Punktwolken verwendet. Die jeweiligen Punktwolken sind die aufgenommenen, zugeordneten Marker sowie die Marker des Handmodells. Dabei werden jedoch nur die drei Marker betrachtet, die sich auf dem Handrücken befinden.

Zur Berechnung der Singulärwertzerlegung wurde die Bibliothek Eigen (Jacob u. Guennebaud, 2018a) verwendet. Falls der in Abschnitt 2.18 auf Seite 24 beschriebene Reflektionsfall auftritt, werden die Einträge die letzte Spalte der Matrix V in jedem Fall negiert ohne die Singulärwerte zu prüfen. Dies war nötig, da sonst nicht alle Reflektionsfälle in der aktuellen Implementierung richtig berechnet wurden. Obwohl dies nach (Arun u. a., 1987) auf zu viel Rauschen hindeutet, konnte beim Testen mit echten Daten bestätigt werden, dass die Bestimmung der Transformationen erfolgreich war.

Die daraus bestimmte Rotation X und Translation T werden für die globale Rotation beziehungsweise Translation der Hand verwendet. Die Hand wird dadurch so positioniert, dass sich die Handmodell-Marker auf dem Handrücken den jeweiligen aufgenommenen Markern annähern. Dabei wird die Rotation der Hand direkt durch eine Rotationsmatrix und nicht mehr durch eulersche Winkel repräsentiert.

3.10.6. Unabhängige Rekonstruktion der Finger

Die Bewegung einzelner Finger ist von anderen Fingern unabhängig. Aus diesem Grund wird das zweistufigen Verfahren erweitert und die Optimierung der Fingerkonfiguration, anstatt für alle Finger gleichzeitig, für jeden Finger einzeln durchgeführt. Dabei müssen dann nur jeweils vier Parameter bestimmt werden und der Fehler wird nur auf den für den jeweiligen Finger relevanten Markern berechnet.

3.10.7. Wiederverwendung bisheriger Frames

Die Wiederverwendung von Handparametern aus vorhergehenden Frames als initiale Parameter bei der Rekonstruktion im aktuellen Frame analog zu (Han u. a., 2018a) ist zwar implementiert, wurde jedoch nicht auf echten Daten getestet.

3.10.8. Manuelle Optimierung der Markerpositionen

Die Wahl eines Handmodells und den entsprechenden Markerpositionen ist relevant für ein gutes Ergebnis bei der Rekonstruktion. In (Han u. a., 2018a) wird eine Optimierung dieser Markerpositionen für den aktuellen Nutzer durchgeführt. In dieser Arbeit wird zugunsten anderer Teilbereiche darauf verzichtet und stattdessen auf eine teilweise automatisierte Optimierung während der Entwicklungszeit zurückgegriffen.

Es wurden eine gleichzeitige Translation aller Marker in Y- und Z-Richtung von ihrer ursprünglichen Ruheposition, also zwei Parameter, untersucht. Die daraus resultierenden Positionen werden dann als die eigentliche Ruheposition bei der Rekonstruktion verwendet. Das System wird auf den aufgenommenen Daten mit verschiedenen Parametern ausgeführt und dann die Parameter ausgewählt, die die meisten gültigen Frames nach dem RMS-Fehler liefern.

In zwei Stufen wird erst ein grober und dann ein feiner Bereich betrachtet. Dazu werden Parameterwerte mit einer gewissen Schrittgröße aus den Intervallen generiert. In Tabelle 3.5 sind die verwendeten Intervalle und Schrittgrößen zu finden. Die Auswahl dieser Intervalle basierte auf einer zuvor manuell und willkürlich bestimmten Translation, die bessere Ergebnisse als keine Translation liefert.

Art	**Y**	**Y-Schritt**	**Z**	**Z-Schritt**
Grob	$[1, 2.2]$	0.2	$[-1, 1]$	0.2
Fein	$[0.5, 1.8]$	0.1	$[-1.4, 0.5]$	0.1

Tabelle 3.5.: Die betrachteten Intervalle bei der Optimierung der Markerpositionen des Handmodells in Abhängigkeit von der Art des Suchbereichs. Die Spalten **Y** und **Z** enthalten die betrachteten Intervalle für die jeweilige Translation in diesen Dimensionen. **Y-Schritt** und **Z-Schritt** beinhalten die entsprechenden Schrittweiten in diesen Intervallen.

3.11. Training von Netzwerken

3.11.1. Modell

Im Folgenden wird die in Abschnitt 3.6.3 auf Seite 38 bereits beschriebene Netzwerkarchitektur verwendet. Dabei wird auf die zusätzliche Permute-Ebene vor der ersten vollständig verbundenen Ebene verzichtet. Bei Batch-Normalization-Ebenen wurde aus Performance-Gründen die *fused*-Variante verwendet. Analog dazu wurde die Aktivierungsfunktion in die erste vollständig verbundene Ebene integriert. Das Training der Netzwerke wurde in JupyterLab-Notebooks durchgeführt.

3.11.2. Trainingsdaten

Die zum TensorFlow-Framework gehörende Keras-Schnittstelle benötigt beim Lernen die Trainingsdaten entweder vollständig oder direkt in Minibatches für den SGD-Algorithmus.

Die in Han u. a. (2018a) veröffentlichten Trainingsdaten liegen als Tracking-Aufnahmen mit richtig zugeordneten Marker-Positionen vor. Beim Lernen werden 168691 Frames verwendet.

Dies steht im Kontrast zu der in Han u. a. (2018a) angegebenen Anzahl von 170330 Frames. Bei der Verarbeitung der Trainingsdaten wurde festgestellt, dass die Anzahl von Frames in den

Metadaten der jeweiligen Dateien nicht übereinstimmt. Selbst dann liegt die Zählung bei nur 170180 Frames. Es fehlen in den mitgelieferten Trainingsdaten entweder 1639 Frames oder die im Paper angegebene Zahl ist nicht korrekt.

Zunächst werden die Trainingsdaten analog zu Deep Labels vorverarbeitet. Es wird eine zufällige Blickrichtung bei der Projektion jedes Tiefenbildes und Berechnung der normalisierten Markerpositionen verwendet. Diese Tiefenbilder werden als Eingabedaten und die jeweiligen normalisierten Markerpositionen als Zieldaten für das Modell beim Trainieren verwendet. Zusätzlich werden für jede Epoche jeweils andere Blickrichtungen verwendet. Die Datensätze für alle Epochen werden vor dem Trainingsbeginn generiert. Dies hat den Vorteil, dass zur Laufzeit keine größeren Vorarbeitungsschritte notwendig sind. Der Export der Trainingsdaten geschieht über eine in C++ eingebundene Python-Methode (siehe Abschnitt 3.6.4 auf Seite 40) und ist in *__init__.py* im Teilprojekt *python_based_utils* implementiert.

Zusätzlich wird auf Vorschlag der Arbeitsgruppe ein Zentrieren der Ein- und Ausgabedaten um 0 mit einer Normalisierung in ded Wertebereich $[-0.5, 0.5]$ betrachtet.

3.11.3. Vorbereitung des Modells

Vor dem Lernen muss in TensorFlow das Modell erst dazu vorbereitet beziehungsweise kompiliert werden. Dabei werden der Optimierungsalgorithmus, die Verlustfunktion sowie weitere Metriken spezifiziert (Abadi u. a., 2015, tf.keras.Model).

3.11.4. Hyperparameter bei der Optimierung

Analog zu Deep Labels wird dabei der Stochastic Gradient Descent mit Minibatches verwendet. Die Größe der Minibatches wurde auf 256 gesetzt. Das Netzwerk wird für 75 Epochen trainiert (Han u. a., 2018a, S.6).

Die in Deep Labels verwendete Fehlerfunktion ist nicht bekannt. Das VGG-Netzwerk, auf welches verwiesen wird, nutzt für Lokalisierungsprobleme einen euklidischen Fehler (Simonyan u. Zisserman, 2015, S.10). Aus diesem Grund wird als Verlustfunktion die mittlere quadratische Abweichung (*mean squared error*; *MSE*) verwendet. Diese steigt, wenn sich die euklidische Distanz zwischen der Vorhersage und dem erwarteten Positionen erhöht (Goodfellow u. a., 2016, S.105).

Die in Deep Labels genutzte Lernrate zum Start des Lernvorgangs führte jedoch dazu, dass sich der Trainingsfehler nach wenigen Epochen nicht mehr verringerte. In Abschnitt 3.11.5 wird auf die Auswahl einer eigenen Lernrate eingegangen. Diese wird dann, wie auch bei Deep Labels jeweils um einen Faktor von 10 nach der 50. und 70. Epoche reduziert. Das Moment des SGD ist nicht in Han u. a. (2018a) spezifiziert. Da ein VGG-artiges Netzwerk verwendet wird, fiel die Entscheidung auf ein Moment von 0.9 (Simonyan u. Zisserman, 2015, S.4). Auf die Nesterov-Variante des Moments wird hier verzichtet, da dieser bei SGD mit Minibatches die Konvergenz nicht verbessert (Goodfellow u. a., 2016, S.291f.).

Aufgrund der Probleme mit der in Deep Labels beschriebenen Lernrate wurden zwei weitere, in (Lathuilière u. a., 2019, S.7) für Regressions-Probleme empfohlene, Verlustfunktionen betrachtet. Dabei handelt es sich um den mittleren absoluten Fehler (*mean absolute error*; *MAE*) und den Huber-Verlust. Die letzte ist zwar in der aktuellen Version in der High-Level-Schnittstelle Keras von TensorFlow implementiert, die in dieser Arbeit genutzte Version enthält jedoch nur eine direkte TensorFlow-Implementierung, die an Keras angepasst verwendet wird. Dabei wird der Verlust über die Batches gemittelt betrachtet[10]. In der Abbildung 3.13 ist diese Anpassung zu finden (Abadi u. a., 2015, tf.keras.losses.Loss, tf.losses.huber_loss).

Des Weiteren werden auch wegen den zuvor beschriebenen Problemen mit der Lernrate die zwei Optimierungsalgorithmen Adam und AdaDelta betrachtet, die bei Optimierungsproblemen empfoh-

[10] Dies orientiert sich an der Implementierung dieses Verlusts in neueren Versionen von (Abadi u. a., 2015).

```
class HuberLoss(losses.Loss):
    def call(self, y_true, y_pred):
        super().__init__(name="Huber loss")
        reduction = tf.losses.Reduction.SUM_OVER_BATCH_SIZE
        return tf.losses.huber_loss(y_true, y_pred,
                                    reduction=reduction)
```

Abbildung 3.13.: Anpassung der Huber-Verlustfunktion an die Keras-Schnittstelle.

len werden (Lathuilière u. a., 2019, S.4). Diese Algorithmen sind in Abschnitt 2.8.8 auf Seite 16 kurz beschrieben. Dabei wurden Standard-Parameter verwendet. Zur Zeit der Implementierung sind es für Adam eine Lernrate von 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-7$, $decay = 0.0$ und keine Verwendung der AMSGrad-Erweiterung (Abadi u. a., 2015, Dokumentation zu *tf.keras.optimizers.Adam*). Für AdaDelta sind es eine Lernrate von 0.001, $\rho = 0.95$ und $\epsilon = 1e-7$ (Abadi u. a., 2015, Dokumentation zu *tf.keras.optimizers.AdaDelta*).

Darüber hinaus wird als Metrik MAE betrachtet, der Auskunft darüber gibt, wie sich die Marker-Koordinaten zwischen Vorhersage und den Trainingsdaten in jeder Achse durchschnittlich Unterscheiden. Diese Metrik wurde nicht hinzugezogen, wenn MAE als Verlustfunktion verwendet wurde. Diese werden auf den Trainingsdaten evaluiert, haben jedoch keinen Effekt auf die Optimierung (Abadi u. a., 2015, Dokumentation zu *tf.keras.Model*).

3.11.5. Auswahl der Lernrate für MSE

Zur Auswahl der Lernrate wird der Fehler auf dem Trainingsdaten für verschiedene Lernraten nach zehn Epochen betrachtet. Die kleine Anzahl an Epochen ermöglicht es mehr Lernraten zu vergleichen. In Abbildung 3.14 sind die betrachteten Lernraten zu finden. Dies wurde aufgrund des bekannten Verhaltens von Trainingsfehlern und Lernraten, wie in Abschnitt 2.7.4 auf Seite 14 beschrieben, durchgeführt. Dies basiert auf der in (Goodfellow u. a., 2016, S.420f.) beschriebenen Gittersuche nach Hyperparametern.

Beim vollständigen Lernen mit der MSE-Verlustfunktion wurden die Lernraten 0.06 und 0.02 betrachtet. Die Erste wurde gewählt, da es sich dabei eine um den Faktor 10 verkleinerte Lernrate von Deep Labels handelt. Die Letztere liefert den kleinsten Trainingsfehler im Vergleich zu anderen Lernraten.

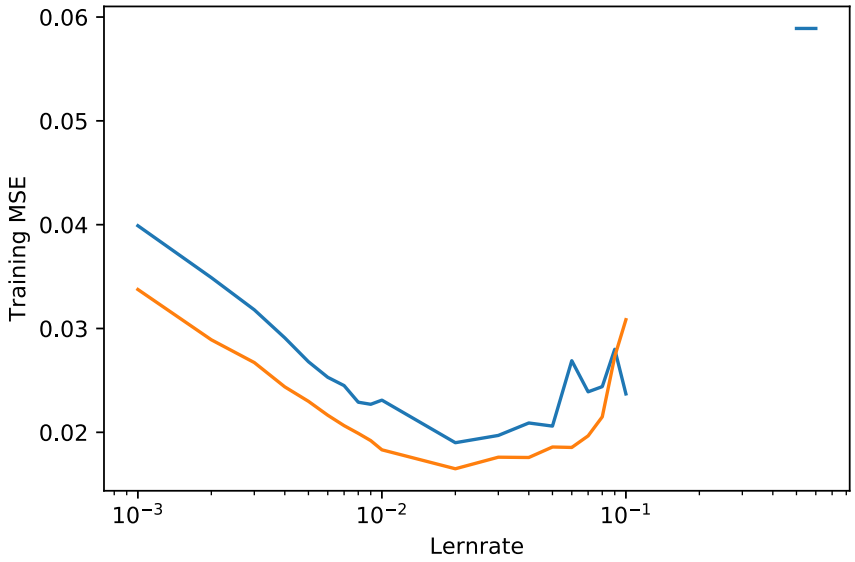

Abbildung 3.14.: Die betrachteten Lernraten mit den jeweiligen Trainingsfehlern nach zehn Epochen.

3.11.6. ResNet-50

Als eine alternative Netzwerkstruktur wurde das in Abschnitt 2.8.9 auf Seite 16 kurz beschriebene ResNet-50 betrachtet. Neuronale Netze dieser Netzwerkarchitektur erreichten erste Plätze in verschiedenen Wettbewerben zur Bilderkennung (He u. a., 2016). Dies ist neben dem VGG-Netzwerk (Simonyan u. Zisserman, 2015) eins der am häufigsten zitierten Architekturen und kann mit einer Anpassung für Regression verwendet werden (Lathuilière u. a., 2019).

TensorFlow bietet eine konfigurierbare Implementierung von ResNet-50 an (Abadi u. a., 2015), die in dieser Arbeit verwendet wird. Da TensorFlow jedoch kaum kaum Dokumentation zur Verwendung enthält, wurde dabei die Dokumentation der High-Level-Schnittstelle Keras (Chollet u. a., 2019) sowie der Quellcode davon herangezogen.

Bei der Initialisierung des Netzwerks lässt sich eine vollständig verbundene Ebene als letzte Netzwerkebene hinzunehmen. Auf diese wird analog zu (Lathuilière u. a., 2019) verzichtet und es wird eine später in diesem Abschnitt beschriebene Regressionsebene genutzt. Es gibt die Möglichkeit die Gewichte eines bereits für ImageNet trainierten Netzwerks zu verwenden. Dies ist laut (Lathuilière u. a., 2019) üblich für ImageNet-basierte Regression. Da sich der aktuelle Einsatzzweck deutlich davon unterscheidet, wird die Initialisierung des Netzwerks so angepasst, dass diese zufällig generiert werden. In diesem Fall werden von Keras verschiedene Verfahren für die jeweiligen Ebenentypen verwendet. Diese sind in (Abadi u. a., 2015, Quellcode zu *tf.keras.applications.ResNet50*) sowie (Chollet u. a., 2019, Quellcode: *applications*-Modul) zu finden und in Tabelle 3.6 aufgeführt. Die Pooling-Ebene des ResNet-50 lässt sich ebenfalls konfigurieren. Es wird das in (Lathuilière u. a., 2019) empfohlene und in der ursprünglichen ResNet-Veröffentlichung (He u. a., 2016) verwendete Global Average Pooling verwendet. Ersteres Paper spricht noch die Verwendung einer Global Max Pooling Ebene an. Der Effekt dieser Veränderung wird untersucht.

Analog zu Abschnitt 3.6.4 auf Seite 40, Abschnitt 3.11.2 und (Han u. a., 2018a) wird als Eingabe ein Tiefenbild und als erwartete Ausgabe die normalisierten Marker verwendet. Aus diesem Grund wird das Eingabeformat des ResNet-50 auf $52 \times 52 \times 1$ gesetzt.

Als Regressionebene wird basierend auf (Lathuilière u. a., 2019) eine vollständig verbundene Ebe-

[11] Dieses Verfahren ist in TensorFlow nach (He u. a., 2015) implementiert.

Art der Gewichte	Initialisierung der Gewichte
Convolution (*kernel*)	*he_normal*[11]
Convolution (*bias*)	0
BatchNormalization (*beta*, *moving average*)	0
BatchNormalization (*gamma*, *moving variance*)	1

Tabelle 3.6.: Die beim Erzeugen von ResNet-50 basierten Netzwerken in TensorFlow verwendeten Verfahren zur Initialisierung von Gewichten (Abadi u. a., 2015, Dokumentation, Source-Code der ResNet-50-Implementierung).

ne mit einer Ziel-Dimension von 57 entsprechend dem Ausgabeformat verwendet. Analog zu der letzten Ebene des in Abschnitt 3.6.3 auf Seite 38 beschriebene Netzwerks wird eine Reshape-Ebene verwendet, die den eindimensionalen Tensor mit 57 Elementen in einen mit den Dimensionen 19×3 konvertiert, sodass das Ausgabeformat den Positionen der 19 normalisierten Markern entspricht. Dabei werden zur zufälligen Generierung der Gewichte und den Bias die Standardwerte von TensorFlow beziehungsweise Keras verwendet. Die Dokumentation zu (Abadi u. a., 2015) gibt an, dass dabei für die Initialisierung der Gewichte (*kernel*) das Glorot-Verfahren (siehe Abschnitt 2.8.7 auf Seite 16) und für den Bias 0 verwendet wird. Da dieses Verfahren für VGG-Netzwerke nach (Simonyan u. Zisserman, 2015) gut geeignet war und das Netzwerk aus (Han u. a., 2018a) darauf basiert, fiel die Entscheidung darauf, diesen Standardwert beizubehalten.

Das Anbinden der neuen Ebenen zum bestehenden Netzwerk ist in Keras einfach, da nur angegeben werden muss, dass diese auf die Ausgabe des ResNet-50 angewendet werden sollen. Im Abschnitt 2.19.4 auf Seite 27 wird auf das Bilden und Verketten von Netzwerken eingegangen. Es wird ein übergeordnetes Netzwerk erzeugt, dessen Eingabe der ResNet-Eingabe und dessen Ausgabe der Reshape-Ebene entspricht. In Abbildung 3.15 wird der Aufbau des neuen Netzwerks verdeutlicht und Abbildung 3.16 zeigt den dazugehörigen Quellcode.

Training

Bei der Optimierung des in Abschnitt 3.11.6 beschriebenen auf ResNet-50 basierenden Netzwerks werden analog zu Abschnitt 3.11.4 die in (Lathuilière u. a., 2019, S.4) empfohlenen Optimierungsalgorithmen Adam und AdaDelta mit Standardeinstellungen betrachtet. Auf die Verwendung von einfachem Stochastic Gradient Descent wurde verzichtet, da dabei eine manuelle Anpassung der Lernrate beim Lernen notwendig ist (siehe Lathuilière u. a., 2019, S.4). Als Verlustfunktionen werden MSE, MAE und der Huber-Verlust (siehe Abschnitt 2.8.6 auf Seite 16) betrachtet, da diese in (Lathuilière u. a., 2019, S.7) bei der Betrachtung neuer Probleme empfohlen werden. Des Weiteren wird der Effekt einer Zentrierung der Ein- und Ausgabewerte um 0 untersucht.

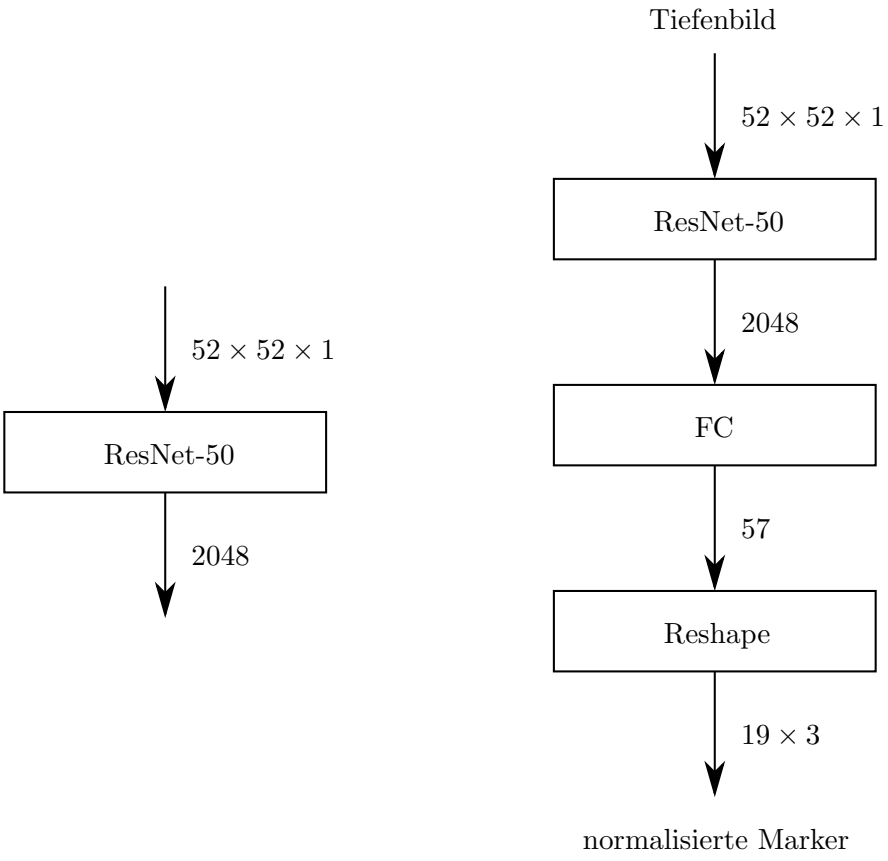

Abbildung 3.15.: Die Darstellung der Netzwerkarchitektur mit den Ein- und Ausgaben sowie ihren Dimensionen. Links ist das ursprüngliche von Keras mit den zuvor beschriebenen Einstellungen erzeugte Netzwerk zu sehen. Rechts ist das ergänzte Netzwerk abgebildet. *FC* ist die vollständig verbundene Regressions-Ebene.

```
1  import tensorflow as tf
2  from tensorflow import keras
3  from tensorflow.keras import layers
4
5  resnet50 = tf.keras.applications.ResNet50(include_top=False, weights=None,
   ↪  pooling='avg', input_shape=(52, 52, 1))
6
7  dense_target = layers.Dense(57, activation='relu',
   ↪  input_shape=resnet50.output_shape)(resnet50.output) # Regressionsebene
8  reshaped = layers.Reshape([19, 3])(dense_target)
9
10 resnet50FC = tf.keras.Model(inputs=resnet50.input, outputs=reshaped)
```

Abbildung 3.16.: Die Anpassung des in TensorFlow implementierten ResNet-50-Netzwerks mit einer Regressions- und Reshape-Ebene zum Training mit Tiefenbildern und normalisierten Markerpositionen analog zu Abschnitt 3.11.2. Die Variable `resnet50FC` enthält das endgültige Netzwerk.

3.12. Aufnahme und Verarbeitung aufgenommener Daten

Zur Aufnahme von echten Daten wurde ein Handschuh basierend auf den Abbildungen aus (Han u. a., 2018a) entwickelt. Dabei wurden die auf den Handrücken gehörenden Marker zunächst auf einem Kartonstück angebracht. Dieses Kartonstück wird dann auf dem Handschuh angebracht. Dies verhindert eine Bewegung der Marker auf dem Handrücken bei einer Daumenbewegung. In den Abbildungen 3.17 und 3.18 ist der verwendete Handschuh zu sehen.

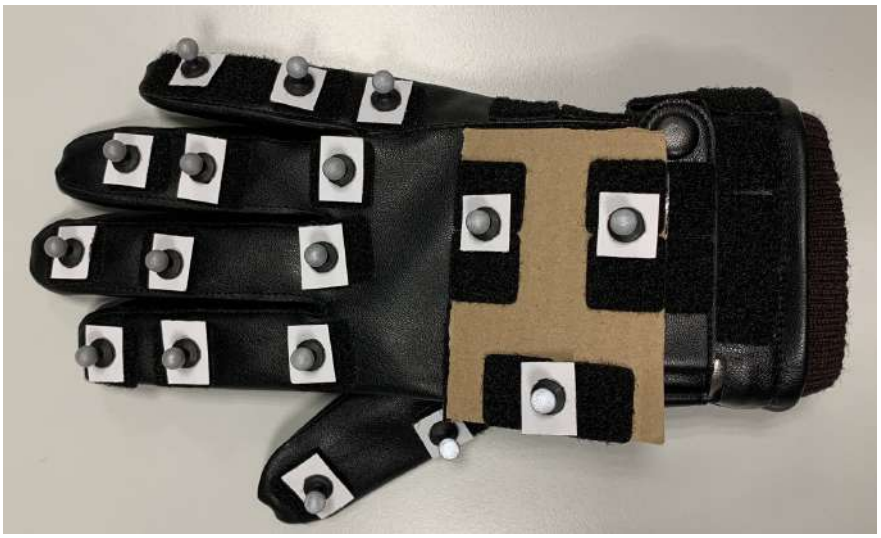

Abbildung 3.17.: Der konstruierte Handschuh (Ansicht von oben).

Abbildung 3.18.: Der konstruierte Handschuh (Ansicht von der Daumenseite).

Die OptiTrack-Kameras waren bei der Aufnahme alle über dem Handschuh an der Decke in einer U-Form angeordnet angebracht, wie es in Abbildung 3.19 zu sehen ist.

Abbildung 3.19.: Das Labor mit dem aufgebauten OptiTrack-System. Die Kameras wurden mit roten Pfeilen markiert.

Zu Beginn der endgültigen Aufnahme wurde die Hand im Raum bewegt. Danach wurde etwa bis zum 600. Frame kaum eine Handbewegung durchgeführt, um aus einem der Frames eine Ruheposition zu ermitteln. Danach wurde die Hand im Raum bewegt sowie Fingerbewegungen für einzelne und mehrere Finger durchgeführt. Bei der Aufnahme wurden Frames, in denen nicht genau 19 Marker sichtbar waren, verworfen und nicht weiter betrachtet. Die Markerpositionen liegen in Zentimetern vor, dies ist nur bei der Rekonstruktion und deren Evaluierung relevant.

Auf dieser endgültige Aufnahme wurde zunächst ein automatisches Labeling mit der Implementierung durchgeführt. Mit Hilfe eines Tools *preview_trc.py*, welches sich im Teilprojekt *python_based_utils* befindet, wurde die Aufnahme manuell gesichtet und die Labels überprüft. Die daraus resultierende korrigierte gelabelte Aufnahme wird dann zur Evaluierung der Netzwerke verwendet.

3.12.1. Generierung zusätzlicher Marker

Basierend auf den aufgenommenen und gelabelten Daten wurde eine synthetische Aufnahme mit zusätzlichen Markern erzeugt. Dabei wurden $0-2$ und $0-4$ Marker zufällig in einem Würfel mit einer Kantenlänge von 25 cm um das Zentrum der Punktwolke erzeugt. Dies ist grob der Raum, den die Hand bei jeder Rotation einnehmen kann. Die zugehörige Implementierung ist im JupyterLab-Notebook *real_data_with_random_additions.ipynb* im Teilprojekt *notebooks* zu finden.

3.13. Netzwerkkommunikation über NatNet und VRPN

Die nachfolgend beschriebenen Komponenten sind im Teilprojekt *hand_tracking_vrpn* und *streaming* zu finden. Die NatNet-Implementierung basiert auf den in NatNet (NaturalPoint, Inc., 2019) mitgeliefertem Handbuch und den mitgelieferten Beispielen. NatNet ist in (Natural Point, Inc., 2018b) dokumentiert. Die in dieser Implementierung verwendete Version ist 3.0. Für die VRPN-Implementierung wurden (VRPN-Entwickler, 2019) sowie (The VR Geeks Association, 2010) und (The VR Geeks Association, 2011) herangezogen.

3.13.1. Empfangen von Markerdaten über NatNet

Das mit NatNet mitgelieferte Beispiel (NaturalPoint, Inc., 2019, NatNetSDK: SampleClient.cpp), auf dem diese Implementierung basiert, verwendet viele globale Variablen. Aus der Dokumentation wird nicht klar, welche Lebenszeiten für Argumente von der NatNet-Programmierschnittstelle benötigt werden. Aus diesem Grund werden nicht konstante Daten, wie Server-Adressen, für die Dauer der Verbindung zwischengespeichert.

Vor dem Empfang von Markerdaten über NatNet muss zunächst nach NatNet-Servern wie OptiTrack Motive im aktuellen Netzwerk gesucht werden. Dabei wird immer der erste gefundene Server

verwendet.

Zum Empfang von Markerdaten muss bei NatNet ein Callback, der ankommende Frames von aktuell sichtbaren Markern verarbeitet, registriert werden. Da nach (Stack Overflow, 2017b) die Verwendung von Exceptions in Callbacks, die von in C entwickelten Bibliotheken aufgerufen werden, nicht sicher ist und die genaue Implementierung von NatNet nicht bekannt ist, werden alle Exceptions in den Callback-Funktionen abgefangen.

3.13.2. Senden und Empfangen von Markerdaten über VRPN

Zum Empfangen und Versenden von Markerdaten über VRPN wurde eine eigene von `vrpn_BaseClass` abgeleitete Klasse `marker_tracker` implementiert.

Analog zu der mitgelieferten Klasse `vrpn_Tracker_Remote` aus (VRPN-Entwickler, 2019) wurde eine von `marker_tracker` erbende Klasse `marker_tracker_remote` eingeführt.

`marker_tracker` bietet dabei eine Methode an, um Frames mit den aktuell sichtbaren Markern in VRPN-Nachrichten über von VRPN bereitgestellte Methoden zu konvertieren. Da verschiedene Systeme unterschiedliche Bitgrößen für C++-Grunddatentypen wie `int` besitzen (cppreference.com, 2019, int type), werden bei der Konvertierung Datentypen mit fester Bitgröße (wie `int32_t`) verwendet. Analog dazu bietet `marker_tracker_remote` eine Methode zum Entpacken von VRPN-Nachrichten zu Frames an.

Zum Versenden von aus einer TRC-Datei importierten Frames wurde eine von `marker_tracker` abgeleitete Klasse `file_marker_tracker` von abgeleitet. Diese versendet in einer Schleife alle Frames der aktuellen Datei. Wird das Ende der Datei erreicht, wird das Versenden wieder am Anfang fortgesetzt. Dabei wird zwischen dem Versand der jeweiligen Frames entsprechend der Datenrate der TRC-Datei pausiert, um eben diese Datenrate beim Versand zu erreichen.

3.13.3. Weiterleitung von Markerdaten aus OptiTrack Motive über VRPN

Bei der NatNet-Bibliothek von OptiTrack werden nur Binärdateien für Windows mitgeliefert (NaturalPoint, Inc., 2019, NatNet SDK). Aus diesem Grund fiel die Entscheidung auf eine Weiterleitung der Markerdaten über ein anderes Protokoll, um die Daten auf anderen Betriebssystemen wie Linux verarbeiten zu können. Dabei bot sich die plattformübergreifende VRPN-Bibliothek an, siehe Abschnitt 2.19.1 auf Seite 25 und (VRPN-Entwickler, 2019). Die Weiterleitung wird dabei von einem Programm übernommen, welches auf dem gleichen System wie OptiTrack Motive ausgeführt werden muss.

Bei der Implementierung der Weiterleitung wird das Empfangen der Daten über NatNet analog zu Abschnitt 3.13.1 umgesetzt. Dabei werden die durch den Callback gelieferten Frames in einem Vektor gesichert. Nach dem Aufbau der NatNet-Verbindung wird die VRPN-Mainloop ausgeführt. Dabei wird geprüft, ob der zuvor genannte Vektor neue Frames enthält. Ist dies der Fall werden die Frames analog zu Abschnitt 3.13.2 versendet. Die Mainloop-Methoden der Basisklasse `vrpn_BaseClass` werden jedoch immer ausgeführt, da es sonst Probleme bei dem Verbindungsaufbau neuer Clients gibt. Der Zugriff auf den Vektor ist mit einem Mutex gesichert, da nicht bekannt ist, aus welchem Thread der NatNet-Callback aufgerufen wird.

3.13.4. Verarbeiten von Daten in der Live-Ansicht

Die Daten werden dabei entweder durch VRPN in einem getrennten Thread, in dem die VRPN-Mainloop läuft, oder durch NatNet empfangen. Dabei wird sowohl in dem Thread als auch durch NatNet ein Callback beim Empfang von neuen Frames aufgerufen. In dieser Callback-Funktion wird dann die Pipeline bestehend aus dem Labeling und der Rekonstruktion der Handkonfiguration durch inverse Kinematik auf den empfangenen Markern durchgeführt. Werden Frames dabei nicht

schnell genug verarbeitet, kann es vorkommen, dass der Callback und somit der Empfang bei VRPN beziehungsweise NatNet blockiert wird.

Die Ergebnisse der Pipeline werden dann in einer durch einen Mutex gesicherten Variable gespeichert. In der Live-Ansicht wird diese Variable in jedem Frame ausgelesen, um die Anzeige zu aktualisieren.

Die Implementierung zu diesem Abschnitt ist in *streaming.h* sowie *streaming.cpp* des Teilprojektes *streaming* und in *stream_gui.h* sowie *stream_gui.cpp* des Teilprojektes *live_view* zu finden.

4. Ergebnisse

4.1. Ablauf

Zu Beginn wurde untersucht, wie sich die Reimplementierung mit dem bestehenden Netzwerk aus (Han u. a., 2018a) auf den mitgelieferten Trainingsdaten verhält. Dabei wurde das Labeling getrennt von der Rekonstruktion betrachtet. Dazu wurde die Pipeline auf allen 168691 Frames der Trainingsdaten ausgeführt. Da auf den Trainingsdaten das Labeling bekannt ist, kann dabei der Anteil der richtig gelabelten Frames bestimmt werden. Die Prüfung aller Frames wird nachfolgend als ein Durchlauf bezeichnet. Die Betrachtung des Anteils wird auch in (Han u. a., 2018a) als Maß für die Güte der Ergebnisse verwendet. Des Weiteren wurde währenddessen die Performance des Labeling-Schritts pro Frame betrachtet.

Analog dazu wurden die bereits gelabelten aufgenommenen Daten untersucht und die Anteile an richtig gelabelten Frames bestimmt. Dabei wurden aufgrund des Zufalls bei der Auswahl der optischen Achse zehn verschiedene Durchläufe über die Daten betrachtet.

Daraufhin wurde der unangepasste Rekonstruktionsschritt manuell und nur auf wenigen Frames der Trainingsdaten betrachtet. Da dieser Probleme aufwies, die in Abschnitt 4.2.2 beschrieben sind, wurde die unangepasste Variante nicht auf den echten Daten betrachtet. Im Rahmen dessen wurde, wie in Abschnitt 3.10.2 auf Seite 53 beschrieben, die Performance des Linear Blend Skinnings zu Optimierungszwecken betrachtet.

Dann wurde der Effekt der Anpassungen am Labeling-Verfahren sowohl auf den Trainingsdaten als auch auf den aufgenommenen Daten untersucht. Bei der Verwendung von mehreren festen optischen Achsen existiert kein Zufall mehr und es wurde nur noch ein Durchlauf betrachtet. Dabei wurde auch die Auswirkung der unterschiedlichen in der Implementierung beschriebenen Netzwerkkonfigurationen und der alternativen ResNet-50-Architektur untersucht. Analog zur Reimplementierung werden dabei die Anteile an richtig gelabelten Frames bestimmt. Da bei neu gelernten Netzwerken das ursprüngliche Verfahren generell schlechte Ergebnisse lieferte, wurde auch nur ein Durchlauf betrachtet.

Dem folgend wurde die Auswirkung der Anpassungen an der Rekonstruktion der Handkonfigurationen untersucht. Dabei wurde der RMS-Fehler mit dem in Abschnitt 3.10.3 auf Seite 53 beschriebenen Schwellenwert für alle Frames der aufgenommen Daten betrachtet.

Die Tools, die dabei verwendet wurden, sind in den Teilprojekten *verify_with_training_data* und *label_ik* zu finden. Die Anpassung der Verfahren sowie die Wahl des Netzwerks müssen dabei im Sourcecode angepasst werden, wenn eine unterschiedliche Konfiguration erwünscht ist.
Die Auswertung erfolgte auf dem folgenden System:

```
Betriebssystem: Ubuntu Linux 18.04.3 LTS
CPU: Intel Xeon E3-1230 v3 (3.3 Ghz)
RAM: 24 GB
GPU: NVIDIA GeForce RTX 2070
```

4.2. Reimplementierung

4.2.1. Labeling der Marker

Beim Labeling wurden wie in Abschnitt 4.1 beschrieben, zunächst die Trainingsdaten und dann die aufgenommenen Daten betrachtet. Die Ergebnisse werden im Folgenden beschrieben und daraus werden dann Schlussfolgerungen gezogen.

Auf den 168691 Frames der Trainingsdaten (siehe Abschnitt 3.11.2 auf Seite 55) konnten mit einer unveränderten Reimplementierung keine guten Ergebnisse erreicht werden. Des Weiteren hat das System je nach Projektionsrichtung Probleme beim Labeling von Trainingsdaten der linken beziehungsweise rechten Hand. Aufeinander folgende Durchläufe über alle Trainingsdaten weisen starke Unterschiede auf. Eine hohe Anzahl von Frames die in dem ersten Durchlauf richtig beziehungsweise falsch gelabelt wurden, sind im zweiten Durchlauf jeweils falsch beziehungsweise richtig, also unterschiedlich, gelabelt worden. Diese Anzahl liegt bei etwa 29000 Frames, also etwa 17.19% aller Trainingsdaten.

Die Anzahl der richtig gelabelten Frames in einem konkreten Durchlauf ist mit etwa 48862, also etwa 28.97%, relativ niedrig und das System erreicht nicht den Bereich von der in (Han u. a., 2018a) angegebenen Ergebnisse von 97.76%, wenngleich diese für Testdaten angegeben sind. Dabei sind nur 708 der 84181 Frames für Trainingsdaten der linken Hand, also 0.84% richtig gelabelt worden. Für die rechte Hand liefert das System mit 48164 von 84510, also 56.98%, bessere Ergebnisse. In Abbildung 4.1 ist der Vergleich zwischen der Reimplementierung und den Ergebnissen aus dem Paper dargestellt.

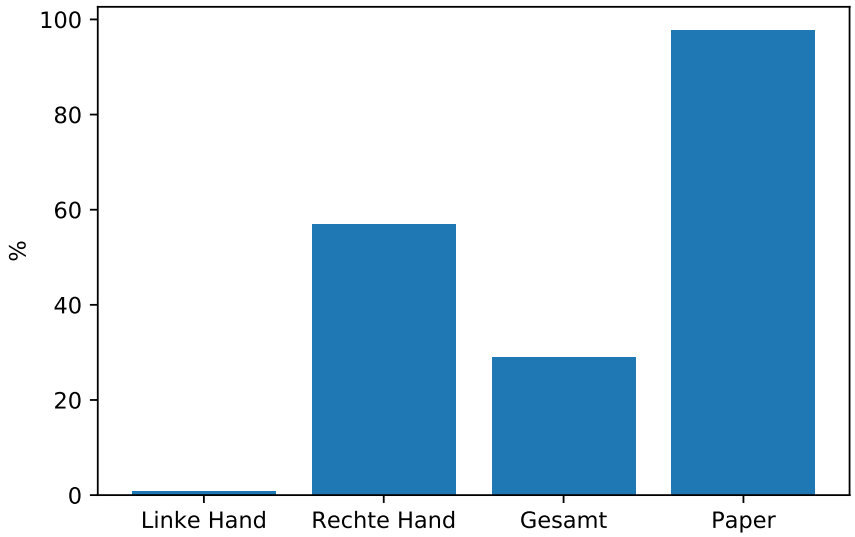

Abbildung 4.1.: Vergleich der Anteile richtig gelabelter Frames der Reimplementierung auf den Trainingsdaten (*Linke Hand*, *Rechte Hand*, *Gesamt*) mit den in (Han u. a., 2018a) angegebenen Ergebnissen auf Testdaten (*Paper*: 97.76%).

Für die 1927 Frames der aufgenommenen Daten für die rechte Hand liefert die Reimplementierung ein besseres Ergebnis. Bei zehn Durchläufen war das schlechteste Ergebnis 1515 (etwa 78.62%), das beste 1550 (etwa 80.44%). Im Durchschnitt wurden 1529.8 (etwa 79.39%) richtig gelabelt. Dies steht im Kontrast zu den in (Han u. a., 2018a) angegebenen 100% für eine Hand bei realen Daten. Dies wird in Abbildung 4.2 graphisch Dargestellt.

Bei der Betrachtung der Trainingsdaten fiel der starke Unterschied zwischen verschiedenen Durchläufen auf. Auch bei echten Daten fällt dieser Unterschied auf, jedoch nicht in einem solchen Ausmaß.

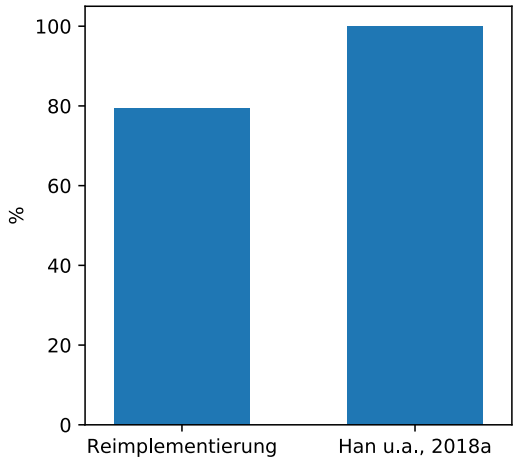

Abbildung 4.2.: Vergleich der Anteile richtig gelabelter Frames zwischen der Reimplementierung auf den aufgenommen Daten mit den in (Han u. a., 2018a) angegebenen Ergebnissen.

Der einzige Unterschied zwischen aufeinander folgenden Durchläufen stellen die jeweils zufällig generierten optischen Achsen bei der Projektion dar. Die Auswahl der besten Projektion basierend auf der Verteilung der Marker auf dem Tiefenbild aus zehn zufälligen Achsen reicht also nicht aus, um zuverlässig gute Achsen für alle Projektionen zu finden.

Ein Beispiel, was gegen die Verwendung von zufälligen Achsen spricht, ist der Fall, dass alle zehn zufällig generierten optischen Achsen nah bei einander liegen beziehungsweise die gleiche Blickrichtung darstellen könnten. Aus diesen kann zwar nach dem Bewertungskriterium eine Projektion gewählt werden. Wenn die grobe Blickrichtung jedoch prinzipiell schlecht ist, kann keins der Tiefenbilder eine gute Eingabe für das neuronale Netzwerk liefern. Dies spricht sowohl gegen eine Verwendung von zufälligen Achsen als auch gegen das aktuelle Auswahlverfahren.

Für einen weiteren Unterschied bei den echten Daten könnte hier die Positionierung der Marker auf dem Handschuh verantwortlich sein, da diese basierend auf Abbildungen aus (Han u. a., 2018a) geschätzt werden musste.

4.2.2. Rekonstruktion des Handmodells

Beim Testen des einstufigen Verfahrens auf Trainings- und echten Aufnahmedaten wurde festgestellt, dass bei zentrierten oder sich in der Nähe des Ursprungs befindenden Marker-Punktwolken ein RMS-Fehler von unter $8mm$ teilweise erreicht werden kann. Daraus folgt, dass die Rekonstruktion der Fingerkonfiguration möglich ist. Sobald sich jedoch die echte Hand vom Ursprung entfernt oder die Marker-Punktwolke nicht im Ursprung zentriert wird, kann das Verfahren in dieser Implementierung die globale Transformation nicht bestimmen. Die resultierenden Parameter sind dann sehr nah an der Ruheposition und der RMS-Fehler steigt entsprechend der Entfernung der echten Hand zum Ursprung.

In (Han u. a., 2018a) wurde keine besondere Behandlung der Parameter zur globalen Transformation angesprochen. Des Weiteren ist die Repräsentation der globalen Rotation durch eulersche Winkel eventuell unterschiedlich zu dem von den Autoren verwendeten Verfahren. Es werden zwar in (Han u. a., 2018a) die Anzahl der Parameter angeführt, jedoch wird kein konkretes Handmodell beschrieben. Darüber hinaus kann die verwendete Implementierung des Levenberg-Marquardt-Solvers unterschiedlich sein. Diese beschriebenen Unterschiede können die Ursache für das Scheitern des ursprünglichen Verfahrens sein.

4.2.3. Performance

Bei den nachfolgenden Messungen der Performance wurde jeweils der erste Frame übersprungen, da dabei noch TensorFlow für die erstmalige Verwendung initialisiert wird.

Die Labeling-Reimplementierung ist langsamer als die in (Han u. a., 2018a) beschriebene. Darin wird für das Labeling 1 ms und für den Rekonstruktionsschritt 2 ms angegeben. Wie in Tabelle 4.1 zu sehen ist, ist diese Reimplementierung durchschnittlich um den Faktor 2.64 langsamer als die in (Han u. a., 2018a). Dies deutet auf mögliche Performance-Probleme der Implementierung hin.

Aufgrund des hohen Maximums bei der Gesamtzeit des Labeling-Schrittes wurde ein Histogramm der Zeiten betrachtet, welches in Abbildung 4.3 zu sehen ist. Daraus lässt sich schließen, dass die Zeit 5 ms üblicherweise nicht übersteigt.

	Durchschnitt	**min**	**max**
Labeling	2.64 ms	2.05 ms	33.74 ms

Tabelle 4.1.: Die durchschnittliche, minimale und maximale Zeit, die das System zum Labeling eines Frames benötigt.

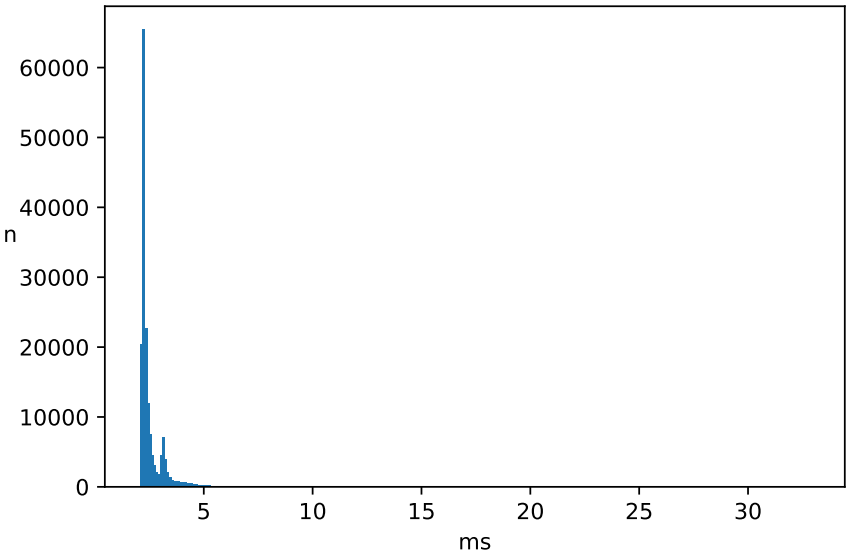

Abbildung 4.3.: Histogramm der Labeling-Performance über alle Trainingsdaten. Dabei werden die Klassen im Intervall $[2, 33]$ mit einer Schrittgröße von 0.1 betrachtet.

Im Hinblick auf die in (Han u. a., 2018a) angegebene Zeit von 2 ms für den Rekonstruktionsschritt, ist die Implementierung schnell genug für 60 Hz und sogar 120 Hz.

Im Folgenden wird kurz auf die Performance von Linear Blend Skinning und die Ergebnisse des Versuchs LBS durch OpenMP zu beschleunigen eingegangen.

Performance von LBS

Die Parallelisierung von Linear Blend Skinning über OpenMP verlangsamte die Rekonstruktion der Handkonfiguration, wie es in Tabelle 4.2 zu sehen ist. Das liegt wahrscheinlich daran, dass jeder `#pragma omp for` Befehl eine neue Gruppe von Threads erzeugt (OpenMP Architecture Review Board, 2018, S.75f). Eine Erzeugung der Threads außerhalb der LBS-Funktion wurde nicht untersucht. Auf Windows war die LBS-Implementierung deutlich langsamer.

Berechnungsart	Zeit
Sequentiell	2899 ns
OpenMP	3202 ns

Tabelle 4.2.: Die Zeit, die beim LBS für die Berechnung der Position aller Marker der Hand benötigt wird, abhängig davon, ob diese Berechnung sequentiell oder parallel mit OpenMP durchgeführt wird.

4.3. Anpassungen am Labeling-Verfahren

4.3.1. Trainingsdaten

Bei dem Verglich verschiedener Auswahlverfahren für die optische Achse liefert die Verwendung der Distanzsumme anstatt des ursprünglichen Verfahrens das beste Ergebnis auf den Trainingsdaten für die rechte Hand. In der Tabelle 4.3 ist jeweils ein Vergleich der Ergebnisse für verschiedene Auswahlverfahren zu finden.

Auswahlverfahren	Beide Hände	Linke Hand	Rechte Hand
(Han u. a., 2018a)	28.97%	0.84%	56.98%
Maximum ohne Multiplikation	29.73%	2.95%	56.41%
Distanzsumme	33.37%	2.37%	64.24%
Überlappungsmaß	29.24%	1.22%	54.97%

Tabelle 4.3.: Vergleich zwischen verschiedenen Auswahlverfahren für die optische Achse auf den Trainingsdaten.

Keines dieser Auswahlverfahren erreichte jedoch eine deutliche Verbesserung für die linke Hand und somit auch nicht für beide Hände. Im späteren Verlauf wird jedoch nur das ursprüngliche Auswahlverfahren betrachtet, um einen direkten Vergleich zur Implementierung aus (Han u. a., 2018a) zu haben.

Die Verwendung von zuvor bestimmten optischen Achsen, die gleichmäßig aus allen Richtungen angeordnet sind (siehe Abschnitte 3.8 auf Seite 44 und 3.9 auf Seite 46), verbessern das Ergebnis auf den Trainingsdaten stark. Die zusätzliche Verwendung eben dieser Achsen mit einer Umkehrung der Tiefe bei der Projektion reduziert den Unterschied zwischen der linken und rechten Hand. Des Weiteren gibt es wie erwartet keine Unterschiede zwischen aufeinander folgenden Durchläufen. In der Tabelle 4.4 ist der Vergleich zwischen der unterschiedlichen Anzahl an optischen Achsen pro Hand zu sehen.

Wenn die Anzahl der optischen Achsen erhöht wird, verbessert sich das Gesamtergebnis und der Unterschied zwischen den Händen verringert sich. Dadurch erhöht sich jedoch auch die Zeit, die zum Labeling der Marker in einem Frame benötigt wird, stark. Die Verwendung von 32 Achsen pro Hand verhindert das Nutzen des Systems mit einer Datenrate von 120 Hz. Auch bei 60 Hz ist die Verwendung, insbesondere unter Berücksichtigung des Rekonstruktionsschritts und vereinzelte Frames, die länger benötigen, schwierig.

Anzahl der optischen Achsen	Beide Hände	Linke Hand	Rechte Hand	Zeit
16	85.53%	83.65%	87.39%	7 ms
32	86.6%[1]	-	-	11.8 ms
64	88%	-	-	-

Tabelle 4.4.: Vergleich zwischen der unterschiedlichen Anzahl an optischen Achsen sowie die für den Labeling-Schritt benötigte Zeit.

Die Verwendung von 32 zufälligen Achsen oder zufälligen Rotationen für jede Hand lieferten dabei mit jeweils 71.4% beziehungsweise 80.6% ein schlechteres Ergebnis und werden nachfolgend nicht mehr berücksichtigt.

4.3.2. Aufgenommene Daten

Aufgrund der Performance-Charakteristik bei der Nutzung einer höheren Zahl an optischen Achsen werden nachfolgend nur die Verwendung von 16 und 32 Achsen betrachtet. Auf den aufgenommenen Daten liefert die Verwendung mehrere optischer Achsen eine deutliche Verbesserung der Ergebnisse. Mit 32 Achsen pro Hand kann das Ergebnis von 100% aus (Han u. a., 2018a) konsistent nahezu erreicht werden. Dabei werden bei 1926 von 1927 Frames die Marker richtig gelabelt. Ein Vergleich der Ergebnisse ist in der Tabelle 4.5 sowie in der Abbildung 4.4 zu finden.

Verfahren	**Anteil von richtig gelabelten Frames**
(Han u. a., 2018a)	100%
Reimplementierung	Im Durchschnitt: 79.39%
16 Achsen für die rechte Hand[2]	98.29% (1894 Frames)
32 Achsen für jede Hand	99.95% (1927 Frames)

Tabelle 4.5.: Vergleich des angepassten Labeling-Verfahrens zur Reimplementierung auf echten Daten. Dabei ist das Ergebnis von (Han u. a., 2018a) eben daraus entnommen worden und bezieht sich nicht auf die in dieser Arbeit aufgenommenen Daten.

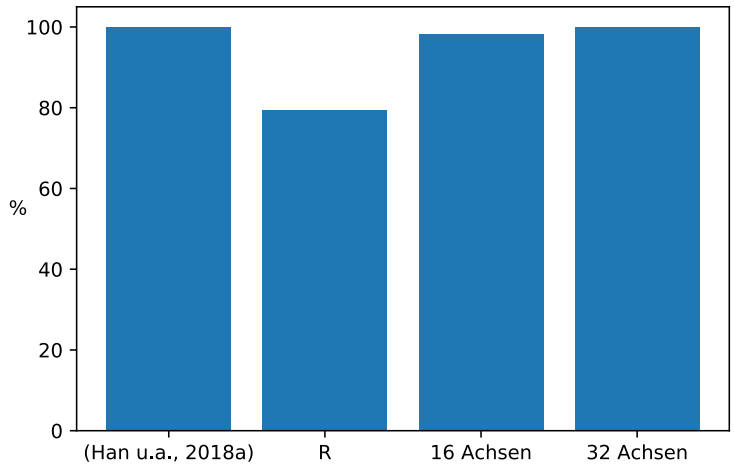

Abbildung 4.4.: Graphische Darstellung des Vergleichs aus Tabelle 4.5. *R* bezeichnet dabei die Reimplementierung.

Im Folgenden wird nur die Verwendung von 32 Achsen für jede Hand betrachtet, da dies das beste Ergebnis lieferte. Diese Variante wird nachfolgende als *fixed32* bezeichnet.

4.4. Verwendung anderer Neuronaler Netzwerke

Mit der in (Han u. a., 2018a) angegebenen Lernrate konnte kein Netzwerk mit der MSE-Verlustfunktion und dem SGD-Optimierungsalgorithmus gelernt werden. Dabei verringerte sich der Verlust nach

[1]Der Unterschied zwischen beiden Händen liegt bei 32 in der ersten und bei 64 in der zweiten Nachkommastelle.
[2]Die Verwendung von Achsen für eine spezifische Hand ist für diese aufgenommenen Daten der rechten Hand problematisch, da so keine Aussage getroffen werden kann, wie gut das System auf beiden Händen funktioniert.

wenigen Epochen nicht mehr.

Wie in Abschnitt 3.11.4 auf Seite 56 und Abschnitt 3.11.5 beschrieben, wurden daraufhin die Lernraten 0.06 und 0.02 betrachtet. Die gelernten Netzwerke konnten dabei die Ergebnisse des in (Han u. a., 2018a) mitgelieferten Netzwerks nicht erreichen. Die Reimplementierung war dabei um etwa einen Faktor von 3 schlechter mit unter 30%. Die Verfahren mit mehreren Achsen erreichten dabei mindestens 90% und bieten somit mit diesen Netzwerken Ergebnisse, die für das Gesamtsystem ausreichend sind. Die Tabelle 4.6 enthält die Ergebnisse mit den beschriebenen Lernraten. Die Lernrate 0.02 liefert dabei ein besseres Ergebnis.

Im Gegensatz zu Abschnitt 4.2.1 auf Seite 66 wird bei dem ursprünglichen Auswahlverfahren mit nur einer optischen Achse nur ein Ergebnis betrachtet, da dieses immer deutlich unter dem Minimum der Ergebnisse des ursprünglichen Netzwerks liegt.

Lernrate	**Reimplementierung**	**fixed32**
0.06	502 (26.05%)	1746 (90.61%)
0.02	550 (28.54%)	1780 (92.37%)

Tabelle 4.6.: Vergleich der Ergebnisse bei verschiedenen Lernraten.

4.4.1. Zentrieren der Ein- und Ausgaben um Null

Das Zentrieren der Ein- und Ausgaben, wie sie in Abschnitt 3.11.2 auf Seite 55 beschrieben ist, verbessert das Ergebnis leicht. Dabei wird jedoch mit der Lernrate 0.06 ein besseres Ergebnis erreicht. In Tabelle 4.7 werden die Ergebnisse zusammengefasst. Interessanterweise ist das Ergebnis mit einer optischen Achse dabei deutlich schlechter. Dies kann jedoch auch am Zufall bei der Auswahl liegen, da nur ein Ergebnis betrachtet wird.

Lernrate	**Reimplementierung**	**fixed32**
0.06	422 (21.9%)	1831 (95.02%)
0.02	640 (33.21%)	1780 (92.37%)

Tabelle 4.7.: Vergleich der Ergebnisse des Zentrierens bei verschiedenen Lernraten.

4.4.2. Effekt von Verlustfunktionen und Optimierungsalgorithmen

Die Verwendung des Optimierungsalgorithmus AdaDelta verbessert ebenfalls das Ergebnis zu reinem SGD. Die Optimierung mit Adam schlug jedoch fehl und das resultierende Netzwerk lieferte bei *fixed32* nur 49 richtig gelabelte Frames.

Die Verlustfunktionen MAE und Huber verbessern das Ergebnis sowohl bei der Verwendung von SGD mit einer Lernrate von 0.6 als auch bei AdaDelta. Da diese aus (Han u. a., 2018a) stammende Lernrate für MAE und Huber bereits gute Ergebnisse lieferte, wurde nicht nach besseren Lernraten gesucht.

Das beste Ergebnis liefert AdaDelta mit der MAE-Verlustfunktion mit 1885 (97.82%) richtig gelabelten Frames. In Tabelle 4.8 sind die Ergebnisse für verschiedene Verlustfunktionen und Optimierungsalgorithmen zusammengefasst.

Für das ursprüngliche Verfahren mit einer optischen Achse liefert keins der Netzwerke mehr als 890 richtig gelabelte Frames bei der Betrachtung eines einzigen Durchlaufs.

Verlustfunktion	SGD 0.6	SGD 0.06	SGD 0.02	AdaDelta
MSE	-	1746 (90.61%)	1780 (92.37%)	1823 (94.60%)
MAE	1867 (96.89%)	-	-	1885 (97.82%)
Huber	1804 (93.62%)	-	-	1831 (95.02%)

Tabelle 4.8.: Vergleich unterschiedlicher Verlustfunktionen und Optimierungsalgorithmen.

4.4.3. ResNet-50

Das ResNet-50 liefert bei Standardparametern bereits sehr gute Ergebnisse unter Betrachtung unterschiedlicher Verlustfunktionen und dem Optimierungsalgorithmus AdaDelta. Mit 1911 (99.17%) richtig gelabelten Frames bei dem Netzwerk mit der Verlustfunktion MAE werden Ergebnisse nah an dem Netzwerk aus (Han u. a., 2018a) erreicht. Zusätzlich wurden die Netzwerke nach 70 anstatt 75 Epochen betrachtet, da sich der Verlust auf den Trainingsdaten wenig zwischen diesen Epochen veränderte. Die Ergebnisse der auf ResNet-50 basierenden Netzwerk ist in Tabelle 4.9 zusammengefasst.

Bei der Verwendung des Adam-Algorithmus zur Optimierung sank der Verlust beim Nutzen der Verlustfunktionen MSE und Huber nach einigen Epochen nicht mehr ab. Mit MAE erreichte Adam zwar einen vergleichsweise kleinen Verlust auf den Trainingsdaten, konnte jedoch verhältnismäßig das Labeling auf weniger Frames richtig durchführen.

Verlustfunktion	AdaDelta Ep.70	AdaDelta Ep.75	Adam Ep.70	Adam Ep.75
MSE	1843 (95.64%)	1850 (96.00%)	-	-
MAE	1911 (99.17%)	1911 (99.17%)	1180 (61.24%)	1382 (71.72%)
Huber	1858 (96.42%)	1867 (96.89%)	-	-

Tabelle 4.9.: Vergleich unterschiedlicher Verlustfunktionen und Optimierungsalgorithmen beim ResNet-50 mit dem *fixed32*-Verfahren. Dabei wurde sowohl die 70. als auch die 75.Epoche betrachtet.

Beim Zentrieren der Ein- und Ausgaben um 0, wie im Abschnitt 3.11.2 beschrieben ist und bereits in Abschnitt 4.4.1 mit der ursprünglichen Netzwerkarchitektur betrachtet wurde, liefert das ResNet für alle in Tabelle 4.9 beschriebenen Kombinationen von Verlustfunktionen und Algorithmen keine Ergebnisse. Beim Lernen bleibt der Verlust nach einigen Epochen hängen und ändert sich nicht bis zur 75.Epoche.

Das Verwenden einer GMP-Ebene anstatt einer GAP-Ebene vor der Regressionsebene führt bei der MSE-Verlustfunktion mit AdaDelta und Adam auch zu einem nicht sinkenden Verlust. Da beide Optimierungsalgorithmen fehlschlugen, wurde die GMP-Variante nicht mehr weiter betrachtet.

Wird das ursprüngliche Verfahren mit nur einer optischen Achse genutzt, sind die Ergebnisse besser, aber immer noch vergleichbar mit denen der ursprünglichen Netzwerkarchitektur. Dabei liefert AdaDelta mit MAE das beste Ergebnis bis auf das Netzwerk aus (Han u. a., 2018a). In Tabelle 4.10 sind die Ergebnisse dazu zusammengefasst.

Verlustfunktion	AdaDelta Ep.70	AdaDelta Ep.75	Adam Ep.70	Adam Ep.75
MSE	452 (23.46%)	575 (29.84%)	-	-
MAE	1399 (72.60%)	1198 (62.17%)	217 (11.26%)	189 (9.81%)
Huber	894 (46.39%)	990 (51.38%)	-	-

Tabelle 4.10.: Vergleich unterschiedlicher Verlustfunktionen und Optimierungsalgorithmen beim ResNet-50 mit dem ursprünglichen Verfahren.

Die Verwendung des ResNet-50 zur Laufzeit ist in der aktuellen Implementierung mit der aktuellen GPU leider nicht möglich, da TensorFlow sowohl beim Lernen als auch bei der erstmaligen Nutzung des gelernten Netzwerks versucht mehr GPU-Speicher als vorhanden zu allozieren[3]. Zwar wird ein Teil noch auf der GPU bei relativ hoher Auslaustung ausgeführt, die Auswertung der 1927 Frames dauert jedoch deutlich länger als bei der *fixed32*-Variante. Auch eine Reduzierung der Batchgröße beim Lernen auf 16 oder 8 konnte dieses Problem nicht beheben.

4.4.4. Verwenden von mehr als 19 Markern

Werden mehr als 19 gleichzeitig sichtbare Marker betrachtet, verschlechtert sich der Anteil an richtig gelabelten Frames auf synthetischen Daten stark. Die Generierung dieser Daten ist in Abschnitt 3.12 auf Seite 61 zu finden. Es wurde nur das beste Verfahren *fixed32* auf dem Netzwerk aus (Han u. a., 2018a) betrachtet. Bei zwei Geistermarkern sinkt der Anteil auf 78.83% und bei vier auf 57.19%.

4.4.5. Durchschnitt mehrerer Labelings

Bei der Verwendung des Durchschnitts mehrerer Labelings anstatt des besten Labelings liefert das System unbrauchbare Ergebnisse. Nur bei 279 (14.48%) von 1927 Frames wurden die Marker richtig zugeordnet. Dieses Verfahren wurde analog zum vorhergehenden Abschnitt nur mit dem Netzwerk aus (Han u. a., 2018a) und 32 optischen Achsen pro Hand betrachtet, da dieses bisher die besten Ergebnisse lieferte.

4.4.6. Performance

Netzwerkart	Verlustfunktion	Algorithmus	Epoche	Iteration	Datum
(Han u. a., 2018a)		SGD	-	400 ms	-
Bestehende Architektur	SGD	MSE	60 s	91.05 ms	353 µs
		MAE	59 s	89.53 ms	351 µs
		Huber	59 s	89.53 ms	351 µs
	AdaDelta	MSE	64 s	97.11 ms	379 µs
		MAE	66 s	100.15 ms	391 µs
		Huber	65 s	98.63 ms	388 µs
ResNet-50	AdaDelta	MSE	111 s	168.44 ms	660 µs
		MAE	115 s	174.51 ms	680 µs
		Huber	112 s	169.95 ms	665 µs
	Adam	MAE	110 s	166.92 ms	650 µs

Tabelle 4.11.: Die benötigte Zeit zum Lernen der jeweiligen Netzwerke mit unterschiedlichen Hyperparametern pro Epoche, Iteration und Trainingsdatum. In (Han u. a., 2018a) wurde für die Performance nur die Zeit pro Iteration angegeben. Die Zeit pro Iteration in der aktuellen Implementierung wurde von TensorFlow nicht direkt geliefert und wurde über die Anzahl der Iterationen des SGD pro Epoche bestimmt, die in TensorFlow abrufbar ist: $\frac{\text{Zeit pro Epoche}}{659}$. Da AdaDelta und Adam nach (Lathuilière u. a., 2019, S.4) Erweiterungen von SGD sind, lässt sich diese Anzahl an Iterationen für diese Varianten weiterverwenden.

[3]Dies könnte mit dem in Abschnitt 3.11.6 auf Seite 58 beschriebene Problem mit NVIDIA-Grafikkarten der RTX-Serie zusammenhängen.

Beim Lernen der Netzwerke wurde festgestellt, dass Netzwerke gleicher Architekturen sich in der Lernperformance ähnlich verhalten. Dabei fällt auf, dass AdaDelta um etwa 5 s pro Epoche langsamer als SGD ist. Auch beim ResNet-50 benötigt AdaDelta minimal länger als Adam. Dieser kleine Unterschied könnte jedoch durch die Computerauslastung während des Lernens erklärt werden.

Der MSE-Verlust ist bei beiden Netzwerkarten mit AdaDelta etwas schneller als der Huber-Verlust. Die MAE-Verlustfunktion ist dabei immer am langsamsten.

Auf (Han u. a., 2018a) basierenden Netzwerke sind beim Lernen schneller als ResNet-50 basierte Netzwerke. Dies liegt daran, dass das ResNet-50 eine deutlich höhere Anzahl an Ebenen und somit Operationen durchführen muss.

Beim Lernen wurden in (Han u. a., 2018a) *zwei* NVIDIA Tesla M40 GPUs verwendet. Dabei ist die in dieser Arbeit verwendete NVIDIA GeForce RTX 2070 nur um einen Faktor von 1.46 schneller (TechPowerUp, 2019b,a). Der Lernvorgang ist jedoch bei dem Netzwerk mit der gleichen Architektur in dieser Mastarbeit um mehr als einen Faktor von 4 schneller. Selbst das deutlich komplexere Netz ist noch mehr als doppelt so schnell lernbar.

Der Hauptunterschied ist die Verwendung von TensorFlow anstatt von LuaTorch, woraus sich schließen lässt, dass ersteres deutlich effizienter implementiert ist. Des Weiteren wurden die Projektionen in dieser Arbeit vor dem Training generiert. Wie es in (Han u. a., 2018a) gemacht wurde, ist jedoch nicht bekannt. Dies könnte auch einen Teil zu den Unterschieden beitragen.

4.4.7. Zusammenfassung der Ergebnisse

Das ursprüngliche in (Han u. a., 2018a) mitgelieferte Netzwerk liefert im Vergleich zu allen in dieser Arbeit neu gelernten Netzwerken das beste Ergebnis. Dies gilt sowohl für das unveränderte Labeling-Verfahren als auch für die angepassten Verfahren.

Das *fixed32*-Verfahren lieferte bei der Netzwerkarchitektur aus (Han u. a., 2018a) und dem Optimierungsalgorithmus AdaDelta mit der MAE-Verlustfunktion mit 97.82% das beste Ergebnis. Bei der auf ResNet-50 basierenden Netzwerkarchitektur war die Verwendung von AdaDelta mit der MAE-Verlustfunktion mit 99.17% am erfolgreichsten.

Die Ergebnisse des ursprünglichen Verfahrens sind mit den neu gelernten Netzwerken deutlich schlechter. Eine Ausnahme ist dabei das mit MAE als Verlustfunktion und dem AdaDelta-Optimierungsalgorithmus trainierte ResNet-50 nach der 70.Epoche mit 1399 (72.6%) richtig gelabelten Frames.

Das Lernen von Netzwerken bei dieser Implementierung mit TensorFlow war deutlich schneller als das Netzwerk aus (Han u. a., 2018a), obwohl dabei ein System aus zwei GPUs verwendet wurde.

4.5. Anpassungen an der Rekonstruktion des Handmodells

Im Gegensatz zum einstufigen Verfahren, dessen Ergebnisse in Abschnitt 4.2.2 auf Seite 67 zusammengefasst sind, kann mit dem zweistufigen Verfahren die globale Position und Rotation der Hand aus den gelabelten Markern bestimmt werden. Beim Testen mit der Ruheposition fiel jedoch auf, dass die Rekonstruktion des Daumens einen Effekt auf alle anderen Fingerkonfigurationen besitzt. Aus diesem Grund wurde die Rekonstruktion der Finger direkt getrennt und das einfache zweistufige Verfahren nicht weiter betrachtet.

Problematisch ist jedoch die Rekonstruktion der Fingerkonfigurationen beim komplexen Bewegungen. In Abbildung 4.5 sind die RMS-Fehler für jedes Frame über die Aufnahmezeit abgebildet. Bei einfachen Bewegungen und der Ruheposition können Frames mit einem RMS von unter 8 mm rekonstruiert werden. Komplexere Bewegungen führen jedoch dazu, dass die Rekonstruktion scheitert (siehe Abschnitt 3.12 auf Seite 61 für die Beschreibung der Bewegungen). Bei 768 (39.85%) der 1927 Frames lag der RMS über dem gewünschten Schwellenwert.

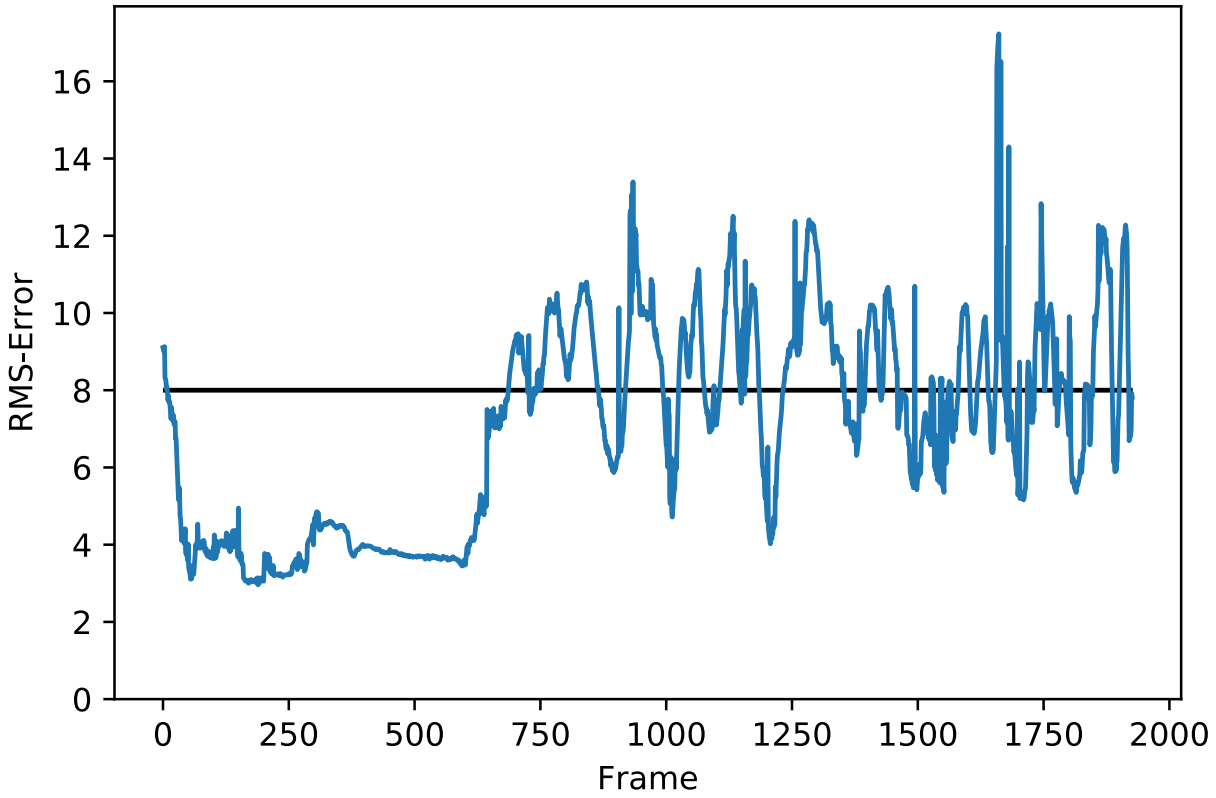

Abbildung 4.5.: Der RMS-Fehler für jedes Frame der Daten über die Zeit der Aufnahme. Der gewünschte Schwellenwert von 8 mm ist mit der schwarzen Linie markiert.

Durch die in Abschnitt 3.10.8 auf Seite 55 beschriebene Optimierung der Markerpositionen des Handmodells konnten deutlich bessere Ergebnisse erzielt werden. In Abbildung 4.6 sind die dazugehörigen RMS-Werte zu finden. Dabei liegen nur noch 35 (1.82%) Frames über dem Schwellenwert. Jedoch ist deutlich in der Live-Ansicht zu erkennen, dass das System immer noch Schwierigkeiten mit komplexen Handbewegungen hat. Das beste Ergebnis lieferte die folgende Translation der Markerpositionen (siehe Abschnitt 3.10.8):

$$\begin{bmatrix} 0 \\ 1.3 \\ -0.8 \end{bmatrix} \tag{4.1}$$

4.5.1. Performance

Der Rekonstruktionsschritt benötigt betrachtet auf den aufgenommen Daten pro Frame im Durchschnitt 3.01 ms. Die Zeiten variieren dabei zwischen 0.89 ms und 19.28 ms. Analog zu Abschnitt 4.2.3 auf Seite 68 wurde hierbei das Histogramm der Zeiten betrachtet, welches in Abbildung 4.7 zu finden ist. Es wurden dabei Klassen im Intervall $[0, 20]$ mit einer Schrittgröße von 0.1 betrachtet. Darin ist zu erkennen, dass bei vielen Frames bis 7.5 ms für den Rekonstruktionsschritt notwendig sind.

Des Weiteren wurde die Performance aller Frames über Zeit betrachtet. Dies ist in Abbildung 4.8 zu finden. Dabei fällt auf, dass für komplexe, von der Ruheposition abweichende Handkonfigurationen mehr Zeit benötigt wird.

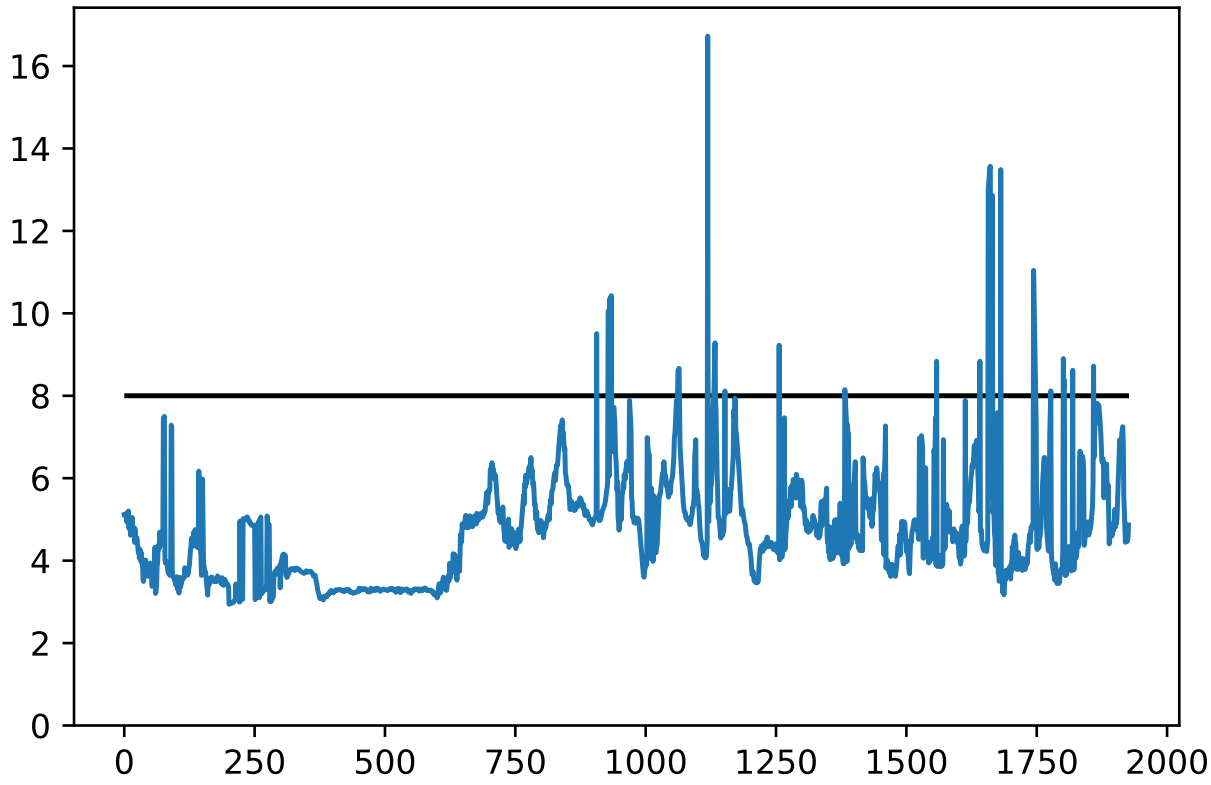

Abbildung 4.6.: Der RMS-Fehler für jedes Frame der Daten über die Zeit der Aufnahme für die optimierten Markerpositionen. Der gewünschte Schwellenwert von 8 mm ist mit der schwarzen Linie markiert.

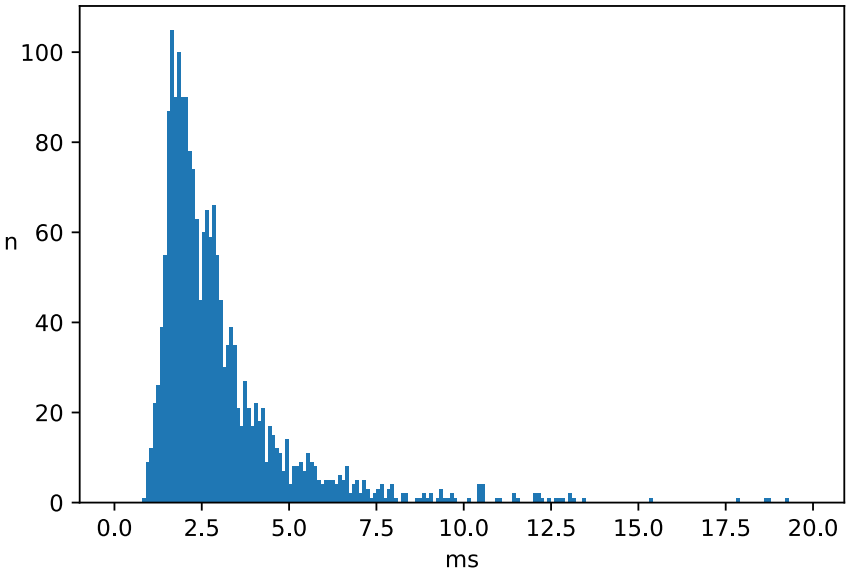

Abbildung 4.7.: Histogramm der Rekonstruktions-Performance über alle aufgenommenen Daten. Dabei werden die Klassen im Intervall $[0, 20]$ mit einer Schrittgröße von 0.1 betrachtet.

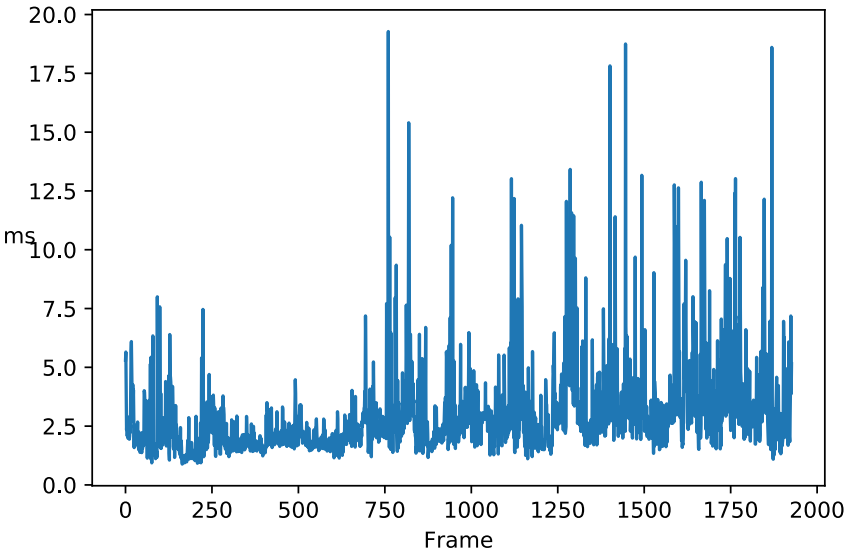

Abbildung 4.8.: Performance des Rekonstruktionsschrittes betrachtet auf allen aufgenommene Frames über Zeit.

5. Fazit

Im Rahmen dieser Masterarbeit wurde das in (Han u. a., 2018a) beschriebene Verfahren zum Labeling von Markern und zur Rekonstruktion der Handkonfiguration daraus implementiert. Zusätzlich zum Paper wurden von den Autoren auch die Trainingsdaten und das beste Netzwerk veröffentlicht, die auch im Rahmen dieser Arbeit verwendet wurden. Bei der Reimplementierung wurde auf die darin beschriebene automatische Anpassung an Handgrößen verzichtet. Des Weiteren wurden keine eigenen neuen synthetischen Trainings- oder Testdaten generiert.

Zur Aufnahme von echten Daten wurde zunächst ein Handschuh erstellt, der auf dem in (Han u. a., 2018a) vorgestellten Handschuhen basiert. Mithilfe dieses Handschuhs konnten dann im Laborraum der Arbeitsgruppe am OptiTrack-System eine Sequenz von Frames aufgenommen werden. Um eine Bewegung der Marker auf dem Handrücken bei einer Daumenbewegung zu verhindern, wurde an dem Handrücken zunächst ein Kartonstück angebracht und die Marker darauf befestigt. Bei der Aufnahme tauchten aufgrund der nicht an den Nutzungszweck angepassten Kamerapositionierung viele Verdeckungen der Marker auf. Diese Frames mit einer Markeranzahl ungleich 19 wurden bei der Betrachtung der Ergebnisse ignoriert.

Bei der Implementierung traten mehrere Probleme an verschiedenen Stellen des in (Han u. a., 2018a) beschriebene Verfahrens auf und die Ergebnisse konnten nicht reproduziert werden. Der Anteil von Frames mit richtig gelabelten Markern lag auf echten Daten etwa 20% unter dem Anteil, der von (Han u. a., 2018a) erreicht wurde. Des Weiteren wurde beim Testen mit den Trainingsdaten festgestellt, dass das System die Marker einer linken Hand kaum zuordnen kann. Bei der Rekonstruktion der Handkonfiguration über inverse Kinematik konnten die globale Position und Rotation nicht bestimmt werden.

Um den Labeling-Schritt zu verbessern, wurden zunächst verschiedene Auswahlverfahren für die optische Achse bei der Projektion betrachtet. Diese lieferten jedoch keine nennenswerten Verbesserungen. Der nächste Schritt war die gleichzeitige Verwendung von 32 optischer Achsen pro Hand und eine darauf folgende Auswahl des besten Labelings basierend auf den paarweisen Differenzen zwischen den Markerpositionen der Eingabe und der Ausgabes des Neuronalen Netzwerks. Dieses nachfolgend als *fixed32* bezeichnete Verfahren führte zu einem deutlich höheren Anteil von 99.95% an richtig gelabelten Frames und erreichte nahezu die in (Han u. a., 2018a) beschriebenen 100%. Die Verwendung des Durchschnitts gleichzeitiger Labelings anstatt des besten Labelings lieferte bei 32 optischen Achsen pro Hand unbrauchbare Ergebnisse und wurde nicht weiter betrachtet.

Darüber hinaus wurden Netzwerke basierend auf der Architektur aus (Han u. a., 2018a) mit dem dort spezifizierten Optimierungsalgorithmus gelernt. Mit der Annahme, dass dabei die MSE-Verlustfunktion verwendet wurde, konnte mit der angegebenen Lernrate von 0.6 kein Netzwerk gelernt werden, da der Verlust auf den Trainingsdaten nach wenigen Epochen nicht mehr sank. Aus diesem Grund wurde das Verhalten anderer ähnlicher Lernraten betrachtet und 0.06 sowie 0.02 als gute Alternativen ermittelt. Letztere lieferte mit dem *fixed32*-Verfahren dabei mit einem Anteil von 92.37% an richtig gelabelten Frames zwar ein gutes Ergebnis, erreichte jedoch nicht die 100% aus (Han u. a., 2018a). Das ursprüngliche Verfahren mit nur einer optischen Achse war dabei jedoch mit unter 30% deutlich schlechter. Des Weiteren wurde eine Zentrierung der Ein- und Ausgaben des Netzwerks um 0 untersucht, dies erreichte mit 95.02% und der Lernrate 0.06 ein etwas besseres Ergebnis.

Daraufhin wurde der Effekt unterschiedlicher Verlustfunktionen und Optimierungsalgorithmen auf dieser Netzwerkarchitektur untersucht. Den höchsten Anteil an richtig gelabelten Frames von

97.82% lieferte dabei der Algorithmus AdaDelta mit der MAE-Verlustfunktion. Auch hier konnte das Ergebnis aus (Han u. a., 2018a) nicht erreicht werden.

Das Verwenden einer anderen auf ResNet-50 basierenden Netzwerkarchitektur lieferte vergleichbare oder bessere Ergebnisse. Der höchste Anteil von 99.17% wurde wieder mit AdaDelta und der MAE-Verlustfunktion erreicht. Dies ist das Ergebnis, was am nächsten an dem aus (Han u. a., 2018a) lag. Aufgrund der Performance-Probleme können die Netzwerke dieser Architektur aktuell nicht zur Laufzeit verwendet werden.

Eine der möglichen Ursachen für die schlechtere Performance auf den echten Daten könnte die Positionierung der Marker auf dem Handschuh sein. Diese ist in (Han u. a., 2018a) nicht explizit beschrieben und musste basierend auf Abbildungen geschätzt werden.

Keines der neuen Netzwerke konnte die Ergebnisse des in (Han u. a., 2018a) mitgelieferten Netzwerks erreichen. Aus den Ergebnissen der verschiedenen Netzwerke ist jedoch zu erkennen, dass die Verwendung von 32 festen optischen Achsen für jede Hand deutliche zuverlässigere Ergebnisse unabhängig vom verwendeten Netzwerk liefert. Die eventuell etwas andere Positionierung der Marker verhindert dabei im Gegensatz zum Verfahren mit nur einer optische Achse das Erreichen von Ergebnissen, die in der Nähe der 100% aus (Han u. a., 2018a) liegen.

Beim Lernen von Netzwerken fiel auf, dass die benötigte Zeit pro Iteration deutlich unter der in (Han u. a., 2018a) angegebenen Zeit liegt. Als Ursachen liegen dabei einen effizientere Implementierung in TensorFlow im Vergleich zu LuaTorch und die Generierung aller Trainingsdaten vor dem Lernvorgang vor.

Das Problem des *fixed32*-Verfahrens ist jedoch, dass es in der aktuellen Implementierung kaum echtzeitfähig ist. Pro Frame werden durchschnittlich etwa 11.8 ms benötigt. Zusammen mit der Zeit von häufig bis zu 7.5 ms, die dann der Rekonstruktionsschritt erfordert, werden die 60 Hz oder gar die 120 Hz aus (Han u. a., 2018a) nicht erreicht.

Bei der Betrachtung von mehr als 19 sichtbaren Markern in einem Frame in synthetischen auf der Aufnahme basierenden Daten wurde mit *fixed32* ein geringerer Anteil richtig gelabelt als durch (Han u. a., 2018a) bei Testdaten und echten Daten. Die Testdaten sind dabei mit zufälligen Geistermarkern erweitert worden. Die echten Daten hatten sichtbare Objekten mit zusätzlichen Markern, wie Stifte oder Controller. Die Ergebnisse auf den echten Daten des *fixed32*-basierenden Verfahrens sind jedoch nicht gut vergleichbar, da bei den echten Daten von (Han u. a., 2018a) die zusätzlichen Marker sich außerhalb der Hand befinden und so die Handmarker weniger stören.

Darüber hinaus wurde für diese Masterarbeit eine plattformunabhängige Echtzeitansicht für das rekonstruierte virtuelle Handmodell sowie die sichtbaren Marker, die das zuvor betrachtete Verfahren für das Labeling und die Rekonstruktion der Handkonfiguration nutzt, entwickelt. Diese Ansicht kann sowohl Aufnahmen aus Dateien wiedergeben als auch Daten über das Netzwerk mit NatNet auf Windows oder mit VRPN plattformunabhängig empfangen.

6. Ausblick

Auf der in dieser Arbeit entstandenen Implementierung kann an verschiedenen Punkten zur Erweiterung angesetzt werden. Dies umfasst sowohl das Labeling, die Rekonstruktion als auch die Darstellung der virtuellen Hand zur Echtzeit.

Ein Hauptpunkt ist bei allen Komponenten der Implementierung eine Optimierung der Performance, damit Daten zur Echtzeit verarbeitet werden können. Unter dem Punkt der Performance wäre auch eine genauere Betrachtung anderer Anzahlen von optischen Achsen im Vergleich zum *fixed32*-Verfahren interessant. Da voneinander unabhängige optische Achsen betrachtet werden, bietet sich hier eine relativ einfache Parallelisierung an.

Beim Lernen der Netzwerke können noch weitere Hyperparameter untersucht werden. Bei der neu für die MSE-Verlustfunktion bestimmten Lernrate könnten die Epochengrenzen für die Reduzierung dieser Lernrate angepasst werden. Es existieren noch andere Verlustfunktionen, wie zum Beispiel der L_2-Verlust (Lathuilière u. a., 2019, S.7), die beim Lernen in Frage kommen können. Bei ResNet-50 wurde eine kleinere Anzahl an Epochen nur bedingt betrachtet, hier wäre eine Aufschlüsselung der Ergebnisse für jede Epoche interessant. Da ResNet-50 laut (Lathuilière u. a., 2019, S.4) weniger gegenüber der Batchgröße anfällig ist, wäre unter anderem aus Gründen der GPU-Speicherbelastung eine Untersuchung kleinerer Batchgrößen ein weiterer Ansatzpunkt. Im Gegensatz zur in dieser Arbeit betriebenen manuellen Optimierung der Hyperparameter, könnten nach (Goodfellow u. a., 2016, S.420ff.) andere Techniken, wie eine zufällige Suche, für die Auswahl der Parameter in Frage kommen.

Eine andere mögliche Erweiterung ist bei der Betrachtung von zeitlich aufeinander folgenden Frames die Wiederverwendung der durch OptiTrack gegebenen Labels, um das Labeling mit dem neuronalen Netz dann nur bei Änderungen in den OptiTrack-Labels durchzuführen. Analog zu (Han u. a., 2018a) könnte dann bei verlorenen Markern nach diesen in der Nähe gesucht werden. Dies wäre aus der Sicht der Performance eine starke Verbesserung. Des Weiteren könnten bei dem Verlust der richtigen Zuordnung nur die nicht zugeordneten Marker betrachtet werden. Dies würde jedoch eine Anpassung am Matching erfordern.

Bei der Rekonstruktion der Handkonfiguration gibt es ebenfalls mehrere Ansatzpunkte. Das Linear Blend Skinning oder die Rekonstruktion der jeweiligen Finger könnten parallelisiert werden. Um die Ergebnisse deutlich zu verbessern, ist jedoch die Entwicklung eines besser angepassten Handmodells notwendig. Die Wiederverwendung der Handkonfiguration vorhergehender Frames als Startwert für die Rekonstruktion nach (Han u. a., 2018a) ist zwar implementiert, ist jedoch ausgeschaltet und wurde nicht untersucht. Dies wäre ein weiterer zu untersuchender Punkt.

Zur Bestimmung der Rotation und Position zwischen zwei Punktwolken, existiert das als *Iterative Closest Point* bezeichnete und in (Besl u. McKay, 1992) beschriebene Verfahren. Dieses kann laut den Autoren Rauschen gegenüber nicht sehr empfindlich und könnte statt des bestehenden Verfahrens für den ersten Schritt der zweistufigen Rekonstruktion verwendet werden.

Zwar wurden für Teile des Labelings Unit-Tests entwickelt, diese müssten jedoch auf das gesamte System erweitert werden, insbesondere wenn die bestehende Implementierung weiterentwickelt werden soll. Um das Labeling und die Rekonstruktion aus anderen Programmen, wie der Unreal Engine, nutzen zu können, kann eine C-Programmierschnittstelle für die aktuelle Implementierung entwickelt werden.

A. Literaturverzeichnis

[Abadi u. a. 2015] Abadi, Martín ; Agarwal, Ashish ; Barham, Paul ; Brevdo, Eugene ; Chen, Zhifeng ; Citro, Craig ; Corrado, Greg S. ; Davis, Andy ; Dean, Jeffrey ; Ghemawat, Matthieu D. ; Goodfellow, Ian ; Harp, Andrew ; Irving, Geoffrey ; Isard, Michael ; Jia, Yangqing ; Jozefowicz, Rafal ; Kaiser, Lukasz ; Kudlur, Manjunath ; Levenberg, Josh ; Mané, Dandelion ; Monga, Rajat ; Moore, Sherry ; Murray, Derek ; Olah, Chris ; Schuster, Mike ; Shlens, Jonathon ; Steiner, Benoit ; Sutskever, Ilya ; Talwar, Kunal ; Tucker, Paul ; Vanhoucke, Vincent ; Vasudevan, Vijay ; Viégas, Fernanda ; Vinyals, Oriol ; Warden, Pete ; Wattenberg, Martin ; Wicke, Martin ; Yu, Yuan ; Zheng, Xiaoqiang: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. `https://www.tensorflow.org/`. Version: 2015. – Software available from tensorflow.org

[Ahuja u. a. 2014] Ahuja, Ravindra K. ; Magnanti, Thomas L. ; Orlin, James B.: *Network Flows. Theory, Algorithms and Applications*. Edinburgh Gate, Harlow, Essex CM20 2JE, England : Pearson Education Limited, 2014. – ISBN 9781292042701. – Person New International Edition

[Ahuja u. a. 1994] Ahuja, Ravindra K. ; Magnanti, Thomas L. ; Orlin, James B. ; Reddy, MR: Applications of network optimization. (1994). `https://dspace.mit.edu/bitstream/handle/1721.1/5097/OR-300-94-26970351.pdf?sequence=1`. – Zuletzt abgerufen am 29. Dezember 2019.

[Albrecht u. a. 2003] Albrecht, Irene ; Haber, Jörg ; Seidel, Hans-Peter: Construction and animation of anatomically based human hand models. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* Eurographics Association, 2003, S. 98–109

[Alexanderson u. a. 2017] Alexanderson, Simon ; O'Sullivan, Carol ; Beskow, Jonas: Real-time labeling of non-rigid motion capture marker sets. In: *Computers & Graphics* 69 (2017), S. 59–67

[Arias 2018] Arias, Pablo: It's Time To Do CMake Right. (2018). `https://pabloariasal.github.io/2018/02/19/its-time-to-do-cmake-right/`. – Zuletzt abgerufen am 10. Januar 2020.

[Aristidou u. Lasenby 2010] Aristidou, Andreas ; Lasenby, Joan: Motion capture with constrained inverse kinematics for real-time hand tracking. In: *2010 4th International Symposium on Communications, Control and Signal Processing (ISCCSP)* IEEE, 2010, S. 1–5

[Arun u. a. 1987] Arun, K S. ; Huang, Thomas S. ; Blostein, Steven D.: Least-squares fitting of two 3-D point sets. In: *IEEE Transactions on pattern analysis and machine intelligence* (1987), Nr. 5, S. 698–700

[Behnel 2019] Behnel, Stefan: Lupa. Lua in Python. (2019). `https://github.com/scoder/lupa`. – Zuletzt abgerufen am 27. Dezember 2019.

[Besl u. McKay 1992] Besl, Paul J. ; McKay, Neil D.: Method for registration of 3-D shapes. In: *Sensor fusion IV: control paradigms and data structures* Bd. 1611 International Society for Optics and Photonics, 1992, S. 586–606

[Boost 2018] Boost: Boost 1.69.0 Library Documentation. (2018). https://www.boost.org/doc/libs/1_69_0/. – Version 1.69. Zuletzt abgerufen am 5. Januar 2020.

[Chintala 2017] Chintala, Soumith: cudnn.torch. (2017). https://github.com/soumith/cudnn.torch/tree/R7. – Branch R7. Zuletzt abgerufen am 27. Dezember 2019.

[Chollet u. a. 2019] Chollet, François ; Google ; Microsoft ; Keras-Entwickler: Keras. (2019). http://keras.io/. – Zuletzt abgerufen am 28. Dezember 2019.

[Collobert u. a. 2019a] Collobert, Ronan ; Kavukcuoglu, Koray ; Farabet, Clément ; Bottou, Leon ; Melvin, Iain ; Weston, Jason ; Bengio, Samy ; Marietho z, Johnny ; Chintala, Soumith ; Leonard, Nicholas ; Tompson, Jonathan ; Zagoruyko, Sergey ; Massa, Francisco ; Dundar, Aysegul ; Jin, Jonghoon ; Canziani, Alfredo ; Desmaison, Alban ; Deltheil, Cedric ; Perkins, Hugh: Torch. Scientific computing for LuaJIT. (2019). http://torch.ch/. – Zuletzt abgerufen am 27. Dezember 2019. Achtung: Weblink ohne HTTPS, da die Homepage fehlerhaft konfiguriert ist.

[Collobert u. a. 2019b] Collobert, Ronan ; Kavukcuoglu, Koray ; Farabet, Clément ; Bottou, Leon ; Melvin, Iain ; Weston, Jason ; Bengio, Samy ; Marietho z, Johnny ; Chintala, Soumith ; Leonard, Nicholas ; Tompson, Jonathan ; Zagoruyko, Sergey ; Massa, Francisco ; Dundar, Aysegul ; Jin, Jonghoon ; Canziani, Alfredo ; Desmaison, Alban ; Deltheil, Cedric ; Perkins, Hugh: torch7. (2019). https://github.com/torch/torch7. – Zuletzt abgerufen am 27. Dezember 2019.

[Collobert u. a. 2017] Collobert, Ronan ; Kavukcuoglu, Koray ; Farabet, Clément ; Bottou, Leon ; Melvin, Iain ; Weston, Jason ; Bengio, Samy ; Marietho z, Johnny ; Chintala, Soumith ; Leonard, Nicholas ; Tompson, Jonathan ; Zagoruyko, Sergey ; Massa, Francisco ; Dundar, Aysegul ; Jin, Jonghoon ; Canziani, Alfredo ; Desmaison, Alban ; Deltheil, Cedric ; Perkins, Hugh: cutorch. (2017). https://github.com/torch/cutorch. – Zuletzt abgerufen am 27. Dezember 2019.

[cppreference.com 2019] cppreference.com: C++ language. (2019). https://en.cppreference.com/w/cpp/language. – Zuletzt abgerufen am 5. Januar 2020.

[CyberGlove Systems Inc. 2019] CyberGlove Systems Inc.: CyberGlove II. (2019). http://www.cyberglovesystems.com/cyberglove-ii. – Zuletzt abgerufen am 11. Januar 2020

[Deserno 2004] Deserno, Markus: *How to generate equidistributed points on the surface of a sphere*. Max-Planck-Institut für Polymerforschung, Ackermannweg 10, 55128 Mainz, Germany, 2004 https://www.cmu.edu/biolphys/deserno/pdf/sphere_equi.pdf. – Zuletzt abgerufen am 08. Dezember 2019.

[Ecma International 2017] Ecma International: *The JSON Data Interchange Syntax*. Rue du Rhône 114, CH-1204 Geneva, Schweiz, 2017 https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. – Zuletzt abgerufen am 19. Dezember 2019.

[Egerváry Research Group on Combinatorial Optimization (EGRES) 2014] Egerváry Research Group on Combinatorial Optimization (EGRES): *LEMON 1.3.1 Documentation*. Eötvös Loránd University, Budapest, Hungary, 2014 http://lemon.cs.elte.hu/pub/doc/1.3.1/index.html. – Zuletzt abgerufen am 30. Dezember 2019.

[Eriksson 2019] Eriksson, Emil: RapidCheck. (2019). https://github.com/emil-e/rapidcheck. – Zuletzt abgerufen am 10. Januar 2020.

[Free Software Foundation 2019] Free Software Foundation: The C Preprocessor: Standard Predefined Macros. (2019). https://gcc.gnu.org/. – Zuletzt abgerufen am 11. Januar 2020.

[Free Software Foundation 2020] Free Software Foundation: The C Preprocessor: Standard Predefined Macros. (2020). https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html. – Zuletzt abgerufen am 5. Januar 2020.

[Gebhardt 2016] Gebhardt, Nikolaus: Irrlicht 3D Engine. Irrlicht Engine 1.8 API documentation. (2016). http://irrlicht.sourceforge.net/docu/. – Zuletzt abgerufen am 5. Januar 2020.

[Glorot u. Bengio 2010] Glorot, Xavier ; Bengio, Yoshua: Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, S. 249–256

[Goldberg 1991] Goldberg, David: What every computer scientist should know about floating-point arithmetic. In: *ACM Computing Surveys (CSUR)* 23 (1991), Nr. 1, S. 5–48

[Goodfellow u. a. 2016] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. Cambridge, Massachusetts, USA : MIT Press, 2016 (Adaptive Computation and Machine Learning). – ISBN 9780262035613

[Google 2019a] Google: Benchmark. (2019). https://github.com/google/benchmark/tree/4abdfbb802d1b514703223f5f852ce4a507d32d2. – Zuletzt abgerufen am 5. Juli 2019.

[Google 2019b] Google: Google Test. (2019). https://github.com/google/googletest. – Zuletzt abgerufen am 18. April 2019.

[Han u. a. 2018a] Han, Shangchen ; Liu, Beibei ; Wang, Robert ; Ye, Yuting ; Twigg, Christopher D. ; Kin, Kenrick: *Online Optical Marker-based Hand Tracking with Deep Labels.* 2018 https://research.fb.com/publications/online-optical-marker-based-hand-tracking-with-deep-labels/. – Zuletzt abgerufen am 5. April 2019.

[Han u. a. 2018b] Han, Shangchen ; Liu, Beibei ; Wang, Robert ; Ye, Yuting ; Twigg, Christopher D. ; Kin, Kenrick: *Online Optical Marker-based Hand Tracking with Deep Labels. Mocap_SIG18_Data.* 2018 https://github.com/Beibei88/Mocap_SIG18_Data. – Zuletzt abgerufen am 27. Dezember 2019.

[He u. a. 2015] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In: *The IEEE International Conference on Computer Vision (ICCV)*, 2015

[He u. a. 2016] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, S. 770–778

[Hoffbeck u. Landgrebe 1996] Hoffbeck, Joseph P. ; Landgrebe, David A.: Covariance matrix estimation and classification with limited training data. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18 (1996), Nr. 7, S. 763–767

[Hughes u. a. 2014] Hughes, John F. ; Dam, Andries van ; McGuire, Morgan ; Sklar, David ; Foley, Jim ; Feiner, Steve ; Akeley, Kurt: *Computer Graphics. Principles and Practice.* Pearson Education, Inc., 2014. – ISBN 978–0–321–39952–6. – Third Edition.

[Hunter u. a. 2019] Hunter, John ; Dale, Darren ; Firing, Eric ; Droettboom, Michael ; The Matplotlib development team: Overview – Matplotlib 3.1.1 documentation. (2019). https://matplotlib.org/contents.html. – Zuletzt abgerufen am 5. Januar 2020.

[Ierusalimschy u. a. 2012] Ierusalimschy, Roberto ; Figueiredo, Luiz H. ; Celes, Waldemar: *Lua 5.1 Reference Manual*. 2012 https://www.lua.org/manual/5.1/manual.html. – Zuletzt abgerufen am 27. Dezember 2019.

[Ierusalimschy u. a. 2019] Ierusalimschy, Roberto ; Figueiredo, Luiz H. ; Celes, Waldemar: *The Programming Language Lua*. 2019 https://www.lua.org/. – Zuletzt abgerufen am 27. Dezember 2019.

[Internet Engineering Task Force (IETF) 2014] Internet Engineering Task Force (IETF): The JavaScript Object Notation (JSON) Data Interchange Format. (2014). https://tools.ietf.org/html/rfc7159. – Zuletzt abgerufen am 10. Januar 2020.

[Jacob u. Guennebaud 2018a] Jacob, Benoît ; Guennebaud, Gaël: Eigen. (2018). http://eigen.tuxfamily.org/dox/. – Achtung: Keine HTTPS-Verbindung zur Webseite des Projekts möglich. Es wurden nur die Gründer des Projekts angegeben, weitere Autoren sind auf der Homepage unter http://eigen.tuxfamily.org/dox/ zu finden. Dokumentation wurde seit der Veröffentlichung der verwendeten Version 3.3.7 im Jahr 2018 aktualisiert. Zuletzt abgerufen am 4. Januar 2020.

[Jacob u. Guennebaud 2018b] Jacob, Benoît ; Guennebaud, Gaël: Eigen-unsupported. (2018). https://eigen.tuxfamily.org/dox/unsupported/index.html. – Siehe Hinweise bei (Jacob u. Guennebaud, 2018a). Zuletzt abgerufen am 4. Januar 2020.

[Kavan 2014] Kavan, Ladislav: *Part I: Direct Skinning Methods and Deformation Primitives. SIGGRAPH Course 2014 – Skinning: Real-time Shape Deformation*. University of Pennsylvania, 2014 https://skinning.org/direct-methods.pdf. – Zuletzt abgerufen am 21. Dezember 2019.

[Kitware, Inc. and Contributors 2019] Kitware, Inc. and Contributors: CMake Reference Documentations. (2019). https://cmake.org/cmake/help/v3.16/. – Zuletzt abgerufen am 10. Januar 2020.

[Kitware, Inc. and Contributors 2020] Kitware, Inc. and Contributors: CMake. (2020). https://gitlab.kitware.com/cmake/cmake. – Zuletzt abgerufen am 10. Januar 2020.

[Krizhevsky u. a. 2012] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: *ImageNet Classification with Deep Convolutional Neural Networks*. 2012 http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf. – Zuletzt abgerufen am 12. Dezember 2019.

[Lathuilière u. a. 2019] Lathuilière, Stéphane ; Mesejo, Pablo ; Alameda-Pineda, Xavier ; Horaud, Radu: A Comprehensive Analysis of Deep Regression. In: *IEEE transactions on pattern analysis and machine intelligence* (2019). https://arxiv.org/pdf/1803.08450.pdf. – Zuletzt abgerufen am 19. Dezember 2019. Entwurfsversion vom 13. Februar 2019.

[Lee u. Kunii 1995] Lee, Jintae ; Kunii, Tosiyasu L.: Model-based analysis of hand posture. In: *IEEE Computer Graphics and applications* 15 (1995), Nr. 5, S. 77–86

[Levoy u. Whitted 1985] Levoy, Marc ; Whitted, Turner: *The Use of Points as a Display Primitive. Technical Report 85-022, Computer Science Department*. University of North Carolina at Chapel Hill, 1985 https://graphics.stanford.edu/papers/points/. – Zuletzt abgerufen am 26. Dezember 2019.

[Li u. a. 2016] Li, Chao ; Yang, Yi ; Feng, Min ; Chakradhar, Srimat ; Zhou, Huiyang: Optimizing memory efficiency for deep convolutional neural networks on GPUs. In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* IEEE, 2016, S. 633–644

[Lin u. a. 2000] Lin, John ; Wu, Ying ; Huang, Thomas S.: Modeling the constraints of human hand motion. In: *Proceedings workshop on human motion* IEEE, 2000, S. 121–126. – https://www.researchgate.net/profile/Kunihiro_Chihara/publication/221470259_3-D_Modeling_of_Human_Hand_with_Motion_Constraints/links/00463526737f0eb341000000.pdf – Zuletzt abgerufen am 2. Januar 2020.

[Matplotlib Contributors 2020] Matplotlib Contributors: jupyter-matplotlib. (2020). https://github.com/matplotlib/jupyter-matplotlib. – Zuletzt abgerufen am 10. Januar 2020.

[Meyers 2015] Meyers, Scott: *Effective Modern C++*. 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA : O'Reilly Media, Inc., 2015. – ISBN 9781491903995

[Microsoft 2019] Microsoft: C++ in Visual Studio — Microsoft Docs. (2019). https://docs.microsoft.com/en-us/cpp/overview/visual-cpp-in-visual-studio?view=vs-2019. – Zuletzt abgerufen am 5. Januar 2020.

[Mueller u. a. 2019] Mueller, Franziska ; Davis, Micah ; Bernard, Florian ; Sotnychenko, Oleksandr ; Verschoor, Mickeal ; Otaduy, Miguel A. ; Casas, Dan ; Theobalt, Christian: Real-time pose and shape reconstruction of two interacting hands with a single depth camera. In: *ACM Transactions on Graphics (TOG)* 38 (2019), Nr. 4, S. 1–13

[Natural Point, Inc. 2017a] Natural Point, Inc.: *Data Export: TRC – NaturalPoint Product Documentation Ver 2.0*. 2017 https://v20.wiki.optitrack.com/index.php?title=Data_Export:_TRC. – Zuletzt abgerufen am 15. Dezember 2019.

[Natural Point, Inc. 2017b] Natural Point, Inc.: *Data Types – NaturalPoint Product Documentation Ver 2.0*. 2017 https://v20.wiki.optitrack.com/index.php?title=Data_Types. – Zuletzt abgerufen am 11. Januar 2020.

[Natural Point, Inc. 2018a] Natural Point, Inc.: *Markers – NaturalPoint Product Documentation Ver 2.0*. 2018 https://v20.wiki.optitrack.com/index.php?title=Markers. – Zuletzt abgerufen am 11. Januar 2020.

[Natural Point, Inc. 2018b] Natural Point, Inc.: NatNet SDK. (2018). https://v20.wiki.optitrack.com/index.php?title=NatNet_SDK_3.0. – Zuletzt abgerufen am 6. Januar 2020.

[NaturalPoint, Inc. 2019] NaturalPoint, Inc.: *OptiTrack - Motion Capture Systems*. 2019 https://optitrack.com/. – Zuletzt abgerufen am 15. Dezember 2019.

[NVIDIA 2018] NVIDIA: CUDA Toolkit Documentation v10.0.130. (2018). https://docs.nvidia.com/cuda/archive/10.0/. – Zuletzt abgerufen am 10. Januar 2020.

[NVIDIA 2019] NVIDIA: cuDNN Developer Guide. (2019). https://docs.nvidia.com/deeplearning/sdk/cudnn-archived/cudnn_765/cudnn-developer-guide/index.html. – Zuletzt abgerufen am 10. Januar 2020.

[OpenMP Architecture Review Board 2018] OpenMP Architecture Review Board: OpenMP. Application Programming Interface. (2018). https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. – Zuletzt abgerufen am 4. Januar 2020.

[Pall 2018] Pall, Mike: LuaJIT. The LuaJIT Project. (2018). https://luajit.org/. – Zuletzt abgerufen am 27. Dezember 2019.

[Pintaric u. Kaufmann 2007] Pintaric, Thomas ; Kaufmann, Hannes: Affordable infrared-optical pose-tracking for virtual and augmented reality. In: *Proceedings of Trends and Issues in Tracking for Virtual Environments Workshop, IEEE VR*, 2007, S. 44–51

[Poskanzer 2016] Poskanzer, Jef: *PGM Format Specification*. 2016 http://netpbm.sourceforge.net/doc/pgm.html. – Teil von Netpbm. Zuletzt abgerufen am 8. Januar 2020.

[Project Jupyter 2018] Project Jupyter: JupyterLab Documentation. (2018). https://jupyterlab.readthedocs.io/en/stable/. – Zuletzt abgerufen am 5. Januar 2020.

[Python Software Foundation 2019a] Python Software Foundation: *json - JSON encoder and decoder*. 2019 https://docs.python.org/3/library/json.html. – Zuletzt abgerufen am 19. Dezember 2019.

[Python Software Foundation 2019b] Python Software Foundation: Python 3.7.6 documentation. (2019). https://www.python.org/. – Versionen 3.6 - 3.7. Zuletzt abgerufen am 28. Dezember 2019.

[Rijpkema u. Girard 1991] Rijpkema, Hans ; Girard, Michael: Computer animation of knowledge-based human grasping. In: *ACM Siggraph Computer Graphics* Bd. 25 ACM, 1991, S. 339–348

[Sanso u. Thalmann 1994] Sanso, Ramon M. ; Thalmann, Daniel: A hand control and automatic grasping system for synthetic actors. (1994). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.8382&rep=rep1&type=pdf. – Zuletzt abgerufen am 11. Januar 2020.

[Schröder u. a. 2014] Schröder, Matthias ; Maycock, Jonathan ; Ritter, Helge ; Botsch, Mario: Real-time hand tracking using synergistic inverse kinematics. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)* IEEE, 2014, S. 5447–5454

[Scratchapixel 2016] Scratchapixel: Placing a Camera: the LookAt Function. (2016). https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/lookat-function. – Zuletzt abgerufen am 11. Januar 2020.

[Siciliano u. Khatib 2008] Siciliano, Bruno (Hrsg.) ; Khatib, Oussama (Hrsg.): *Springer Handbook of Robotics*. Berlin Heidelberg : Springer, 2008. – ISBN 9783540303015

[Simonyan u. Zisserman 2015] Simonyan, Karen ; Zisserman, Andrew: Very Deep Convolutional Networks For Large-Scale Image Recognition. (2015). https://arxiv.org/abs/1409.1556v6. – Zuletzt abgerufen am 13. Dezember 2019.

[Skiena 2012] Skiena, Steven S.: *The Algorithm Design Manual*. Department of Computer Science, State University of New York at Stony Brook, New York, USA : Springer-Verlag London Limited, 2012. – ISBN 978–1–84800–069–8. – Second Edition.

[Spruyt 2014] Spruyt, Vincent: *A geometric interpretation of the covariance matrix*. 2014 (Computer vision for dummies). https://www.visiondummy.com/2014/04/geometric-interpretation-covariance-matrix/. – Zuletzt abgerufen am 24. Dezember 2019.

[Stack Overflow 2011] Stack Overflow: M_PI works with math.h but not with cmath in Visual Studio. (2011). https://stackoverflow.com/questions/6563810/. – Zuletzt abgerufen am 9. Januar 2020

[Stack Overflow 2015] Stack Overflow: Box plot with min, max, average and standard deviation. (2015). https://stackoverflow.com/questions/33328774/box-plot-with-min-max-average-and-standard-deviation. – Zuletzt abgerufen am 10. Januar 2020

[Stack Overflow 2017a] Stack Overflow: Setting weights in Keras model. (2017). https://stackoverflow.com/a/47210750. – Zuletzt abgerufen am 9. Januar 2020

[Stack Overflow 2017b] Stack Overflow: Throwing an exception in C++ in a C callback, possibly crossing over dynamic library boundary... is it safe? (2017). https://stackoverflow.com/questions/10956062/throwing-an-exception-in-c-in-a-c-callback-possibly-crossing-over-dynamic-lib. – Zuletzt abgerufen am 9. Januar 2020

[Stack Overflow 2019] Stack Overflow: What is the proper way to use 'pkg-config' from 'cmake'. (2019). https://stackoverflow.com/questions/29191855/what-is-the-proper-way-to-use-pkg-config-from-cmake. – Zuletzt abgerufen am 10. Januar 2020

[TechPowerUp 2019a] TechPowerUp: NVIDIA GeForce RTX 2070 Specs — TechPowerUp GPU Database. (2019). https://www.techpowerup.com/gpu-specs/geforce-rtx-2070.c3252. – Zuletzt abgerufen am 8. Januar 2020.

[TechPowerUp 2019b] TechPowerUp: NVIDIA Tesla M40 Specs — TechPowerUp GPU Database. (2019). https://www.techpowerup.com/gpu-specs/tesla-m40.c2771. – Zuletzt abgerufen am 8. Januar 2020.

[The Go Authors 2019] The Go Authors: *Package json*. 2019 https://golang.org/pkg/encoding/json/. – Supported by Google, zuletzt abgerufen am 19. Dezember 2019.

[The SciPy community 2019] The SciPy community: NumPy v1.16 Manual. (2019). https://numpy.org/doc/1.16/. – Version 1.16.3. Zuletzt abgerufen am 28. Dezember 2019.

[The VR Geeks Association 2010] The VR Geeks Association: Tutorial - Use VRPN. (2010). http://www.vrgeeks.org/vrpn/tutorial---use-vrpn. – Achtung: Kein HTTPS bei diesem Link. Zuletzt abgerufen am 6. Januar 2020.

[The VR Geeks Association 2011] The VR Geeks Association: Tutorial - VRPN Server. (2011). http://www.vrgeeks.org/vrpn/tutorial---vrpn-server. – Achtung: Kein HTTPS bei diesem Link. Zuletzt abgerufen am 6. Januar 2020.

[VRPN-Entwickler 2019] VRPN-Entwickler: Virtual Reality Peripheral Network. (2019). https://github.com/vrpn/vrpn/wiki. – Eine vollständige Liste der Authoren ist unter https://github.com/vrpn/vrpn/wiki/Authors zu finden. Zuletzt abgerufen am 5. Januar 2020.

[Westover 1990] Westover, Lee: Footprint evaluation for volume rendering. In: *ACM Siggraph Computer Graphics* 24 (1990), Nr. 4, 367–376. https://cs.swan.ac.uk/~csbob/teaching/sciVisGraz/vorlesung/VolVisSplatting/westover-1990--splatting.pdf. – Zuletzt abgerufen am 26. Dezember 2019.

[Wikipedia 2019a] Wikipedia: Minimum-cost flow problem. (2019). https://en.wikipedia.org/wiki/Minimum-cost_flow_problem. – Zuletzt abgerufen am 31. Dezember 2019

[Wikipedia 2019b] Wikipedia: PLY (file format). (2019). https://en.wikipedia.org/wiki/PLY_(file_format). – Zuletzt abgerufen am 8. Januar 2020

[Wikipedia 2019c] Wikipedia: Quadratisches Mittel. (2019). https://de.wikipedia.org/wiki/Quadratisches_Mittel. – Zuletzt abgerufen am 9. Januar 2020.

[Wikipedia 2019d] Wikipedia: Retroreflector. (2019). https://en.wikipedia.org/wiki/Retroreflector. – Zuletzt abgerufen am 11. Januar 2020

[Wikipedia 2020] Wikipedia: Netpbm format. (2020). https://en.wikipedia.org/wiki/Netpbm_format. – Zuletzt abgerufen am 8. Januar 2020

[Zhang u. a. 2019] Zhang, Hao ; Bo, Zi-Hao ; Yong, Jun-Hai ; Xu, Feng: InteractionFusion: real-time reconstruction of hand poses and deformable objects in hand-object interactions. In: *ACM Transactions on Graphics (TOG)* 38 (2019), Nr. 4, S. 1–11

B. Details der Implementierung

B.1. Verzeichnisstruktur

Die Tabelle B.1 beschreibt die Verzeichnisstruktur im Wurzelverzeichnis der Implementierung.

Eintrag	Beschreibung
cmake	Zusätzliche CMake-Skripte, die beim Buildvorgang notwendig sind.
components	Die Komponenten der Implementierung, die Unterverzeichnisse werden in der Arbeit als Teilprojekte bezeichnet.
debug-output	Ein leere Verzeichnis in welches Debug-Informationen abgelegt werden.
paper-training-data	In diesem Ordner werden die Trainingsdaten von (Han u. a., 2018a) erwartet. Diese befinden sich in dem Unterordner *training_data* von (Han u. a., 2018b).
recorded-test-data	Enthält verschiedene Aufnahmen. *complex_labeled_dir* enthält dabei die gelabelte Aufnahme, die bei der Auswertung verwendet wurde. *complex_labeled_with_random_additions_dir* enthält die obige Aufnahme mit bis zu zwei zufälligen Markern in jedem Frame.
trained-model	Enthält das ins TensorFlow-Format konvertierte Modell von (Han u. a., 2018a) (*tf-keras-model-2019-05-20.h5*) sowie eine Auswahl gelernter Netzwerke.
CMakeLists.txt	Die CMake-Build-Beschreibung.
prepare_hand_tracking.py	Das Skript zum Vorbereiten der Python-Umgebung und der Abhängigkeiten.
generated-training-data	Ein Teil der vorarbeiteten Trainingsdaten für das Lernen von Netzwerken. Der zweite Teil befindet sich auf der 2. DVD.

Tabelle B.1.: Ein Ausschnitt der Einträge im Wurzelverzeichnis der Implementierung.

B.2. Verwendete Paketskripte für die Netzwerk-Abhängigkeiten

In Abschnitt 3.6.1 auf Seite 37 wurde darauf eingegangen, welche Abhängigkeit für LuaTorch und das Netzwerk aus (Han u. a., 2018a) benötigt werden. In der Tabelle B.2 sind die zugehörigen Paketskripte vermerkt, die für die Installation unter ArchLinux verwendet werden.

Name des Pakets	**Webadresse**[1]
torch7-git	`https://aur.archlinux.org/packages/torch7-git`
torch7-cudnn-r7-git	`https://aur.archlinux.org/packages/torch7-cudnn-r7-git`
torch7-cutorch-git	`https://aur.archlinux.org/packages/torch7-cutorch-git`
torch7-cwrap-git	`https://aur.archlinux.org/packages/torch7-cwrap-git`
torch7-nn-git	`https://aur.archlinux.org/packages/torch7-nn-git`
torch7-paths-git	`https://aur.archlinux.org/packages/torch7-paths-git`

Tabelle B.2.: Eine Liste von Paketskripten für die Abhängigkeiten, die für das in (Han u. a., 2018a) mitgelieferte Netz benötigt werden.

[1] Abgerufen am 28. Dezember 2019.