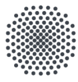


Entwicklung und Validierung einer automatisierten Fahrfunktion mit einem Fahrroboter

Development and Validation of an Automated Maneuvering Function Using a Driving Robot

STUDIENARBEIT

von

cand. fmt. Johanna Hable
Matr.-Nr.: 3617364

Betreuer:	Philipp Klein, M. Sc. (DLR e.V.)
Prüfer:	Prof. Dr.-Ing. A. Wagner (IFS) Prof. Dr.-Ing. T. Siefkes (DLR e.V.)

vorgelegt an der Universität Stuttgart
Institut für Fahrzeugtechnik Stuttgart
Lehrstuhl Kraftfahrwesen

Stuttgart, 28.02.2023

Eidesstattliche Erklärung

Hiermit versichere ich, Johanna Hable, dass ich die vorliegende Arbeit, bzw. die darin mit meinem Namen gekennzeichneten Anteile, selbständig verfasst und bei der Erstellung der Arbeit die einschlägigen Bestimmungen, insbesondere zum Urheberrechtsschutz fremder Beiträge, eingehalten und nur die angegebenen Quellen und Hilfsmittel benutzt habe.

Soweit meine Arbeit fremde Beiträge (z. B. Bilder, Zeichnungen, Textpassagen) enthält, erkläre ich, dass ich diese Beiträge als solche gekennzeichnet (z. B. Zitat, Quellenangabe) habe und dass ich eventuell erforderliche Zustimmungen der Urheber zur Nutzung dieser Beiträge in meiner Arbeit eingeholt habe.

Für den Fall der Verletzung Rechter Dritter durch meine Arbeit erkläre ich mich bereit, der Universität Stuttgart einen daraus entstehenden Schaden zu ersetzen bzw. die Universität Stuttgart von eventuellen Ansprüchen Dritter freizustellen.

Die Arbeit ist weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen. Ferner ist sie weder vollständig noch in Teilen bereits veröffentlicht worden. Das elektronische Exemplar stimmt mit den anderen Exemplaren überein.

Stuttgart, 28. Februar 2023

Johanna Hable

Erklärung zum Nutzungsrecht

Hiermit übertrage ich, Johanna Hable, der Universität Stuttgart das Eigentum an einem von mir der Bibliothek des Instituts für Fahrzeugtechnik Stuttgart (IFS) kostenlos zur Verfügung gestellten Exemplars meiner

Studienarbeit

mit dem Titel

Entwicklung und Validierung einer automatisierten Fahrfunktion mit einem Fahrroboter

und räume dem Institut für Fahrzeugtechnik Stuttgart (IFS) der Universität Stuttgart, an dieser Arbeit und den im Rahmen dieser Arbeit entstandenen Arbeitsergebnissen ein kostenloses, zeitlich und räumlich unbeschränktes und vom IFS übertragbares Nutzungsrecht für Zwecke der Forschung, der Lehre, des Studiums und der Nutzung der Arbeit für Zwecke der Institutsbibliothek ein. Das IFS ist insbesondere berechtigt, das ihr von mir eingeräumte Nutzungsrecht ganz oder teilweise auf das Forschungsinstitut für Kraftfahrwesen und Fahrzeugmotoren Stuttgart (FKFS), mit welchem das IFS kooperiert und das die Fertigung dieser Arbeit unterstützte, zu übertragen.

Mir ist bekannt, dass die Erfassung meiner Arbeit im Online-Katalog der Bibliothek eine dauerhafte, weltweite Sichtbarkeit der bibliografischen Daten der Arbeit (Titel, Autor, Erscheinungsjahr etc.) bedeutet.

Stuttgart, 28. Februar 2023

Johanna Hable

Danksagung

Diese wissenschaftliche Abhandlung entstand im Rahmen des Masterstudiengangs “Fahrzeugtechnik (M.Sc.)” an der Universität Stuttgart innerhalb des Zeitraums von Oktober 2022 bis März 2023. Aufbauend auf dem Bachelorabschluss im klassischen Maschinenbau an der Technischen Hochschule Deggendorf richte ich mich durch die Wahl der Spezialisierungen *Kfz-Technik* und *Automatisiertes und vernetztes Fahren* neu aus. Am Institut für Fahrzeugkonzepte des DLR e.V. in Stuttgart erhielt ich die Möglichkeit, mich mit dem spannenden Entwicklungs- und Forschungsfeld “Autonomes Fahren” nicht nur in der Theorie auseinanderzusetzen, sondern auch selber aktiv einen Beitrag in Form dieser Studienarbeit zu leisten.

An erster Stelle möchte ich einen besonderen Dank meinem Betreuer Philipp Klein aussprechen, welcher mir ermöglichte, wertvolle Erfahrungen in der Softwareentwicklung für autonome Fahrzeuge, der Computer Vision und dem Maschinellen Lernen zu sammeln. Er unterstützte mich bei fachlichen Belangen immer tatkräftig, aber ließ mir dennoch ausreichend Freiraum, um meine eigenen Ideen zu verfolgen. Diesen großen Spielraum für die Entwicklung schätzte ich sehr, da ich meine Kreativität uneingeschränkt ausleben durfte.
Ich bedanke mich außerdem bei dem Abteilungsleiter Herrn Dr.-Ing. Stephan Schmid, welcher mir neben der Anfertigung der Studienarbeit auch eine ergänzende Tätigkeit als studentische Hilfskraft in seiner Fachabteilung eröffnete. Herrn Prof. Dr.-Ing. Tjark Siefkes danke ich für seine Beteiligung als äußerst freundlichen Ansprechpartner und Vermittler.
Meiner Wertschätzung gegenüber meinen Kollegen am DLR möchte ich hiermit ebenfalls Ausdruck verleihen. Explizit adressieren möchte ich Mascha Katharina Brost, Samuel Hasselwander, Christian Ulrich und Simone Ehrenberger. Die Aufgaben, die ich für euch bearbeiten durfte, haben mir Einblicke in viele weitere Forschungstätigkeiten am DLR erlaubt und meiner beruflichen Weiterentwicklung als Ingenieurin verholfen. Außerdem hat mir die Zusammenarbeit mit euch sehr viel Spaß bereitet. Diese Momente werde ich gerne in positiver Erinnerung behalten.

Zuletzt spreche ich ein herzliches Dankeschön an meinen befreundeten Ingenieur Julian Panko aus. Er hat mich auch in schwierigen Zeiten stets unterstützt und durch seinen Rückhalt erheblich zu dem Erfolg dieser Entwicklungsarbeit beigetragen.

// Kurzzusammenfassung

Die vorliegende Arbeit erläutert den Entwicklungsprozess einer prototypischen, modularen Softwarefunktion zur automatisierten Lösung eines hochpräzisen Rangiermanövers mit Bezug zu dem U-Shift-Fahrzeugkonzept des DLR Instituts für Fahrzeugkonzepte in Stuttgart. Durch das Zusammenspiel der Funktionsbausteine Umfeldüberwachung, Steuerung, Lokalisierung und Trajektorienplanung wird ein miniaturisierter Fahrroboter dazu befähigt, einen festgelegten Zielbereich unter Einhaltung gewisser Lagetoleranzen zu befahren. Verfolgt wird hierbei konsequent ein reiner Computer Vision-Ansatz. Als Datenquelle dient ausschließlich eine frontal angebrachte, monokulare und kostengünstige Weitwinkelkamera. Ein besonderer Fokus liegt auf der Erschließung des Zielbereichs in Form einer Landmark Detektion mithilfe von Deep Learning-Methoden. Für das Training des Modells liegt ein speziell für diesen Anwendungsfall erstellter Datensatz mit 13244 gelabelten Bildern vor. Das durch Anwendung des Transfer Learnings entwickelte Convolutional Neural Network (CNN) mit ResNet50-Backbone erzielt eine Genauigkeit von bis zu 97 %. Diese Ergebnisse stellen schließlich die Basis für die visuelle Odometrie zur Lokalisierung der Kamera im Raum dar. Über einen universellen PID-Regler erfolgt die Steuerung der ebenen Fahrzeugbewegung entlang einer parabolischen Solltrajektorie.
Als Resultat wird eine valide Fahrfunktion vorgestellt, welche in einer Büroumgebung Positioniergenauigkeiten vergleichbar zu aktuellen autonomen Parkfunktionen aufweist und eine ausreichende Robustheit gegenüber möglichen Störobjekten besitzt.

Abstract

This paper illustrates the development process of a prototypical, modular software function for the automated execution of a high-precision driving maneuver with reference to the U-Shift concept of the DLR institute for vehicle concepts in Stuttgart. A miniaturized driving robot is enabled to travel on a defined target area while maintaining certain positional tolerances. Involved function modules are perception, control, localization and trajectory planning. During development a computer vision approach is consistently pursued. The relevant data is generated by only a frontally mounted, monocular and inexpensive wide-angle camera. A special focus is placed on the identification of the target area in the form of a landmark detection using deep learning methods. For the training of the model, a dataset containing 13.244 labeled images is available that was created specifically for this application. By applying transfer learning the development of a Convolutional Neural Network (CNN) with ResNet50 backbone results in up to 97 % model accuracy. These outputs are fundamental for visual odometry to localize the camera in the real world. A universal PID-controller is used to control the planar vehicle motion along a parabolic reference trajectory. As result, a valid driving function is presented, which provides positioning accuracies in an office environment comparable to current autonomous parking functions and shows sufficient robustness against possible disturbing objects.

Inhaltsverzeichnis

Abkürzungsverzeichnis II

Abbildungsverzeichnis IV

Tabellenverzeichnis VI

1 Einführung 1
- 1.1 Motivation 1
- 1.2 Zielsetzung 3
- 1.3 Aufbau der Arbeit 6

2 Grundlagen 7
- 2.1 Subsysteme für Autonomes Fahren 7
- 2.2 Computer Vision 9
- 2.3 Maschinelles Lernen 16
- 2.4 Miniaturisierter DIY-Fahrroboter “JetBot” 22

3 Stand der Technik 27
- 3.1 Exteroperceptive Sensorik 27
- 3.2 Prinzipverwandte automatisierte Fahrfunktionen 29
  - 3.2.1 Spurhalteassistenz und Lane Following 29
  - 3.2.2 Autonomes Parken 32

4 Entwicklung der Funktionsbausteine 35
- 4.1 Perception und Objekterkennung 35
  - 4.1.1 Eignung klassischer Computer Vision-Algorithmen 35
  - 4.1.2 Landmark Detektion mit Deep Learning 40
- 4.2 Steuerung 54
- 4.3 Lokalisierung 59
  - 4.3.1 Kontexterfassung aus Bildpunktepositionen 59
  - 4.3.2 Visuelle Odometrie 61
- 4.4 Trajektorienplanung 66

5 Integration der Softwarekomponenten und Funktionsvalidierung 69
- 5.1 Version 1 69
- 5.2 Version 2 70
- 5.3 Validierung und Integrationstests 76

6 Übertragbarkeit auf das U-Shift-Fahrzeugkonzept 81

7 Resümee und Ausblick 84

8 Quellenverzeichnis 86

9 Anhang 92
- 9.1 Pythoncode zu Version 2 der Fahrfunktion 92
- 9.2 Randbedingungen der Integrationstests 106
- 9.3 Ergebnisse der Robustheitsuntersuchung 107

Abkürzungsverzeichnis

API	Application Programming Interface (engl.) *Programmierschnittstelle*
AVM	Around View Monitoring (engl.)
BEV	Birds-eye view (engl.) *Vogelperspektive*
BGR	Blau-Grün-Rot
CHW	Channel-Height-Width (engl.)
CPU	Central Processing Unit (engl.) *Prozessor*
CV	Computer Vision
DL	Deep Learning (engl.)
DRAM	Dynamic random-access memory (engl.)
FCN	Fully Convolutional Network (engl.)
FN	False negative (engl.)
FoV	Field of View (engl.)
FP	False positive (engl.)
FPS	Frames per second (engl.) *Bildwiederholungsrate*
F-SMC	Fuzzy-Sliding Mode Control (engl.) *Fuzzy-Gleitregimeregelung*
GPS	Global Positioning System (engl.) *Globales Positionsbestimmungssystem*
GPU	Graphics Processing Unit (engl.) *Grafikprozessor*
GUI	Graphical User Interface (engl.) *Grafische Benutzerschnittstelle*
HMI	Human Machine Interface (engl.) *Mensch-Maschine-Schnittstelle*
HOG	Histogram of oriented gradients (engl.)
HSV	Hue-Saturation-Value (engl.)
HWC	Height-Width-Channel (engl.)
IMU	Inertia measurement unit (engl.) *Inertiale Messeinheit*
i.O.	in Ordnung
IoT	Internet of Things (engl.)
IPM	Inverse perspective mapping (engl.)
IR	Infrarot
KI	Künstliche Intelligenz
KS	Koordinatensystem
LD	Landmark Detektion
LiDAR	Light detection and ranging (engl.)
MAD	Managed Automated Driving (engl.)
ML	Maschinelles Lernen
MP	Mittelpunkt
MPC	Model Predictive Control (engl.) *Modelprädiktive Regelung*

n.i.O.	nicht in Ordnung
PDC	Park Distance Control (engl.)
RAM	Random-access memory (engl.)
RGB	Rot-Grün-Blau
RNN	Recurrent neural network (engl.) *Rekurrentes neuronales Netz*
RoI	Region of Interest (engl.)
SDK	Software Development Kit (engl.)
SFM	Structure from motion (engl.)
SLAM	Simultaneous localization and mapping (engl.) *Simultane Positionsbestimmung und Kartierung*
SRAM	Static random-access memory (engl.)
SSD	Single-shot detector (engl.)
TN	True negative (engl.)
TP	True positive (engl.)
UZS	Uhrzeigersinn
VO	Visuelle Odometrie
YOLO	You only look once (engl.)

Abbildungsverzeichnis

1.1 Übersicht ausgewählter modularer Fahrzeugkonzepte für die Mobilität der Zukunft [1, 2, 3, 4] . . . 1
1.2 Prinzipskizze der modifizierten Fahraufgabe für den Designprozess [Eigene Darstellung] 4
1.3 Mögliche Setups für die Entwicklung der Rangierfunktion mithilfe eines Fahrroboters [Eigene Darstellung] . . . 5
1.4 Entwicklungsprozess der automatisierten Rangierfunktion [Eigene Darstellung] 6
2.1 Mulitlayer-System für autonomes Fahren [5, S. 325] . . . 8
2.2 Vergleich HWC- und CHW-Layout digitaler Farbbilder [eigene Darstellung] . . . 11
2.3 Mögliche Nichtlinearitäten (Verzeichnungen) bei Kamerabildern [eigene Darstellung] . 15
2.4 Gegenüberstellung des klassischen Machine Learning und des jüngeren Deep Learning-Ansatzes zur Bildklassifizierung [6] . . . 18
2.5 Ausgewählte Grundelemente eines Convolutional Neural Network (CNN)[eigene Darstellung] . . . 20
2.6 Hardware-Setup DIY-Fahrroboter "Jetbot" v0.4.3 [7][eigene Darstellung] . . . 23
3.1 Gegenüberstellung unterschiedlicher Ansätze zur Umfelderkennung [8] . . . 29
4.1 Ausgewählte Beispiele der Resultate einer Objektdetektion basierend auf Pixelfarbwerten [eigene Darstellung] . . . 37
4.2 Ausgewählte Beispiele der Resultate einer Objektdetektion durch Anwendung des Canny-Kantendetektors und einer probabilistischen Hough-Transformation [eigene Darstellung] . . . 39
4.3 Prinzipskizze des punktbasierten Konzepts zur visuellen Detektion des Zielbereichs der Rangierfunktion [eigene Darstellung] . . . 41
4.4 Festlegung der Modell-Outputs und des verwendeten Bildkoordinatensystems [eigene Darstellung] . . . 41
4.5 Auszug aus dem ersten Trainingsdatensatz zur DL-basierten Landmark Detektion und Speichersystematik der Labelinformationen [eigene Darstellung] . . . 46
4.6 Vergleich der ResNet50- (a) und EffNetB0-Modellkonfigurationen (b) zur Landmark Detektion mit Fokus auf die Nutzung der vortrainierten Backbone-Parameter als Initialisierungsschritt (Modellvarianten V2x) [eigene Darstellung] . . . 48
4.7 Manuelle Hyperparameteroptimierung der besten Modellvarianten aus erster Trainingsrunde [eigene Darstellung] . . . 49
4.8 Gegenüberstellung der Lernkurven der optimierten Modelle basierend auf dem ResNet50- und dem EfficientNetB0-Backbone [eigene Darstellung] . . . 51
4.9 Absolute Häufigkeiten der Lage der Flächenmittelpunkte in den Trainingsdaten, jeweils festgelegt durch die vier Landmarks [eigene Darstellung] . . . 53
4.10 Einfache proportionale Antriebssteuerung des Fahrroboters basierend auf der horizontalen Abweichung des Zielobjekts von der Bildmittellinie (x = 0) [eigene Darstellung] 56

4.11 Normierte Regelgröße eines Regelkreises mit sprungförmiger Störung unter Variation der eingesetzten Reglertypen bei gleichbleibender Regelstrecke mit der Übertragungsfunktion $G_S(s) = 1/(1+T_s)^4$ [9, S. 129] 57
4.12 Schematische Darstellung eines Regelungskonzepts bei Mehrgrößensystemen [eigene Darstellung] 58
4.13 Konzept zur relativen Positionsbestimmung basierend auf geometrischen Bildpunktezusammenhängen (qualitativ) [eigene Darstellung] 61
4.14 Pseudo-Code zur Ableitung der Rotationswinkel nach Euler aus einer gegebenen Rotationsmatrix [10] 65
4.15 Homographiebasierte Positionsschätzung der monokularen Kamera in der Ebene: (a) Markierungen auf Boden, (b) Markierungen an Wand [eigene Darstellung] 65
4.16 Trajektorienplanung für die ebene Fahrzeugbewegung innerhalb festgelegter Bereichsgrenzen durch Einfügen einer parabolischen Funktion [eigene Darstellung] 67
5.1 Kontrollstruktur und schematische Darstellung der Schichten der Softwarefunktion (Version 2) [eigene Darstellung] 71
5.2 Integrationstest zur Positioniergenauigkeit der Funktionsversion 2 mit Detektionsmodell ResNet_V211 (07.12.2022) in Büroumgebung bei Tageslicht [eigene Darstellung] 77
6.1 Toleranzbetrachtung bei der hochpräzisen Positionierung des U-Shift Fahrzeugs während des Modulwechsels. [eigene Darstellung, nicht maßstabsgetreu] 83

Tabellenverzeichnis

2.1 Technische Daten des NVIDIA 4GB Jetson Nano^{TM} [11] 24

4.1 DL-Modellkonfigurationen für die Detektion der Landmarks 44

4.2 Evaluation und Verifikation der entwickelten DL-Modelle zur Landmark Detektion basierend auf den jeweiligen Testdatensätzen der Trainingsdatensammlung 54

1 Einführung

Der stetige technologische Fortschritt macht auch vor dem Verkehrssektor nicht halt. Die Automobilindustrie sieht sich mit einer herausfordernden Zukunft konfrontiert mit viel Transformationsbedarf und vor allem einem notwendigen Umdenken der Gesellschaft hinsichtlich der individuellen Fortbewegungsgewohnheiten. Diesen Wandel hin zu einer ressourcenschonenden und klimaneutralen Mobilität gilt es erfolgreich zu meistern.
Die daraus hervorgehenden Megatrends werden in der Fachwelt unter dem Begriff *CASE* zusammengefasst [12]. Im Detail handelt es sich dabei um:

Connected Services: Fahrzeugvernetzung,
Autonomes Fahren,
Shared Mobility: Fahrzeugflotten und gemeinsame Benutzung von Fahrzeugen und
Elektroantriebe.

1.1 Motivation

Ein Fahrzeugkonzept, welches die *CASE*-Trends alle in sich vereint, stellt das *U-Shift* des Instituts für Fahrzeugkonzepte des DLR e.V in Stuttgart dar [1]. Besonderer Fokus liegt hierbei auf einer neuen Art der Modularisierung der Fahrzeugaufbauten. Dieser Ansatz wird ebenfalls von weiteren Forschungsinstitutionen als auch in der Industrie verfolgt. Vergleichbaren Forschungstätigkeiten zu dem *U-Shift* mit einem “on-the-road modular konfigurierbarem Fahrzeugaufbau” [13] gehen unter anderem die Rinspeed AG mit dem Konzept “Snap” [2], die Continental AG mit ihren digitalen Mock-Ups “CUbe” und “BEE” [3] oder die Mercedes-Benz AG mit der Studie “Vision URBANETIC” [4] nach. In Abbildung 1.1 werden diese modularen Fahrzeugkonzepte mit Fokus auf dem *U-Shift* des DLR präsentiert.

Abb. 1.1: Übersicht ausgewählter modularer Fahrzeugkonzepte für die Mobilität der Zukunft [1, 2, 3, 4]

Die Kernidee dieser Konzepte besteht in der Trennung der Antriebseinheit von dem Passagier- oder Transportraum mit der Möglichkeit, den Verbund während des Betriebs zu konfigurieren. Neben Multifunktionalität und Kostensenkung ermöglicht dieses Vorgehen eine effiziente Ausnutzung der Transportmittel, indem sich die flexibel einsetzbaren Fahrkomponenten möglichst in einem 24h-Betrieb befinden. Eine besonders hohe Effizienz wird erzielt, wenn auch der Wechsel der Transportkapseln autonom während des laufenden Betriebs erfolgt, ohne Einsatz von Personal oder längeren Wartezeiten.

Bei den Konzepten der Continental AG und der Mercedes-Benz AG sind Drohnen vorgesehen, welche die unterschiedlichen Kapseln auf das Fahrgestell von oben setzen [3, 4]. Die Rinspeed AG arbeitet mit integrierten Hubbeinen in den Kapseln, um die Antriebseinheit darunter positionieren zu können und die Kapseln schließlich darauf abzusetzen [2]. Das *U-Shift* hebt sich durch die charakteristische U-förmig gestaltete Antriebseinheit von den anderen Konzepten ab. Der Sockel der Kapseln wird als passgenaues Gegenstück zu dieser Aussparung innerhalb der Antriebseinheit ausgearbeitet. Für die Konfiguration des Modulverbunds nutzt das *U-Shift* eine im Fahrwerk integrierte Hubfunktion. Wird der Kapselwechsel initiiert, senkt sich der Aufbau ab, die Transportkapsel wird entriegelt und abgestellt. Die Aufnahme eines neuen Moduls ist nun möglich. Hierzu muss das Driveboard dazu befähigt werden, die vorgesehene Kapsel in einem definierten Bereich zu erkennen, gezielt anzusteuern und sich darunter mit den L-förmigen Tragschienen möglichst mittig zu positionieren. Der gesamte Rangiervorgang findet bei niedrigen Geschwindigkeiten im Bereich des Schritttempos statt. Unterstützt wird die finale Zentrierung des Transportmoduls innerhalb der Antriebseinheit durch konische Elemente an den Führungsschienen. Die Fahrbereitschaft des neuen Verbunds wird durch erneutes Verriegeln und Anheben erreicht. Für einen reibungslosen Ablauf des Modulwechsels sind eine hochpräzise Positionierung und Ausrichtung der Fahreinheit relativ zu der Kapsel entscheidend. Bereits geringe laterale Versätze von wenigen Zentimetern oder kleine Schiefstellungen können zur Verkantung von Driveboard und Kapsel oder zu Kollisionen bei der Aufnahme führen. Je zentrierter die Kopplung der beiden Module erfolgt, desto weniger Verschleiß entsteht auch an den Führungsschienen.
Als Projektziel ist außerdem festgelegt, die hierfür notwendige kostspielige Technik möglichst vollständig in der Antriebseinheit unterzubringen, da hiervon bei einem Dauerbetrieb eine deutlich geringere Stückzahl benötigt wird. Die einfach gehaltenen Transportkapseln lassen sich demnach in großem Umfang produzieren und vermarkten. Sie sind individuell an ihren jeweiligen Einsatzzweck anpassbar und bieten Potenzial sowohl zur Personen- als auch Güterbeförderung [1, 14].

Wie aus den einleitenden Worten hervorgeht, ergeben sich durch die Entwicklung autonomer modularer Fortbewegungsmittel vielversprechende Mobilitätskonzepte. Diese werden jedoch auch von neuen Herausforderungen begleitet, welche einer separaten Betrachtung und Lösung bedürfen. Abgeleitet wird die in dieser Arbeit zu betrachtende Fahraufgabe aus den Anforderungen, welche im Rahmen des U-Shift-Fahrzeugkonzepts an den Transportmodulwechsel gestellt werden. Um einen durchgängig autonomen und somit effizienten Betrieb der Driveboards zu ermöglichen, ist nicht nur die primäre Fahraufgabe - der Transport der Kapseln von Start- zu Endpunkt - fahrerlos zu bewältigen. Auch die “on the road” Modulkonfiguration soll im späteren Betrieb des *U-Shift* durch die Anwendung einer für diesen speziellen Anwendungsfall entwickelten Rangierfunktion autonom gelöst werden.

1.2 Zielsetzung

Der Fokus der vorliegenden Arbeit liegt auf dem Design einer Softwarefunktion als Proof of Concept einer rein kamerabasierten, automatisierten Abwicklung eines Rangiervorgangs durch den Einsatz eines Fahrroboters als Rapid Prototyping-Werkzeug.
In erster Linie ist ein Computer Vision - Ansatz durch Verwendung monokularer exteroperceptiver Kameras zur Erfassung der notwendigen Daten für die autonome Entscheidungsfindung zu verfolgen. Zur Extraktion und Verarbeitung der Informationen in den gewonnenen Bilddaten ist der Stand der Technik aus dem Bereich *Computer Vision* in Kombination mit den Fortschritten in dem Forschungs- und Entwicklungsfeld *Künstliche Intelligenz* (v.a. Maschinelles Lernen und/oder Deep Learning) zu berücksichtigen. Fundierte Kenntnisse in diesem Umfeld dienen als Ausgangsbasis für die Entwicklung der Rangierfunktion, vergleiche Kapitel 2 und 3.
Besonders die Grenzen und die bestehenden Herausforderungen einer rein visuellen und monokularen Umfelderfassung als essentielle Softwareschicht sind hierbei herauszuarbeiten. Neben der Funktionsvalidierung ist außerdem die Übertragbarkeit des Ergebnisses auf den im vorangegangenen Kapitel aufgezeigten Kapselwechsel des *U-Shift* zu erörtern, vergleiche Kapitel 6.

Während der Designphase wird gezielt auf einen miniaturisierten Fahrroboter im Büromaßstab zurückgegriffen. Dieses Vorgehen ermöglicht einen agilen Entwicklungsprozess des Softwareprototypen und eine frühzeitige Funktionsvalidierung als auch Evaluierung der Eignung des Konzepts für das *U-Shift*. Die automatisiert zu erfüllende Fahraufgabe ist hinsichtlich der verfügbaren Hardware anzupassen. Bei dem verwendeten System handelt es sich nicht um eine skalierte Repräsentation der ursprünglichen Ausgangssituation, Modulverbund aus feststehender Kapsel und beweglicher Antriebseinheit, welche in Kapitel 1.1 näher beschrieben wurde. Die Rollen von Driveboard und Kapsel sind im Folgenden vertauscht. Der Kontext bleibt jedoch weiterhin erhalten, auch wenn die Kapsel als bewegliches Objekt mit eingebauter Sensorik und der U-förmige Bereich zwischen den beiden Flügeln des Driveboards als statisches Zielobjekt betrachtet werden. Der Fahrroboter stellt somit die Kapsel dar, welche sich unter Zuhilfenahme exteroperceptiver Sensorik relativ zu einem geometrisch definierten Zielbereich positioniert und diesen möglichst zentriert befahren muss. Die erläuterten Modifikation der ursprünglich zu automatisierenden Fahraufgabe illustriert die Prinzipskizze in Abbildung 1.2.

Zur Eingrenzung des U-förmigen Zielbereichs sind im weiteren Verlauf der Arbeit matt gelbe Markierungsstreifen mit einer Länge von ca. 37 cm und einer Breite von ca. 10 cm aus dem Freizeitsportbereich vorgesehen. Berücksichtigung finden zwei mögliche Anwendungsfälle solcher Kennzeichnungen:

Fall 1: Die Markierungsstreifen definieren wie bei Parkplätzen den zur Verfügung stehenden Bereich für das Abstellen des Wagens, siehe die primären Bereichsbegrenzungen in Abbildung 1.2. Hierfür werden diese parallel mit definiertem Abstand auf dem Boden direkt angrenzend an eine feste Wand (= sekundäre Begrenzung) angebracht. Die dadurch festgelegte Fläche mit einem definierten Einfahrtsbereich legt mit ihrem Mittelpunkt die Endposition des Fahrroboters fest.

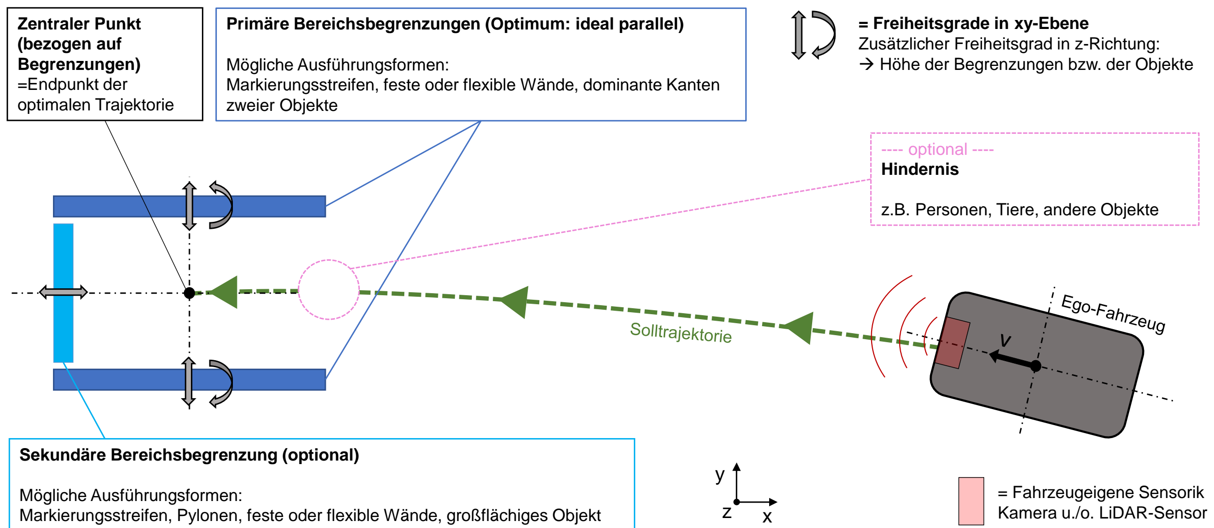


Abb. 1.2: Prinzipskizze der modifizierten Fahraufgabe für den Designprozess [Eigene Darstellung]

Fall 2: Anstelle von Bodenmarkierungen lassen sich die Streifen alternativ auch hochkant und unter Einhaltung einer parallelen Anordnung an einer Wand oder einer ähnlichen senkrechten Oberfläche platzieren. Auch in dieser Konfiguration wird eine Fläche definiert, welche zwar nicht direkt einen Einfahrtsbereich für den Roboter festlegt, aber dennoch einen anzusteuernden Zielbereich vorgibt.

Diese aus der Prinzipskizze abgeleitete Definition der Randbedingungen wird durch die Darstellung der beiden relevanten Setups in Abbildung 1.3 verdeutlicht.

Insofern sich der Roboter vor der Einfahrtsöffnung des U-förmigen Zielbereichs befindet und sich dieser vollständig im Sichtfeld der Frontkamera befindet, detektiert das System die Bereichseingrenzungen als statische Zielobjekte. Die Objekterkennung ist echtzeitfähig und möglichst robust gegenüber umweltbedingten Störeinflüssen auszulegen, damit eine einwandfreie Erfassung des Bereichs "on the fly" während des Fahrbetriebs realisierbar ist. Neben der Objekterkennung ist außerdem eine Lokalisierung dieser relativ zu dem Egofahrzeug (Fahrroboter) erforderlich. Abhängig von der erkannten Position muss das System in der Lage sein, eine geeignete Fahrstrategie zu wählen, um in möglichst kurzer Zeit und mit geringem Korrekturaufwand eine zentrierte Einfahrt in den Zielbereich durchzuführen. Hierbei sind geeignete Toleranzen hinsichtlich der zulässigen Verdrehung und dem maximalen lateralen Versatz zwischen der Mittellinie des Fahrroboters und des Zielbereichs festzulegen. Eine mögliche Skalierung dieser Rahmenbedingungen auf das *U-Shift* ist während der Entwicklung nicht explizit zu berücksichtigen, da der Funktionsprototyp nicht für einen späteren Einsatz im Realfahrzeug vorgesehen ist. Eine optionale Erweiterung stellt die Verknüpfung der Fahrfunktion mit einem Kollisionsvermeidungssystem (engl.: Collision Avoidance) dar. Dieses Vorgehen zieht somit auch die Existenz von möglichen Hindernissen auf dem Kurs des Fahrroboters in Betracht. Für die angestrebte Erarbeitung einer Grundfunktionalität ist dies jedoch nicht zwingend erforderlich.

Wie aus den Abbildungen 1.2 und 1.3 hervorgeht, besitzt die automatisiert zu lösende Fahraufgabe nun deutliche Parallelen zu einem Parkvorgang. Aus diesem Grund wird bei der Analyse des technischen Standes bei der kamerabasierten Umfeldwahrnehmung das

Augenmerk auf prinzipverwandte Systeme zur Fahrbahnspurerkennung und zur Darstellung autonomer Parkvorgänge gelegt, vergleiche Kapitel 3.2. Sie dienen als Benchmark für die Entwicklung der Rangierfunktion, besonders hinsichtlich der für die korrekte Funktionalität entscheidenden Perception-Task.

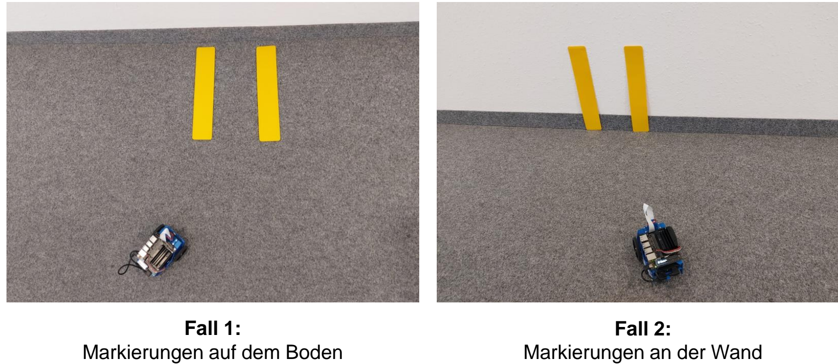


Abb. 1.3: Mögliche Setups für die Entwicklung der Rangierfunktion mithilfe eines Fahrroboters [Eigene Darstellung]

1.3 Aufbau der Arbeit

Angestrebt wird die Aufteilung der Funktionalität in in sich geschlossene Aufgabenbereiche - Perception, Lokalisierung, Trajektorienplanung und Steuerung. Diese Entscheidung basiert auf dem in der Automobilindustrie weit verbreiteten Vorgehen zur Entwicklung von Softwarelösungen für autonomes Fahren, welches in Kapitel 2.1 genauer erörtert wird. Die Entwicklung der jeweiligen Funktionsbausteine erfolgt agil und zumeist parallelisiert. Bei Erreichen eines Meilensteins werden die entsprechenden Komponenten versioniert, integriert und schließlich die Funktionalität mithilfe des eingesetzten Fahrroboters validiert. Somit wird ein evolutionärer Softwareprototyp aufgebaut, welcher jederzeit erweitert und an sich ändernde Randbedingungen angepasst werden kann. Eine solches Entwicklungsprozessmodell findet neben dem allgemein bekannten V-Modell ebenfalls Anwendung bei der Softwareentwicklung in der Industrie [15].
Auch der Aufbau der folgenden Abhandlung spiegelt den Entwicklungsprozess wider. Abbildung 1.4 visualisiert qualitativ den Zusammenhang zwischen dem Aufbau der Arbeit und der Zeitschiene der Entwicklungsphase.

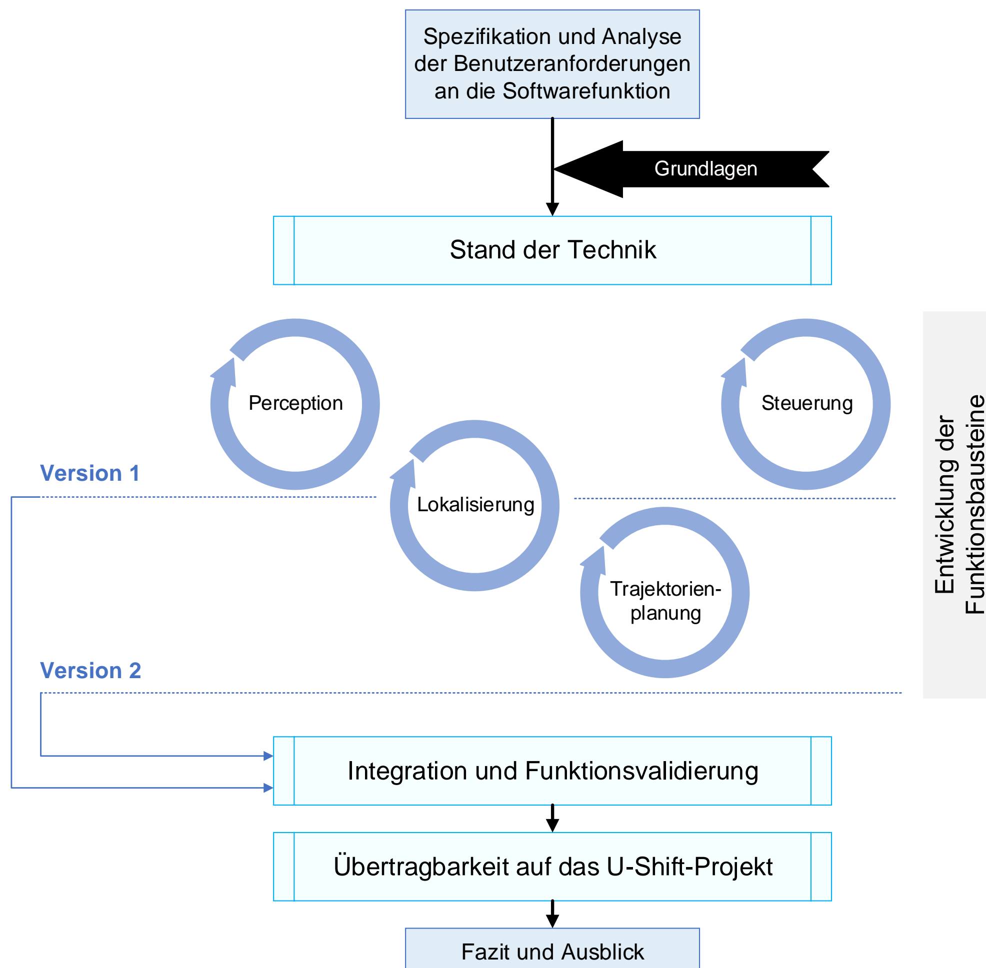


Abb. 1.4: Entwicklungsprozess der automatisierten Rangierfunktion [Eigene Darstellung]

2 Grundlagen

Die folgenden Abschnitte dienen als Fundament für das Verständnis des weiteren Inhalts. Neben der Vorstellung von Begrifflichkeiten, welche wiederkehrend in der vorliegenden Arbeit auftreten, soll vor allem der Wissensstand der Leserschaft egalisiert werden. Zunächst erfolgt eine Erörterung der Subsysteme, welche zur Realisierung autonomer Fahrfunktionen benötigt werden. Die auf Automobilanwendungen zugeschnittenen Grundlagen der Fachdisziplinen Computer Vision (CV) und Maschinelles Lernen (ML; engl.: Machine Learning), welche in der Entwicklung der Rangierfunktion eine entscheidende Rolle spielen, werden anschließend aufgeführt. Den Abschluss dieses Grundlagenkapitels stellt eine kurze Beschreibung des für das Rapid Prototyping verwendeten Fahrroboters dar.

2.1 Subsysteme für Autonomes Fahren

Grundlegend existieren zwei gegensätzliche Herangehensweise, um einem Fahrzeug zur Autonomie zu verhelfen. Um den Menschen im System so realitätsgetreu wie möglich durch logische und arithmetische Berechnungen zu ersetzen, sind sogenannte End-to-End-Anwendungen zu bevorzugen. Hierbei werden aus vorgegebenen Eingabegrößen, meist in Form sehr großer Datenmengen, ohne Zwischenschritte oder jeglichen Datentransfer, direkt verwendbare Ausgabewerte bestimmt [12]. Eine solche Implementierung eines autonomen Systems bildet die Intuition des Menschen bei der Entscheidungsfindung sehr gut ab. Aus der Vielzahl an Sinneseindrücken werden in kürzester Zeit auszuführende Aktionen abgeleitet ohne Zugriff auf die Verarbeitungsschritte, die währenddessen durchlaufen werden. Als Beispiel sei an dieser Stelle der von Rathour et. al (2018) vorgestellte Ansatz einer End-to-End-Parkfunktion für autonome Fahrzeuge genannt. Nach erfolgter Extraktion der relevanten Informationen aus den Kamerabildern durch ein neuronales Netz in Kombination mit weiteren Optimierungen wird unmittelbar eine Lenkwinkelvorgabe an das Regelungssystem der Fahrzeugdynamik ausgegeben, um das Parkmanöver ordnungsgemäß durchzuführen [16]. Die bei dieser Lösungsstrategie eingesetzten Systeme besitzen oft eine sehr hohe Komplexität, weshalb die Rechenschritte zwischen In- und Output nur schwer logisch nachvollziehbar und kaum zugänglich sind - ähnlich einer Black Box.
Demgegenüber stehen die hohen Sicherheits- und Qualitätsanforderungen in der Automobilindustrie, welche nur durch beherrschte Systeme und Prozesse erfüllt werden können. Die übergeordnete und vielschichtige Aufgabe “Autonomes Fahren” kann auf mehrere in sich schlüssige und überschaubare Ebenen heruntergebrochen werden. Erst durch das Zusammenwirken dieser eigenständigen Untersysteme wird die Autonomie des Gesamtsystems dargestellt [12]. Etabliert hat sich in diesem Umfeld in den letzten Jahren die serielle Verknüpfung der einzelnen Komponenten durch einen durchgängigen Datenstrom. Der Output eines Bausteins wird direkt als Input an den Nächsten übergeben, bei Bedarf mit Zwischenschritten zur Datentransformation[17]. Eine solche “Daten-Pipeline” ist wie eine sequentielle Abfolge von Datenverarbeitungsschritten zu verstehen, wobei ein direkter Transfer der Daten über den Arbeitsspeicher (RAM, DRAM, SRAM) oder einen Zwischenspeicher zwischen den einzelnen Schichten ohne laufzeitverlängernde oder ressourcenintensive Zugriffe auf den Datenspeicher des Mikroprozessors erfolgt [18].
Eine solche Unterteilung des Berechnungspfades zwischen den Rohdaten und der gewünschten Ausgabe in einzelne verknüpfte Schichten wird in Abbildung 2.1 aufgezeigt.

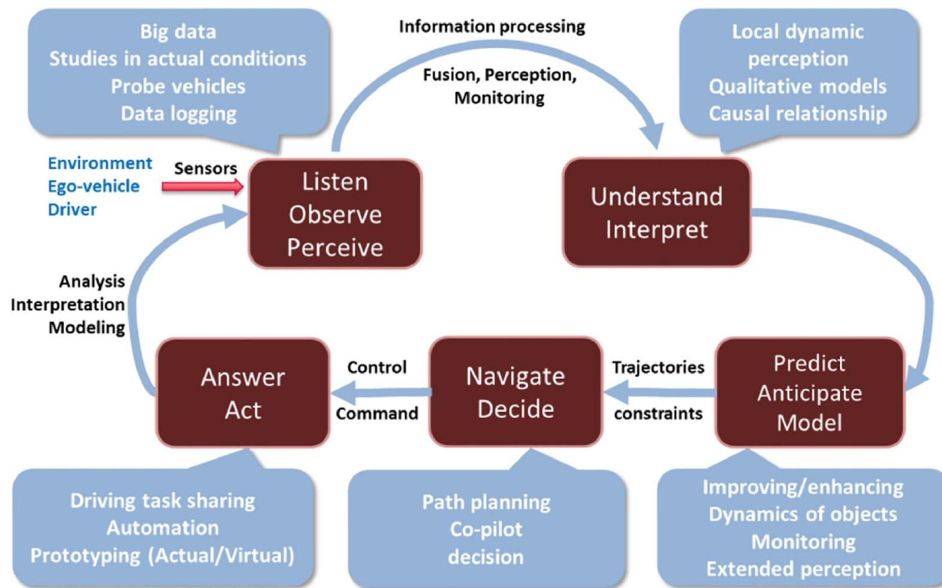


Abb. 2.1: Mulitlayer-System für autonomes Fahren [5, S. 325]

Angelehnt an diese Einteilung lassen sich folgende elementare Bausteine zur Realisierung einer vollautomatisierten Fahrfunktion definieren:

- **Perception** (oder auch Wahrnehmung):
  Das Umfeld des Fahrzeugs ist mithilfe geeigneter Sensorik zu erfassen und anschließend die Situation, in der sich das Fahrzeug befindet, zu interpretieren. Wichtige Elemente sind hierbei unter anderem die Objekterkennung oder die Identifikation von markanten Attributen der Umgebung, um daraus Rückschlüsse auf vorhandene Begebenheiten und Randbedingungen (z.B. Witterungs- und Straßenverhältnisse) zu ziehen. Neben der Überwachung der äußeren Einflussgrößen auf die Fahrsituation fällt unter den Begriff Perception auch die Erfassung interner Systemvariablen, der sogenannten Odometriedaten, anhand verschiedener Sensoren. Die Perception bildet die essentielle Grundlage für alle folgenden Funktionsbausteine, da sie in jedem Zeitschritt Informationen in Form eines Verständnisses der Situation, in der sich das Fahrzeug momentan befindet, bereitstellt. Diese werden direkt von den daran anknüpfenden weiteren Algorithmen genutzt und verarbeitet.

- **Localization** (oder auch Lokalisierung):
  Dieser Baustein hat zur Aufgabe, das Fahrzeug in den Kontext der ermittelten Umgebung einzufügen. Wo genau befindet sich das Fahrzeug in der aktuellen Situation? Festzulegen ist, ob die absolute Position in einem festen Welt-Koordinatensystem oder die Lage des Ego-Fahrzeugs relativ zu Bezugspunkten aus dem wahrgenommenen Umfeld gesucht wird.

- **Planning** (oder auch Trajektorienplanung):
  Nach der erfolgreichen Auffassung der Fahrsituation zusammen mit der Kenntnis über den Standort des Fahrzeugs kann eine optimale Trajektorie zur Erfüllung der

übergeordneten Fahraufgabe definiert werden. Der Begriff “optimal” ist allerdings stark situationsabhängig und ergibt sich zudem aus den zuvor festgesetzten Optimierungsparametern und Randbedingungen. So wird zum Beispiel bei einem normalen Fahrspurwechsel auf der Autobahn eher eine Trajektorie mit möglichst geringer resultierender Querbeschleunigung als optimal angesehen unter dem Aspekt des maximalen Insassenkomforts. Befindet sich allerdings das autonome Fahrzeug in einer Situation, in der ein Auffahrunfall auch durch eine Notbremsung nicht mehr zu verhindern wäre, so könnte in dieser speziellen Notlage ein abrupter Wechsel in eine benachbarte freie Fahrspur als optimal angesehen werden.

- **Control** (oder auch Steuerung):
  Um der vorgegebenen Trajektorie folgen zu können, muss diese Information schließlich in Stellgrößen für die an der Fahraufgabe beteiligte Aktuatorik transferiert werden. Für diese Aufgabe können die bekannten Schnittstellen zur Vorgabe von Führungsgrößen durch den menschlichen Fahrer genutzt werden - die Gaspedalstellung, die Bremspedalstellung und der Lenkradwinkel. Da jedoch die Mensch-Maschine-Schnittstelle (engl.: Human Machine Interface, HMI) im autonomen Fahrzeug entfallen kann, muss nicht mehr zwingend auf diese Stellglieder zurückgegriffen werden. Ein eleganter Weg ist die Verwendung von internen Größen, welche bisher durch den Fahrer nicht beeinflussbar waren. Beispielhaft, ohne Anspruch auf Vollständigkeit, fallen hierunter das Antriebsdrehmoment an den Rädern, der Bremsflüssigkeitsdruck oder eine Lenkwinkelvorgabe über die Höhe des Drehmoments und die Drehrichtung eines E-Motors am Lenkgetriebe im Kontext einer vollelektrischen Lenkung. Generell spricht man hierbei von Drive-by-Wire-Systemen.

2.2 Computer Vision

Wenn Computer das Steuer im Fahrzeug vollständig übernehmen sollen, dann ist die Fähigkeit gefordert, Eindrücke aus der Umgebung des Fahrzeugs aufzufassen und auf diese dynamisch zu reagieren [12]. Da nicht davon ausgegangen werden kann, dass in absehbarer Zeit eine radikale Ausrichtung der Infrastruktur auf autonome Fahrzeuge weltweit stattfinden wird, müssen die Automatisierungssysteme sich der vorhandenen Informationsquellen im Straßenverkehr bedienen und sich darin zurechtfinden. Die heutige Gestaltung des globalen Straßennetzes zielt hauptsächlich auf die visuelle Wahrnehmung des Menschen ab. Verkehrsschilder mit ihrer spezifischen Form- und Farbgebung zur Festlegung von geltenden Verkehrsregeln, Ampeln mit ihren charakteristischen Farben Rot, Gelb und Grün zur Regelung des Verkehrsflusses oder auch die weißen oder gelben Fahrbahnmarkierungen mit ihrem hohen Kontrast zum Asphalt werden vom menschlichen Fahrer leicht gesehen und registriert.
Mit der Frage, wie man bei einem Computer oder Roboter genau dieses Sehen realisiert und ihm somit Einblicke in seine Umgebung ermöglicht, beschäftigt sich das große und eigenständige Fachgebiet der Computer Vision.
Um CV-Algorithmen anwenden zu können, bedarf es zunächst der Bereitstellung von Rohdaten durch eine geeignete Sensorik. Im Folgenden werden dementsprechend die Grundlagen zur Datenerzeugung unter dem Einsatz von Farbbildkameras aufgeführt.

Die Digitalkamera erzeugt aus Photonen, welche auf geordnete Bildsensoren treffen, elektrische Signale. Kameras sind passive Sensoren und arbeiten wie der menschliche Sehapparat im sichtbaren Wellenlängenbereich (ca. 350 nm bis 700 nm). Auf die Erzeugung von Digitalbildern wird an dieser Stelle nicht im Detail eingegangen. Vielmehr ist die Datenstruktur und die Bedeutung von Bildpunkten als Projektion von realen Punkten des dreidimensionalen Raums in den Fokus zu stellen. Speziell werden Besonderheiten von monokularen Kameras mit großem Field of View (FoV), auch bekannt als Fish-Eye-Kameras - herausgearbeitet, da diese einfache und kostengünstige Kameraausführung für Anwendungen im Automotive-Bereich eine hohe Attraktivität besitzt.

Für die gezielte Extraktion von Informationen aus den bereitgestellten Rohdaten einer Digitalkamera muss man sich der Datenstruktur vorab bewusst sein. Grundsätzlich werden bei der Digitalisierung der eingefangenen Lichtstrahlen mehrere Quantisierungsschritte durchlaufen. Bereits beim Auftreffen der Photonen auf die flächig und rasterförmig angeordneten Bildsensoren findet eine erste Diskretisierung der zweidimensionalen Bildebene statt. Die Bildsensoren bilden somit die kleinsten Bestandteile eines digitalen Bildes - die Pixel - zwischen denen unterschieden werden kann. Bei Farbbildkameras wird innerhalb jedes Pixels ein Farbwert je nach detektierter Lichtintensität der Wellenlängenbereiche der drei Grundfarben Rot, Grün und Blau (RGB-Farbraum) gebildet und dessen Wertebereich je nach Farbtiefe quantisiert. So kann mit einer oft verwendeten Wortlänge von 8 Bit zwischen 256 (2^8) ganzzahligen Werten differenziert werden, was einem Wertebereich von [0...255] entspricht. An dieser Stelle ist anzumerken, dass manche Kameras auch das BGR-Farbmodell verwenden, wodurch es schnell zu falschen Darstellungen von Farben kommt, wenn BGR-Farbchannel unter Annahme eines RGB-Formats eingelesen und visualisiert werden. Letztlich folgt eine zeitliche Diskretisierung bestimmt durch die Abtastrate der elektronischen Signale der Bildsensoren, wodurch sich die Aktualisierungsrate oder auch Bildwiederholungsrate (engl.: Frames per second, FPS) bei Digitalkameras ergibt [19].
Es sei ergänzend der HSV-Farbraum begrifflich eingeführt, welcher ebenfalls neben den wohl bekannten RGB- und BGR-Modellen über die Vorgabe von drei Parametern eine exakte Definition eines Farbwerts erlaubt. Dieses Farbmodell bewegt sich nahe der menschliche Farbwahrnehmung [20].

Mathematisch werden die erzeugten digitalen Bilddaten in Form von Matrizen beschrieben. Jedes Element einer solchen Bildmatrix stellt ein Pixel dar, welches einen speziellen Farbwert repräsentiert. Somit besitzt jedes Pixel eine feste Adresse, über die auf den dort befindlichen Wert zugegriffen werden kann. Dies mündet in der Abgrenzung zwischen dem HWC- und dem CHW-Speicherlayout der Bilddaten. Bei Ersterem wird strikt die mathematische Abbildungsvorschrift in Matrizenschreibweise angewendet. Die erste Dimension der Matrix enthält alle Bildzeilen und bestimmt somit die Höhe (engl.: Height) in Pixeln. Über die zweite Dimension werden die Bildspalten aufgeschlüsselt, wobei die Anzahl dieser Spalten die Breite (engl.: Width) in Pixeln widerspiegelt. Bei diesem Layout befinden sich unter Annahme eines Farbbildes die Werte der einzelnen Farbchannel als Vektor in einer dritten Dimension der Bildmatrix. Es handelt sich somit um ein Vektorfeld mit den Bildpixeln als diskrete Punkte in der Ebene [20]. Dem gegenüber steht das CHW-Layout, welches in der ersten Matrixdimension die Farbchannel differenziert und darauf folgend die Bildzeilen und -spalten. Es werden mehrere Filter erzeugt mit konstanter Höhe und Breite, wobei jedes Filter für jedes Pixel den Wert eines Farbchannels enthält.

Das Bild in Farbe entsteht durch eine Überlagerung dieser Filter oder Channel, was eine intuitivere Anwendung des Prinzips des additiven Mischens der Grundfarben darstellt. Je nach verwendetem Farbraum, also Anzahl der Farbchannel, werden die Bilddaten in mehrere zweidimensionale Matrizen unterteilt. Abbildung 2.2 verdeutlicht diese Unterschiede durch eine Gegenüberstellung der beiden grundlegenden Speicherlayouts von digitalen Kamerabildern am Beispiel eines RGB-Farbbildes mit einer Auflösung von 7 x 8 Pixeln und einer Farbtiefe von 8 Bit.

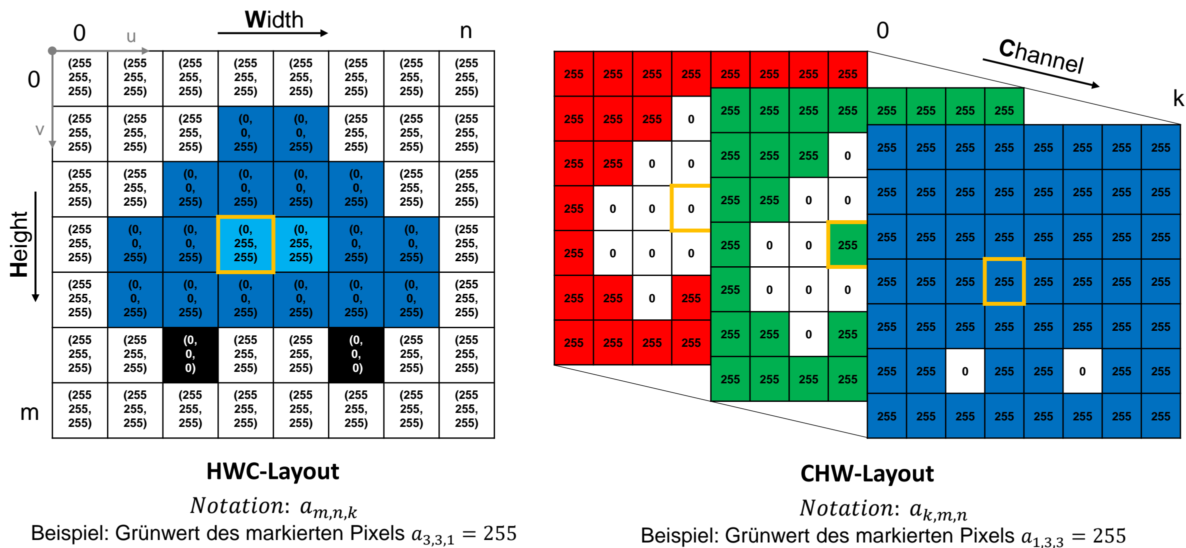


Abb. 2.2: Vergleich HWC- und CHW-Layout digitaler Farbbilder [eigene Darstellung]

Zwar liegen nun zu diskreten Orten in einer Bildebene gewisse Werte vor, welche ein Abbild der eingefangenen Szene mit dem Ursprung im oberen linken Eck darbieten, aber allein aus diesen Daten ist ohne weitere Angaben kein Rückschluss auf die tatsächliche Position oder die Größe von den dargestellten Bildobjekten zu ziehen. Um die Relation zwischen den zweidimensionalen Bildpunkten und den im Sichtfeld befindlichen dreidimensionalen Körpern mathematisch eindeutig festzulegen, wird das Zentralprojektionsmodell herangezogen. Rein mathematisch betrachtet handelt es sich hierbei um zwei Koordinatentransformationen und eine Projektion der realen 3D-Koordinaten in die Bildebene. Diese Abbildungsvorschrift ist vollständig definiert, wenn folgende Matrizen bekannt sind [21, 22]:

Extrinsische Matrix
3D-Koordinatentransformation: Welt-KS (X_W, Y_W, Z_W) $\Leftrightarrow$ Kamera-KS (X_K, Y_K, Z_K)

Das Koordinatensystem einer monokularen Kamera besitzt seinen Ursprung im Objektiv und weist eine Ausrichtung der z-Achse entlang der optischen Achse auf. Eine solche Vorgabe besitzt das Welt-KS nicht, da es im Raum ortsfest, aber dennoch frei positionierbar ist. Aufgabe der vorzunehmenden Tranformation ist, die translatorische Verschiebung $\vec{t}$ und die Rotationsmatrix R zwischen beiden Koordinatensystemen zu bestimmen, sodass ein bekannter 3D-Punkt P_W in der realen Welt über Formel 2.1 in das kartesische Kamera-KS (P_K) überführt werden kann. Mit Formel 2.2 erfolgt zusätzlich eine Aufschlüsselung der Komponenten der Rotationsmatrix.
Dieser mathematische Zusammenhang wird in der projektiven Geometrie meist in Form

einer homogenen Matrix E nach Gleichung 2.3 mit der Bezeichnung “Extrinsische Matrix” wiedergegeben. Durch die Einführung homogener Koordinaten $\tilde{P}_K$ und $\tilde{P}_W$ erfolgt eine Dimensionserweiterung, welche einen Skalierungsfaktor zwischen dem realen Objekt und seinem zweidimensionalem Abbild berücksichtigt. Sind alle 12 Parameter der 4 x 4 Matrix E mit Rang 3 bekannt, lässt sich die Koordinatentransformation durch eine einfache Matrixmultiplikation des homogenen Raumpunktes mit der Matrix E realisieren, siehe Formel 2.4. Die gesuchten Parameter sind abhängig von den 6 Freiheitsgraden im dreidimensionalen Raum (dreimal translatorisch: t_x, t_y, t_z , dreimal rotatorisch $\vartheta, \varphi, \omega$)[21]. Sie sind auch als “Parameter der äußeren Orientierung einer Kamera” [20] bekannt.

$$P_K = \mathbf{R}\, P_W + \vec{\mathbf{t}} \tag{2.1}$$

$$\mathbf{R} = \begin{pmatrix} cos(\vartheta) & -sin(\vartheta) & 0 \\ sin(\vartheta) & cos(\vartheta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\varphi) & -sin(\varphi) \\ 0 & sin(\varphi) & cos(\varphi) \end{pmatrix} \begin{pmatrix} cos(\omega) & -sin(\omega) & 0 \\ sin(\omega) & cos(\omega) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

$$\mathbf{E} = \left[\begin{array}{c|c} \mathbf{R} & \vec{\mathbf{t}} \\ 0^T & 1 \end{array}\right] = \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{array}\right] \tag{2.3}$$

$$\tilde{P}_K = \mathbf{E}\, \tilde{P}_W \tag{2.4}$$

Intrinsische Matrix
Perspektivische Transformation: Kamera-KS (X_K, Y_K, Z_K) ⇔ Bildsensor-KS (x, y)
Interne Transformation: Bildsensor-KS (x, y) ⇔ Diskrete Bildkoordinaten (u, v)

Unter der Annahme einer allgemein gültigen Zentralprojektion ist festgelegt, dass sich alle Lichtstrahlen im Brennpunkt der Kameralinse, dem zentralen Punkt, schneiden. Wird die Bildebene, welche die photoelektrischen Bildsensoren enthält, entlang der optischen Achse gedanklich verschoben, erfolgt eine skalierte Abbildung der Realität in der zweidimensionalen Ebene, je nach Abstand zum Brennpunkt der Kameraoptik. Man spricht in diesem Zusammenhang bei der Distanz zwischen der retinalen Ebene und dem Brennpunkt von der Brennweite f (engl.: focal length) als interner Kameraparameter. Den Ursprung des zweidimensionalen Koordinatensystems bildet der Kamerahauptpunkt als Durchstoßpunkt der optischen Achse in der Bildebene. Festgehalten wird diese Projektionsvorschrift ebenfalls in Form einer auf die Brennweite f = 1 normierten Matrix, siehe Formel 2.5. Da die Bildsensoren örtlich quantisierte Signale liefern, ist eine Überführung der kontinuierlichen Bildsensorkoordinaten (x, y) in die diskreten Bildkoordinaten (u, v) anhand der Faktoren k_u und k_v notwendig. Diese Faktoren stellen jeweils den Kehrwert der Kantenlänge eines Pixels entlang der entsprechenden Koordinatenachse dar. Zusätzlich erfolgt eine Verschiebung des Bildursprungs durch die Parameter u_0 und v_0 in die linke obere Ecke des Bildes, um der üblichen Nummerierung der Zeilen und Spalten der Bildmatrix Rechnung zu tragen, vergleiche Abb. 2.1. Um diese Matrix zahlenmäßig definieren zu können, müssen somit vier Freiheitsgrade eingeschränkt werden [21]. Oft wird in der Literatur zusätzlich ein Scherungswinkel (engl.: skew factor) als weiterer, fünfter Parameter angegeben, welcher jedoch aufgrund der stetig sinkenden Fertigungstoleranzen der Bild-

sensoren immer mehr an Bedeutung verliert und dementsprechend als vernachlässigbar angesehen werden kann [ebd].
Beide Rechenoperationen können unter Verwendung der Intrinsischen Matrix I gleichzeitig abgedeckt werden. Die Pixelkoordinaten im digitalen 2D-Bild stellen schließlich das Produkt aus Intrinsischer Matrix I und dem realen Punkt im Kamera-KS $\tilde{P}_K$ dar, siehe Gleichung 2.6.

$$\mathbf{I} = \underbrace{\begin{bmatrix} k_u & 0 & u_0 \\ 0 & k_v & v_0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Interne Trans.}} \cdot \underbrace{\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Perspekt. Trans.}}$$

$$= \begin{bmatrix} a_u & 0 & u_0 \\ 0 & a_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} mit\ f = 1,\ a_u = fk_u,\ a_v = fk_v \tag{2.5}$$

$$\tilde{p}_{u,v} = \mathbf{I}\,\tilde{P}_K \tag{2.6}$$

Zwischen den Pixelkoordinaten und den realen 3D-Punktkoordinaten kann nun bidirektional umgerechnet werden, wenn das Produkt aus Extrinsischer und Intrinsischer Kameramatrix bzw. die insgesamt 10 Freiheitsgrade (exklusive des Scherungsfaktors der Bildsensoren) bekannt sind. Da die zuvor eingeführten homogenen Koordinaten eine weitere Dimension für den Skalierungsfaktor S besitzen, ist eine eindeutige Lösung der gesamten projektiven Koordinatentransformation nur durch eine Normierung der Koordinaten auf S möglich. Letztlich wird der Abbildungsprozess in diesem Kameramodell durch zwei lineare Gleichungen für u = U/S und v = V/S beschrieben [21]. Die Gesamtgleichung mit der allgemeinen Transformationsmatrix Q und den darin aufgelisteten Quaternionen $q_{i,j}$ lautet wie folgt:

$$\tilde{p}_{u,v} = \begin{bmatrix} U \\ V \\ S \end{bmatrix} = \mathbf{I}\,\mathbf{E}\,\tilde{P}_W = \mathbf{Q}\,\tilde{P}_W = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \end{bmatrix} \cdot \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix} \tag{2.7}$$

Als Untergruppen der komplexen projektiven Transformation existieren weitere vereinfachte Kameramodelle, welche die Anzahl an unbekannten Parametern gezielt reduzieren. Unter anderem fallen hierunter die orthografische Projektion (auch Parallelprojektion) mit insgesamt 8 Parametern und die affine Projektion mit 9 Parametern, jeweils exklusive dem Scherungsfaktor [20].
Der komplexe Abbildungsprozess vereinfacht sich auch, wenn man spezielle Fälle betrachtet. Befinden sich alle 3D-Punkte im Raum auf einer Fläche (Koplanarität) und es werden Bilder aus zwei unterschiedlichen Perspektiven eingefangen oder zwei Bildebenen besitzen denselben Kamerahauptpunkt und unterscheiden sich nur durch eine reine Rotation, dann besteht zwischen den Bildpunkten in den Bildebenen ebenfalls ein mathematischer Zusammenhang [22, 23]. Dieser Spezialfall ist unter dem Begriff Homographie

bekannt [ebd]. Die Bildpunkte können direkt über eine 3 x 3 Matrix H von einer Bildebene in die andere transferiert werden, insofern entsprechende Punktkorrespondenzen zur Berechnung der Matrixelemente vorliegen. Die Standard-Bibliothek für Computer Vision *OpenCV* bietet hierfür eine vorgefertigte Funktion zur Ermittlung von H über vier vorzugebende Referenzpunkte, welche die acht Freiheitsgrade einschränken [24]. Die Lösung ist ebenfalls bis auf einen Skalierungsfaktor eindeutig.

Wie aus den vorangegangenen Erläuterungen hervorgeht, lässt sich der gesamte Abbildungsprozess einer Digitalkamera durch je nach gewähltem Kameramodell mehr oder weniger Parametern beschreiben. Das Auffinden dieser Parameter erfolgt über eine Kamerakalibrierung, bei der genau der oben beschriebene mathematische Zusammenhang zwischen den Koordinaten eines Punktes im Welt-KS und dessen zweidimensionale Abbildung zahlenmäßig erörtert wird. Es ist zu beachten, dass die extrinsischen Parameter von der Bewegung der Kamera abhängig sind und nur die Werte der Intrinsischen Matrix nach einmaliger Bestimmung ohne Änderung an der Hardware der Kamera, vor allem der Brennweite f und den Bildsensoren, konstant bleiben. Ist die Bewegung des Kamera-KS in dem festen Welt-KS nicht mathematisch eindeutig beschreibbar, müsste nach jeder Lageänderung eine Kalibrierung stattfinden, um die äußeren Kameraparameter zu aktualisieren. Unterstützt wird der herkömmliche, manuelle Kalibriervorgang durch Kalibrierkörper, welche 3D-Punkte in einem festen Welt-KS vorgeben [20]. Wird der Fall betrachtet, dass es sich um eine bewegliche Kamera handelt, wie es bei der Anwendung im Fahrzeug zu erwarten ist, so können auch 3D-Referenzpunkte mit bekannter Lage im Welt-KS zur Kalibrierung herangezogen werden. Hiervon sind mindestens vier Stück erforderlich, um die acht Parameter (4 x 2 Gleichungen) des einfachen Parallelprojektionsmodells zu definieren und bestenfalls dienen 5 bis 6 bekannte Punkte zur Bestimmung der mindestens 10 Größen des allgemeinen linearen Zentralprojektionsmodells. Ohne manuelle Vorgabe dieser Punkte zur Kalibrierung im Raum arbeiten Algorithmen zur Selbstkalibrierung, welche aus mehreren Bildaufnahmen aus unterschiedlichen Blickwinkeln Punktkorrespondenzen berechnen und diese statischen Punkte schließlich zur Kalibrierung nutzen [25]. Ein weiteres, ähnliches Vorgehen stellt die planare Kalibrierung nach Zhang dar. Statt eines Kalibrierkörpers wird schlicht ein planares Muster mit markanten Punkten verwendet. Oft reicht bereits ein Ausdruck eines schwarz-weißen Schachbrettmusters (engl.: Checkerboard) auf einem DIN-A4-Blatt aus. Da es sich hierbei um eine projektive Transformation von 2D nach 2D-Koordinaten handelt, sind mehrere Bildaufnahmen aus unterschiedlichen Kameraperspektiven notwendig, um genügend Gleichungen zur Bestimmung der Unbekannten zu erhalten [20].

Die bisherigen Aufführungen unterliegen der Annahme eines rein linearen Zusammenhangs zwischen den realen Punkten und deren Abbilder im Kamerabild. Dieses Vorgehen ist bei Kameras mit normalem oder kleinem FoV auch zulässig [25]. Besonders bei Weitwinkelkameras oder Kameras mit Fischaugenlinse (engl. fish eye lens) mit einem Sichtwinkel bis 190° sind nach Süße (2014) Nichtlinearitäten bei dem Abbildungsprozess zu berücksichtigen [20]. Sie sind bekannt als Verzeichnungen (engl.: distortion) und lassen sich in rotationssymmetrisch und tangentialsymmetrisch unterteilen [ebd]. Eine Übersicht der möglichen Verzeichnungserscheinungen ist der Abbildung 2.3 zu entnehmen.

Die Idee hinter der Modellierung der Nichtlinearitäten besteht darin, den radialen Abstand der Bildpunkte zum Bildmittelpunkt im verzerrten Zustand (x_d, y_d) in Abhängigkeit

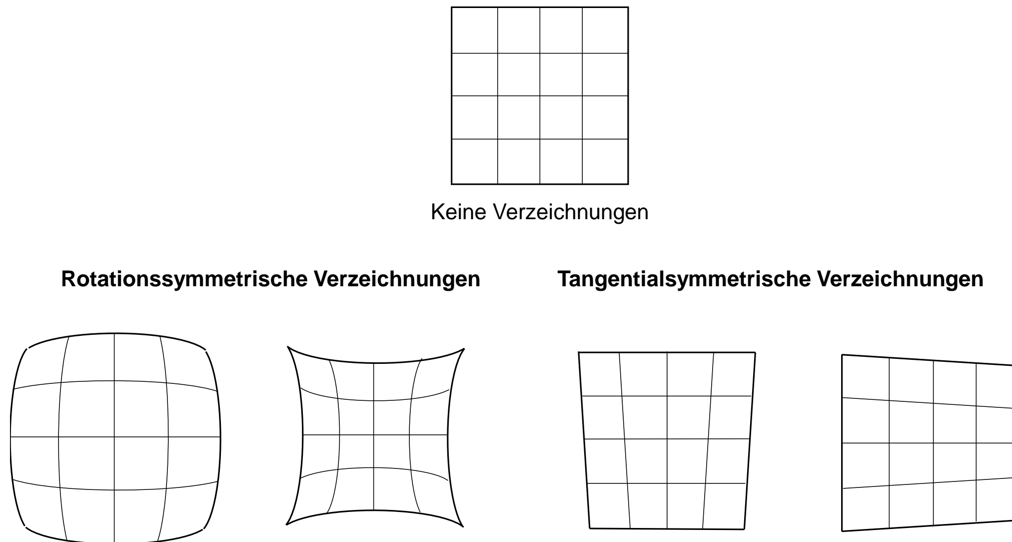


Abb. 2.3: Mögliche Nichtlinearitäten (Verzeichnungen) bei Kamerabildern [eigene Darstellung]

von der Größe des unverzerrten Bildes (x, y) in einer nichtlinearen Funktion abzubilden. Diese Vorgehensweise eignet sich vor allem für die häufig auftretenden rotationssymmetrischen Verzeichnungen. Geschieht dies über einen Polynomansatz, so ist man mit dem Problem konfrontiert, dass sich die Gleichung nicht analytisch auflösen lässt, um das gewünschte Ergebnis aus der Kalibrierung - ein Bild ohne Verzeichnungen - zu erhalten [25]. Aus diesem Grund wurden in der Literatur bereits Näherungsfunktionen und auch Modelle, welche ohne Polynomfunktion arbeiten, vorgestellt [ebd][23]. Neben der eher aufwendigen Korrektur von Nichtlinearitäten über Linsenaufsätze lassen sich Verzeichnungen auch über bereits standardisierte Bildverarbeitungsalgorithmen bereinigen. Da in der Computer Vision Bibliothek *OpenCV* abgeschlossene Funktionen zur Kalibrierung einer Kamera unter Berücksichtigung von nichtlinearem Verhalten zur Verfügung stehen, wird an dieser Stelle auch nur auf das hierfür verwendete Modell für die Verzeichnungen eingegangen [26]. In den folgenden Formeln bezeichnet *r* den euklidischen Abstand zwischen dem verzeichneten Bildpunkt und dem Zentrum der Verzeichnung.

Die radialsymmetrischen Verzeichnungen werden demnach beschrieben durch:

$$x_d = x \cdot (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.8}$$
$$y_d = y \cdot (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.9}$$

Dagegen sind die tangentialsymmetrischen Verzeichnungen definiert mit den Gleichungen:

$$x_d = x + [2p_1 xy + p_2(r^2 + 2x^2)] \tag{2.10}$$
$$y_d = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \tag{2.11}$$

Berücksichtigt man nun bei dem vorgestellten linearen Abbildungsprozess nach dem Zentralprojektionsmodell ebenfalls die Formeln zur Formulierung der möglichen Verzeichnun-

gen mit ihren fünf Parametern (k_1, k_2, p_1, p_2, k_3), so erhöht sich die Summe an zahlenmäßig zu bestimmender Parameter bei der Kamerakalibrierung auf mindestens 15. Mithilfe der Funktionsbibliothek in *OpenCV* lassen sich diese hier vorgestellten Kameramatrizen als auch die Koeffizienten der nichtlinearen Verzeichnungsfunktionen über die Kalibrierung mit einem Schachbrettmuster schnell offenlegen [26]. Jedoch ist zu beachten, dass bei der Korrektur der Nichtlinearitäten ein Upsampling stattfindet und somit ungewolltes Rauschen entstehen kann, was den weiteren Bildverarbeitungsprozess möglicherweise stört [27].

Zuletzt ist eine kurze Diskussion bezüglich der Tiefenbestimmung bei monokularen Kameras zu führen. Die digitalen Abbilder der Realität enthalten keine direkten Tiefeninformationen, wie zum Beispiel die 3D-Punktewolke eines LiDAR-Sensors. Da die exakte mathematische Bestimmung der Modellparameter einer Digitalkamera über eine Kalibrierung oft sehr aufwendig und störanfällig ist, existieren in der Praxis viele Ansätze zur Annäherung des fehlenden Objektattributs, sogenannte 3D-Rekonstruktionen [20, 28]. Diese Algorithmen nutzen jedoch meist ein a-priori Wissen über die Geometrie des zu rekonstruierenden Objekts oder andere Hinweise im Kamerabild und sind somit auf spezielle Anwendungsfälle beschränkt [20, 22]. Ein weit verbreitetes Vorgehen zur Gewinnung von Tiefeninformationen mit einer monokularen Kamera ist bekannt als "Structure from Motion (SFM)" [27]. Über die fest definierte Kamerabewegung werden Referenzpunkte in der 3D-Welt getrackt und durch Selbstkalibrierung und Triangulation lassen sich Rückschlüsse auf die dritte Dimension der Bildpunkte in der realen Welt ziehen [ebd]. Werden Stereokameras als eine Anordnung zweier Einzelkameras mit festem Abstand eingesetzt, wird die dritte Dimension über die Parallaxe (oder auch als Disparität bezeichnet) eingefangen. Auch der menschliche Sehapparat nutzt diesen Stereoaufbau, um dreidimensionales Sehen zu ermöglichen. Die Implementierung von Stereokameras im Automobil verlangt allerdings hohe Rechenkapazitäten zur Verknüpfung der parallelen Videodatenströme und verursacht dadurch einen größeren Leistungsverbrauch im Vergleich zu den kostengünstigen Einzelkameras [29]. Um ein möglichst ressourcenschonendes und effizientes kamerabasiertes System zu entwickeln, sollte an erster Stelle die Eignungsprüfung von monokularen Kameras für den spezifizierten Anwendungsfall stehen. Dementsprechend enthält dieses Kapitel bewusst keine tiefergreifenden Ausführungen zur Stereokameratechnik.

2.3 Maschinelles Lernen

Der vorangegangene Abschnitt verdeutlicht, dass sich Digitalkameras sehr gut dazu eigenen, Datenströme mit hohem Informationsgehalt zu erzeugen. Bereits bei dem Standard HD-Format, auch bekannt als 720p, mit einer Auflösung von 720 x 1280 Pixeln und einer angenommenen Farbtiefe von 8 Bit pro Farbkanal (entspricht genau einem Byte) beläuft sich die Dateigröße auf 2,7 MB (= 720 x 1280 x 8 Bit x 3 / 1024 KB/MB). Wird nun eine Aktualisierungsrate von 30 FPS gefordert, muss innerhalb einer Sekunde eine Datenmenge von 81 MB weitergeleitet, verarbeitet und gegebenenfalls gespeichert werden. In nur einer Stunde erzeugt eine solche nicht hochauflösende Farbbildkamera knapp 4,86 GB an Rohdaten.

Die Kunst besteht darin, aus diesen enormen Datenmengen möglichst effizient und mit hoher Ausbeute brauchbares Wissen zu gewinnen. Genau an dieser Stelle setzt das

Maschinelle Lernen (ML) an. Aus der Datenflut, welche dem Automatisierungssystem stetig zur Verfügung gestellt wird, sind selbstständig Zusammenhänge zu finden ohne Vorgabe eines genauen Lösungswegs durch vorherige Programmierung eines Softwareentwicklers [12]. Die Maschine arbeitet somit nicht mehr nur stur einen vorgegebenen Programmcode ab, sondern besitzt in begrenztem Umfang einen gewissen Spielraum, wo dessen künstliche Intelligenz gefragt ist. In der Literatur wird das Maschinelle Lernen oft der schwachen Künstlichen Intelligenz (KI) zugeordnet und ist dementsprechend als Teilgebiet des großen Schlagwortes *Künstliche Intelligenz* zu sehen [30]. Im Gegensatz zur starken KI, welche die Imitierung des menschlichen Bewusstseins fokussiert, liegt der Anwendungsbereich von ML-Algorithmen bei der Lösung von eindeutig definierten Aufgaben überwiegend in einem technischen Kontext [ebd]. Betrachtet aus einer anderen Perspektive kann ML ebenso als Werkzeug in der Datenanalyse klassifiziert werden [31].

Auf die reinen mathematischen Hintergründe reduziert, wird eine parametrisierte Funktion gesucht, welche den Elementen aus einer Menge A die entsprechenden Werte des Lösungsraums als Menge B zuordnet [31]. Anhand dieser Definition lässt sich vor dem Loslassen eines Algorithmus auf die zu verarbeitenden Daten die essentielle Fragestellung klären, ob es sich bei der gewünschten Anwendung um ein Klassifizierungs- oder ein Regressionsproblem handelt. Den Unterschied stellen die Ergebnismengen dar. Wird eine Klassifizierung erwartet, dann muss der Lösungsraum des Algorithmus diskret entworfen werden, sodass ein eindeutiges Ergebnis (die Klassenzuordnung) zur Verfügung steht. Bei der Regression werden kontinuierliche Werte als Ausgangsgrößen erwartet. Die Menge B ist somit eine Teilmenge der reellen Zahlen $\mathbb{R}^n$ [31]. Generell sollte von den ML-Algorithmen nicht die eine perfekte Lösungsfunktion gefordert werden, da immer von einer unvollkommenen Datenmenge auszugehen ist. Das Ziel besteht darin, die beste Lösungsfunktion für den Anwendungsfall durch Approximation zu finden, damit der Fehler (Abweichung vom erwarteten Sollwert) für alle Ausgabewerte betragsmäßig möglichst klein wird.

Als Spezialgebiet des ML ist das Deep Learning (DL) aufzuführen. Ihren Durchbruch feierten die Deep Learning-Methoden im Zusammenhang mit der wachsenden Relevanz der Bildverarbeitung bei der visuellen Wahrnehmung (Perception) der Umgebung eines autonom fahrenden Fahrzeugs als auch in anderen Robotik- oder Computeranwendungen. Bei großen und vor allem höherdimensionalen Daten stoßen klassische ML-Algorithmen an ihre Leistungsgrenze [12]. Diese Sättigung weisen neuronale Netze nicht auf, weshalb sie sich hervorragend zur automatisierten Extraktion von Informationen bei umfangreichen, unstrukturierten und mehrdimensionalen Datensammlungen eignen und sich innerhalb des letzten Jahrzehnts als Stand der Technik bei der Umfeld- oder Objekterkennung im autonomen Fahren etabliert haben. Diese Abgrenzung zwischen den konventionellen ML-Ansätzen und den jüngeren Deep Learning-Lösungen durch neuronale Netze verdeutlicht Abbildung 2.4.
Die Grundelemente eines neuronalen Netzes stellen die Neuronen (auch Knoten genannt) mit den Kanten als Verbindungen zwischen den einzelnen Knoten einer jeden Schicht (engl.: Layer) dar. Die Anordnung dieser Neuronen folgt einem Schichtaufbau, wobei jedes Netz mindestens eine Ein- und Ausgabeschicht besitzt mit einer jeweiligen Anzahl an Neuronen abgestimmt auf die Größe des In- bzw. Outputs. Dazwischen können unbegrenzt weitere Layer mit einer beliebigen Menge an Neuronen platziert werden. Je mehr solcher unzugänglicher Hidden Layers in einem neuronalen Netz zu finden sind, desto “tiefer” ist dieses und desto komplexere Funktionen können durch das Modell an-

genähert werden. Jene Netze mit mehr als einem Hidden Layer fallen entsprechend ihrer Tiefe unter den Begriff "Deep Learning".

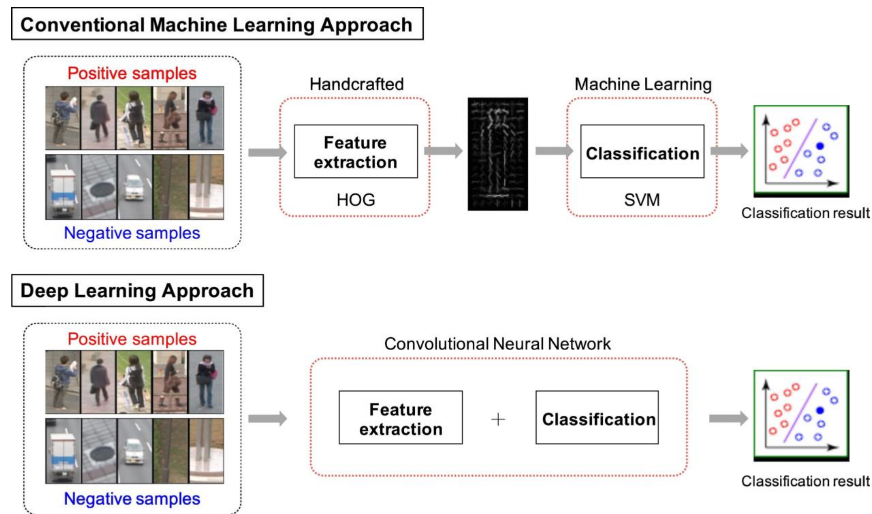


Abb. 2.4: Gegenüberstellung des klassischen Machine Learning und des jüngeren Deep Learning-Ansatzes zur Bildklassifizierung [6]

Da im Rahmen dieser Arbeit ein kamerabasierter Ansatz zur Automatisierung des Rangiermanövers verfolgt wird, ist der Fokus auf die speziell für die Bild- und Audioverarbeitung entworfenen *Convolutional Neural Networks (CNN)* zu legen. Im Folgenden werden einige wichtige Bestandteile solcher Modelle ohne Anspruch auf Vollständigkeit aufgezeigt. Neben einer kurzen Beschreibung eines jeden Bausteins fasst Abbildung 2.5 diese schließlich zusammen und bietet eine Visualisierung der theoretischen Ausführungen. Für detailliertere Angaben zu allen möglichen Bestandteilen eines neuronalen Netzes wird auf die Dokumentation der zwei beliebten Python-Bibliotheken - Pytorch und Keras - zur Programmierung von DL-Anwendungen verwiesen [32, 33]. Es ist zu beachten, dass die hier verwendeten Bezeichnungen an die Pytorch-Nomenklatur angelehnt sind.

Convolution Layer:
Dieses essentielle Element eines jeden CNNs stellt die Grundlage der automatisierten Extraktion von Bildmerkmalen (z.B. Kanten, Farbübergänge, geometrische Formen, etc.) dar. Der englische Ausdruck "Convolution" ist auf deutsch mit der mathematischen Operation der Faltung gleichzusetzen. Eine Faltungsoperation auf Bildern wirkt wie ein lineares Filter mit der sogenannten "Feature Map" als Ergebnis [20]. Hierfür wird die Größe und Anzahl der Kernel H als Filter festgelegt und mit einer definierten Schrittgröße (engl.: Stride) über die zweidimensionale Bildebene geführt. Anwendung findet bei jedem betrachteten Pixel Formel 2.12 mit u und v als Pixelkoordinaten der zweidimensionalen Bildmatrix I. Trainierbare, variable Parameter eines Convolution Layers sind die Zelleninhalte der Kernel. Bei einem Kernel der Größe 3 x 3 resultieren neun anpassbare Parameter. In Abbildung 2.5 ist ein Kernel mit typischen Werten zum Auffinden von senkrechten Kanten abgebildet. Je mehr solcher Layer in Reihe geschaltet werden, desto komplexere und oft auch zusammengesetzte Merkmale können extrahiert und den weiteren Verarbeitungsschritten bereitgestellt werden.

$$(I * H)(u,v) = \sum_{j=-n}^{n} \left(\sum_{i=-n}^{n} I(u+i, v+j) \cdot H(n+i, n+j) \right) \tag{2.12}$$

$$\text{mit } H \in \mathbb{R}^{2n+1 \times 2n+1}$$

Pooling Layer:
Sie dienen hauptsächlich der Auflösungsreduzierung in einem neuronalen Netz, um die Anzahl der durchzuführenden Rechenoperationen und die Größe des Modells einzugrenzen. Weitläufig bekannte Methoden sind die Auswahl des höchsten Pixelwerts in einem festgelegten Bildbereich (Max. Pooling) oder die Mittelung der Werte (Average Pooling) in diesem Bereich. Wird zum Beispiel bei einer 3 x 3-Bildmatrix ein 2 x 2-Max. Pooling mit der Schrittweite von einem Pixel durchgeführt, entsteht ein neues Bild mit einer Auflösung von 2 x 2 Pixeln.

Flatten:
Wie bereits aufgeführt sind neuronale Netze in der Lage mehrdimensionale Eingangsdaten zu verarbeiten. Häufig fordert der Anwendungsfall jedoch eindimensional organisierte Ausgabewerte. Eine klassische Ausnahme stellen Modelle zur Bildsegmentierung dar. Von ihnen wird nicht ein Zeilenvektor sondern eine Bildmaske mit selbiger Höhe und Breite des Originalbildes zur Zuordnung/Klassifizierung eines jeden Pixels als Output erwartet.
Der Flatten-Befehl reiht schlicht alle Zeilen einer 2D-Matrix in einem einzigen Zeilenvektor aneinander.

Fully Connected Layer (FCL):
In der Pytorch-Umgebung wird für die Initialisierung eines Fully Connected Layers eine lineare Schicht genutzt, welche die Linearisierung des Inputs eines jeden Neurons abbildet. Unter Keras ist dieses Modul als “Dense Layer” geführt. Diese Art von Layer ist meist in den abschließenden Schichten eines CNN zu finden. Sie führen auf Basis der vorliegenden Merkmale, welche in dem Bild zu finden waren, die eigentliche Regression oder Klassifizierung durch und werden deshalb auch häufig als “Detection Head” referenziert. FCLs zeichnen sich dadurch aus, dass jeder Knoten einer Schicht eine Verbindung zu jedem Neuron der nächsten Schicht besitzt. Die Kantengewichte, welche ausdrücken, wie stark die Verbindung zwischen zwei Neuronen gewichtet wird, können ebenfalls in Matrixschreibweise festgehalten werden. Sie formen die trainierbaren Parameter eines FCLs. Bei vier Eingängen und drei Ausgängen ergibt sich eine 3 x 4-Matrix der Kantengewichte und somit 12 einstellbare Parameter.
An dieser Stelle soll kurz auf das Prinzip hinter der Berechnung einer Knotenverbindung anhand des einfachen Perzeptrons, bestehend aus zwei Neuronen E1 und E2 in der Eingabeschicht, ein Neuron A als Ausgabe und den beiden Kantengewichten w_{A1} und w_{A2} eingegangen werden. Der Ausgabewert berechnet sich über eine gewichtete Linearkombination der Eingänge, wobei das Ergebnis noch gezielt durch Definition der Aktivierungsfunktion f manipuliert bzw. angepasst werden kann:

$$A = f\left(E1 \cdot w_{A1} + E2 \cdot w_{A2}\right) \tag{2.13}$$

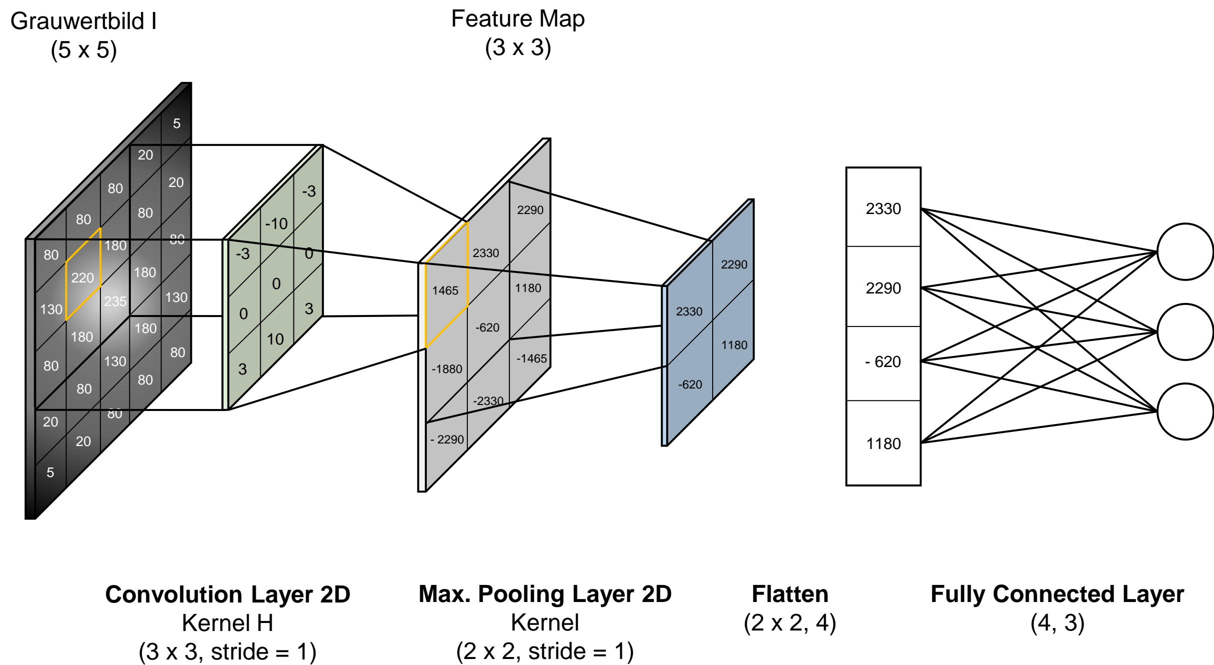


Abb. 2.5: Ausgewählte Grundelemente eines Convolutional Neural Network (CNN)[eigene Darstellung]

Weitere Module:
Ebenfalls zu erwähnen sind die Dropout Layer, welche in der Trainingsphase das Risiko des Überhitzens eines Pfades mindern. Durch die Vorgabe eines reellen Zahlenwertes zwischen Null und Eins wird festgelegt, wie hoch die Wahrscheinlichkeit ist, dass während des Trainings zufällig Verbindungen zwischen zwei Neuronen abgeschnitten werden, indem beliebige Elemente des Layer-Inputs auf Basis einer Bernoulli-Verteilung zu Null gesetzt werden. Durch die gezielte Einschränkung des Netzes während des Trainings erfolgt eine gleichmäßigere Ausnutzung aller verfügbaren Pfade bei der Lösungsfindung.
Weiterhin seien namentlich Padding, Resizing, Normalization und Transformer Layer erwähnt. Sie werden jedoch nicht als essentielle Grundlage für das Verständnis der Arbeitsweise von CNNs gesehen.

Auf eine Übersicht der vorgefertigten linearen als auch nicht-linearen Aktivierungsfunktionen sowie der Loss-Funktionen zur Bestimmung des ausschlaggebenden Fehlers für die Optimierung der Modellparameter wird hier verzichtet. Auf die in dieser Arbeit verwendeten Funktionen bei der Definition eines Modells zur Ausführung der Perception Task wird im Hauptteil genauer eingegangen.
Zu erkennen ist, dass bei der Anwendung von CNNs zur Bildverarbeitung ebenfalls Matrizen- und Vektoroperationen dominieren und sie sich somit ideal zur Ausführung auf dafür zugeschnittene Grafikprozessoren eignen. Da bei mehrdimensionalen Größen die Tensoralgebra der Linearen Algebra vorzuziehen ist, werden meist alle beteiligten Größen als Tensoren an die GPU übergeben. Interessierten Lesern wird für tiefere Einblicke in

die Tensoralgebra die Lektüre von Süße, 2014 empfohlen [20, 415-448].

Nach der erfolgreichen Abgrenzung zwischen Machine Learning und Deep Learning und der Diskussion des prinzipiellen Aufbaus von CNNs stellt sich abschließend die Frage, wie solche Netzarchitekturen ihre künstliche Intelligenz erlernen. Die existierenden Lernmethoden lassen sich grob drei Kategorien zuordnen [31, 12]:

- Unüberwachtes Lernen (engl.: Unsupervised Learning)
- Überwachtes Lernen (engl.: Supervised Learning)
- Bestärkendes Lernen (engl.: Reinforcement Learning)

Der grundlegende Lernvorgang wird anhand des überwachten Lernens aufgezeigt, da laut Nolting (2021) 90 % aller Anwendungsfälle dieses Verfahren nutzen [12, S. 107]. Hierfür ist zunächst zu verstehen, dass ein neuronales Netz bidirektional durchlaufen werden kann. Befindet es sich nach dem Training in Anwendung mit einem festen Parametersatz wird das Netz nur vorwärts von Input zu Output durchgerechnet (engl.: forward propagation). Während des Trainings wird jedoch zur Aktualisierung der Parameter auch rückgerechnet (engl.: back propagation). Entscheidend für ein erfolgreiches Training ist der bereitgestellte und bei Bedarf durch Data Augmentation aufgewertete Datensatz. Er stellt später den Erfahrungsschatz des intelligenten Netzes dar, auf dem dessen Aussagen beruhen. Für jedes Bild in diesem Datensatz wird ein Label definiert, welches die gewünschten Ausgabewerte enthält. Dem Modell wird somit mitgeteilt, was es erkennen soll, aber nicht wie [6].
Typischerweise teilt sich eine Trainingsepoche in zwei Phasen auf, dem Optimieren der Modellparameter aufgrund der Vorgabe der Trainingsbilder und eine Validierung der aktualisierten Parameter mit einem kleineren, von dem ursprünglichen Datensatz abgespaltenen Testdatensatz. In der ersten Phase trifft das Netz zuerst seine Aussagen zu dem vorgegebenen Trainingsdatensatz, woraus sich aus der Abweichung zwischen den Soll- und Ist-Ausgabewerten unter der Zuhilfenahme einer vordefinierten Loss-Funktion der Fehler berechnen lässt. Über die rückwärtige Bestimmung der partiellen Ableitungen aller trainierbaren Parameter und das Gradientenabstiegsverfahren werden die Parameter identifiziert, welche entsprechend einer vorgegebenen Lernrate angepasst werden müssen, um schließlich ein globales Minimum des Fehlerraums zu finden. Diese Backpropagation wird durch vorgefertigte Optimizer-Module automatisch durchgeführt. Hier schließt sich auch der Kreis zu dem ursprünglichen Ziel des Machine Learnings. Durch Anpassung der variablen Parameter im Training soll das Modell mit möglichst geringem Fehler den Zusammenhang zwischen Ein- und Ausgang abbilden. Die zweite Phase dient lediglich der Beurteilung des Trainingsfortschritts nachdem alle Trainingsbilder einmal behandelt und die Parameter aktualisiert wurden.
Überwachtes Training von neuronalen Netzen benötigt nicht nur eine enorme und hochwertige Datensammlung, sondern ist zudem auch sehr rechen- und zeitintensiv je nach Anzahl der zu berücksichtigenden Modellparameter. Da das globale Minimum nur iterativ durch eine mehrmalige Abfolge von Training und Testen ermittelt werden kann, müssen oft viele Epochen durchlaufen werden bis eine klare Konvergenz der Lernkurve zu erkennen ist und ein weiteres Training keine merkliche Steigerung der Vorhersagegenauigkeit mehr erwarten lässt.
Um eine möglichst hohe Generalisierungsfähigkeit des DL-Modells zu erzielen, sollte das Training abgebrochen werden sobald Overfitting auftritt. Dieser Effekt äußert sich da-

durch, dass der Fehler während der Trainingsphasen weiter sinkt, aber sich gleichzeitig die Ergebnisse der Testphasen verschlechtern. Das Netz fängt an, die Trainingsbilder auswendig zu lernen und verliert dadurch seine Anpassungsfähigkeit an Situationen außerhalb des Trainings. Bestärkende Faktoren für ein solches Verhalten können unter anderem ein zu einseitiger oder zu kleiner Datensatz oder eine zu hohe Komplexität des Modells für den vorgesehenen Anwendungsfalll sein.

Eine gute Alternative zu dem aufwendigen Training eines Modells “from scratch” ist das Transfer Learning. Für viele bekannte DL-Modellarchitekturen sind Kantengewichte und Parametersätze trainiert mit öffentlich zugänglichen Datensätzen verfügbar. Bei CNNs kann das Transfer Learning im Wesentlichen auf zwei Arten umgesetzt werden. Einerseits lässt sich eine große Auswahl an Modellen zusammen mit vortrainierten Kantengewichten als Initialisierungsschritt aus den Python-Bibliotheken für Deep Learning in das eigene Projekt laden. Da nun bereits gute Startwerte für die zu optimierenden Parameter vorliegen, müssen deutlich weniger Epochen abgehandelt werden, bis das Optimum des Parametersatzes, abgestimmt auf den speziellen Anwendungsfall, gefunden wird. Weiterhin bietet sich die Möglichkeit an, nur die letzten Layer eines Netzes zu trainieren und somit die automatisierte Feature Extraction, welche hauptsächlich durch die Convolution Layer bewerkstelligt wird, als Backbone für das eigene Modelle aus einem vorherigen Training zu übernehmen und einzufrieren. Mit dem eigenen Trainingsdatensatz werden dann nur die letzten Schichten, meist in Form von FCLs, trainiert. Der Bedarf an Daten und die Rechenzeit vor allem bei sehr großen Modellen lassen sich mit diesem Vorgehen drastisch senken [31]. Im weiteren Verlauf dieser Arbeit wird das Transfer Learning aktiv angewendet und evaluiert.

Eine sehr ausführliche und viel detailreichere Übersicht zu den in diesem Kapitel behandelten Themen liefern Alzubaidi et. al. (2021) [34].

2.4 Miniaturisierter DIY-Fahrroboter “JetBot”

Hardware

Da nicht nur zur Entwicklung der automatisierten Fahrfunktion sondern auch für eine erste Funktionsvalidierung ein Fahrroboter als Rapid Prototyping-Tool eingesetzt wird, ist das vorhandene Hard- und Software-Setup offen zu legen. Bei der Beurteilung der Funktionalität und der Analyse der Übertragbarkeit der Ergebnisse auf das Realfahrzeug sind ebenfalls diese in der Entwicklungsphase vorhandenen Rahmenbedingungen und Einschränkungen zu berücksichtigen. Unter dem Namen “JetBot” verbirgt sich ein Open-Source-Baukasten aus dem Hause NVIDIA mit ausführlicher Dokumentation zur einfachen und schnellen Inbetriebnahme von KI- und IoT-Anwendungen. Die verbauten Komponenten entstammen der vorgeschlagenen Stückliste für die aktuelle Version v0.4.3 des Roboter-Setups [7]. Neben dieser Plattform ist eine Vielfalt an weiteren ähnlichen Aufbauten von Drittanbietern verfügbar [ebd]. Den Zusammenbau des miniaturisierten DIY-Fahrroboters verdeutlicht die Übersicht in Abbildung 2.6.

Angetrieben wird der Fahrroboter über die beiden 5V-Gleichstrommotoren, welche ihre Leistung an das jeweilige Vorderrad abgeben. Es ist keine Lenkung inbegriffen, da

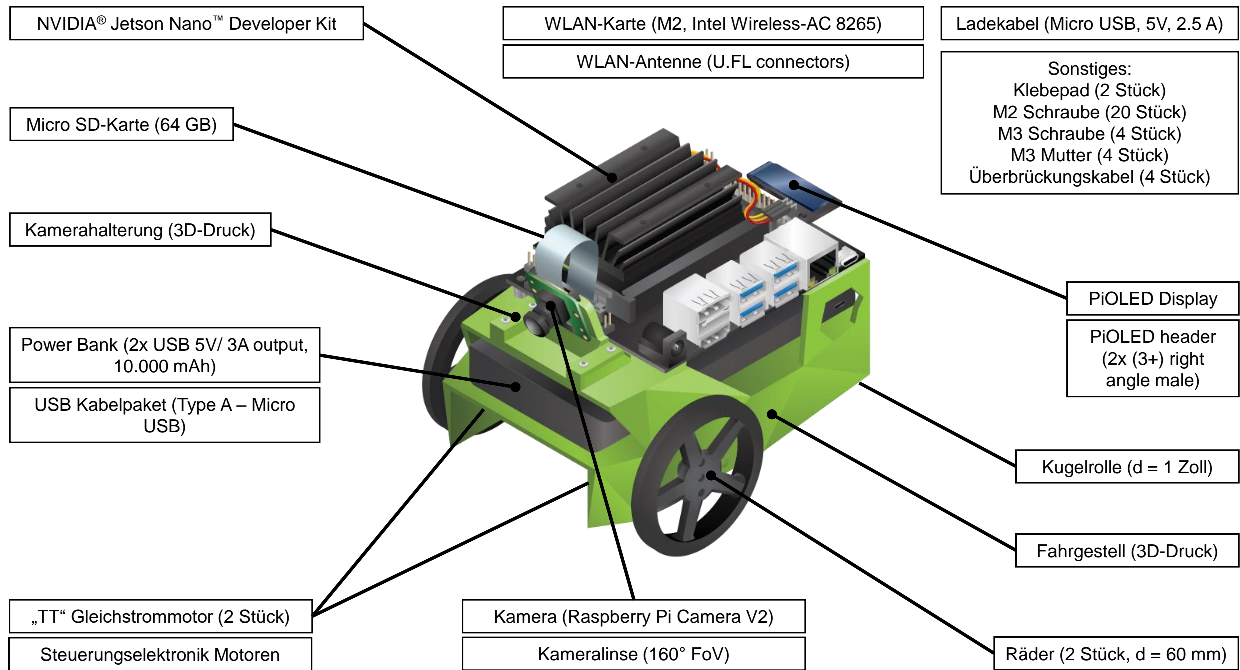


Abb. 2.6: Hardware-Setup DIY-Fahrroboter “Jetbot” v0.4.3 [7][eigene Darstellung]

die longitudinale als auch die laterale Fahrzeugbewegung über die individuelle Vorgabe der Drehzahl des E-Motors bestimmt wird. Als mobile Energiequelle dient ein Akkumulator mit einer Kapazität von 10.000 mAh und einer passenden Nennspannung von 5V, sodass keine Um- oder Wechselrichter notwendig sind. Der dritte Bodenkontakt, ausgebildet als bewegliche Plastikkugel, sorgt für eine Art Rollenfreilauf und die statische Bestimmtheit des Roboters. Als einziger exteroperceptiver Sensor dient eine monokulare Digitalkamera mit einem Weitwinkelobjektiv zur Erfassung des frontalen Umfelds als farbiger Videofilm. Weitere Daten werden nicht erhoben, da weder Odometriedaten über eine inertiale Messeinheit (engl.: Inertial measurement unit, IMU) bereitgestellt werden, noch anderweitige Sensorik eingesetzt wird. Für eine spätere Implementierung der automatisierten Fahrfunktion müssen weitaus mehr Datenschnittstellen vorgesehen werden, welche der Recheneinheit des Automatisierungssystems interne Fahrzeuggrößen vor allem des Antriebssystems (z.B. Raddrehzahl, Querbeschleunigung, Lenkwinkel, etc.) zur Regelung der Fahrdynamik bereitstellen und bei Bedarf auch eine Sensorfusion ermöglichen. Für die Untersuchung der Funktionsfähigkeit und Genauigkeit einer rein kamerabasierten Ausführung ist das vorliegende Hardware-Setup ausreichend.
Eine bedeutende Komponente stellt die WLAN-Karte zusammen mit der Antenne dar. Sie ist in Abbildung 2.6 nicht gekennzeichnet, da das WLAN-Modul unterhalb der Kühlrippen des Entwicklerboards platziert ist. Sind der JetBot sowie der Arbeitsrechner des Entwicklers mit dem gleichen WLAN-Netz verbunden, können über die webbasierte Entwicklungsumgebung *JupyterLab* Jupyter Notebooks (Dateiformat: .ipynb) mit darin enthaltenem Programmcode in der Hochsprache Python erstellt, bearbeitet und dank der Python-Programmierschnittstelle (engl.: Application Programming Interface, API) direkt ausgeführt werden. Erste Ideen oder einzelne Programmbauteile lassen sich schnell und unkompliziert in der Realität testen, debuggen und validieren ohne vorheriges Kompilieren in eine maschinennahe Programmiersprache und Flashen des Programmstands auf den Speicher eines Microcontrollers. Das Herzstück von JetBot umfasst das Entwick-

lerboard Jetson Nano™ von NVIDIA, welches bis zu 472 GFLOP/s[1] als eingebettetes System ausführt [35]. Eine Zusammenfassung der technischen Daten der 4GB-Variante des Jetson Nano™ ist in Tabelle 2.1 zu finden. Die 2GB-Variante wird von NVIDIA nicht mehr unterstützt.

Tabelle 2.1: Technische Daten des NVIDIA 4GB Jetson Nano™ [11]

GPU	NVIDIA Maxwell™-GPU mit 128 Kernen
CPU	Quad-core ARM® Cortex®-A57 MPCore-Prozessor
Arbeitsspeicher	4 GB, 64-Bit-LPDDR4
Datenspeicher	microSD (je nach verwendeter Speicherkarte, hier: 64 GB)
Videokodierung	1x 4K30 | 2x 1080p60 | 4x 1080p30 | 9x 720p30 (H.264/H.265)
Videodekodierung	1x 4K60 | 2x 4K30 | 4x 1080p60 | 8x 1080p30 | 18x 720p30 (H.264/H.265)
Netzwerk	Gigabit-Ethernet, M.2 Key E
Kamera	2x 15-polige 2-Lane MIPI CSI-2-Kameraanschlüsse
Anzeige	1 x HDMI 2.0, 1 x DP 1.2
USB	4x USB 3.0 Typ A-Anschlüsse 1x USB 2.0 Micro-B-Anschluss
Sonstige E/A	40-poliger Header (UART, SPI, I2S, I2C, PWM, GPIO) 12-poliger Header für Automatisierung 4-poliger Header für Lüfter 4-poliger Header für POE DC-Strombuchse Tasten “Power", “Force Recovery” und “Reset"
Mechanik	100 mm x 79 mm x 30,21 mm (Höhe umfasst Trägerplatine, Modul und Kühlung)

[1] FLOP/s: engl. Maßeinheit für die Leistungsfähigkeit eines Mikroprozessors, Floating Point Operations per second (zu deutsch: Gleitkomma-Operationen pro Sekunde)

Software
Neben dem Baukasten für das Hardware-Setup stellt NVIDIA ebenfalls ein Software Entwicklungskit (engl.: Software Development Kit, SDK) namens JetPack bereit. Auf dem JetBot v0.4.3 mit dem Jetson Nano Entwicklerkit kann zu diesem Zeitpunkt (Stand: Dezember 2022) als aktuellste Version JetPack v4.5, veröffentlicht zum Jahresanfang 2021, in Betrieb genommen werden [neueste Version: JetPack 5.0.2, veröffentlicht am 16.08.2022] [7][36]. Anzumerken ist ebenfalls, dass auf dem JetBot nicht die aktuelleste Version von Pytorch zur Programmierung neuronaler Netze in Python einsetzbar ist. Neben Python 3.6 steht Torchvision 1.8 auf dem Jetson Nano zur Verfügung. Auf die Frameworks und Bibliotheken zur Entwicklung von Deep Learning Modellen wurde im vorangegangenen Kapitel 2.3 bereits eingegangen. Folgende Software ist auf dem JetBot verfügbar, nachdem das JetPack v4.5 Image auf eine Speicherkarte geflasht und diese eingesetzt wurde. Für eine detailliertere Übersicht wird auf die betreffende Internetseite von NVIDIA verwiesen [37].

- Linux-Treiberpaket von NVIDIA (NVIDIA L4T 32.5)
- TensorRT 7.1.3: Optimierung der Laufzeit von Deep Learning-Modellen
- cuDNN 8.0: Bibliothek “CUDA Deep Neural Network” für die effiziente Implementierung von standardmäßigen Routinen bei der Programmierung von neuronalen Netzen
- CUDA 10.2: Toolkit für die Beschleunigung von Anwendungen auf der GPU
- Multimedia API: Bibliotheken für die Kameraschnittstelle bzw. Sensortreiber
- Computer Vision: Paket, geschnürt aus Vision Programing Interface (VPI) 1.0, OpenCV 4.1.1 und Visionworks 1.6
- Entwicklertools: NVIDIA Nsight Systems 2020.5 und NVIDIA Nsight Graphics 2020.5

Da es sich bei dem JetPack SDK um ein abgestimmtes und optimiertes Softwarepaket handelt, sollten möglichst Änderungen durch Updates oder Neuinstallationen einzelner Anwendungen vermieden werden. Dies führt sonst schnell zu Kompatibilitätsproblemen zwischen den einzelnen Komponenten und eine eingeschränkte Funktionalität des JetBots. Ist bereits eine SD-Karte mit Image vorhanden, kann das Software Setup auch über den Download von vorgefertigten sogenannten “Docker Containers” erfolgen [7].

Bei erstmaliger Inbetriebnahme des JetBots muss einmalig eine Verbindung mit dem WLAN hergestellt werden. Hierfür sind Ein- und Ausgabegeräte, mindestens ein Monitor und eine Tastatur, über die USB-Schnittstellen anzuschließen und der JetBot mit Strom zu versorgen. Es sollte automatisch ein Boot stattfinden. In JetPack v4.5 ist keine grafische Ubuntu-Benutzeroberfläche (Ubuntu GUI) mehr verfügbar, sodass Eingaben, wie zum Beispiel die Herstellung der WLAN-Verbindung, über das Terminal und die entsprechenden Befehle abgewickelt werden. Nach erfolgreichem Abschluss des Hard- und Softwaresetups kann auf dem PiOLED Display die IP-Adresse abgelesen werden, über welche der Zugriff auf die Entwicklungsumgebung *JupyterLab* erfolgt. Voraussetzung hierfür ist, dass alle verwendeten Geräte als auch der JetBot sich innerhalb der Reichweite des gleichen WLAN-Netzes befinden, wodurch der Einsatzbereich des Fahrroboters stark eingeschränkt wird - zumeist auf Büroumgebungen mit ausreichend guter WLAN-Verbindung.

Nicht zuletzt ist die wohl wichtigste Limitation bei der Anwendung des Fahrroboters zur Funktionsvalidierung aufzuführen - die Begrenzung der Leistungsaufnahme. Aufgrund der eingesetzten Power Bank mit einer maximalen Stromabgabe von 3 A über die USB-Kabel ist der kleinere von den beiden einstellbaren Power Modes (5 W und 10 W) zu wählen. Dadurch kann zusätzlich sichergestellt werden, dass ein Großteil der Verlustleistung in Form von Wärme über die Kühlrippen erfolgreich abgeführt werden kann und einer Überhitzung vorgebeugt wird. Jedoch wird die Performance deutlich verringert, da nur zwei von den verbauten vier CPU-Kernen in diesem Modus aktiv sind und auch sonst die allgemeine Rechenleistung (CPU und GPU zusammen) gedrosselt wird.

Um vor allem das rechenintensive Training von neuronalen Netzen zu beschleunigen, ist hierfür ein stationärer PC mit Windows-Betriebssystem und der Grafikkarte NVIDIA T1000, einem DualCore Intel-Prozessor der 11. Generation mit 32 GB Arbeitsspeicher und den neuesten Versionen von Python, CUDA, OpenCV und den Pytorch-Paketen (Stand: Oktober 2022) vorgesehen. Die verbaute GPU besitzt 896 CUDA-Kerne, 4 GB Speicher und ist erst bei einer Leistungsaufnahme von über 50 W abgeriegelt [38].

3 Stand der Technik

Vor der Abhandlung des Hauptteils werden aktuelle Entwicklungen im Bereich der Umfelderfassung für autonomes Fahrens zusammengefasst. Insbesondere liegt der Fokus dieses Kapitels auf bekannten Lösungsansätzen oder auch bereits im Einsatz befindlichen Softwarefunktionen zur Erfüllung vergleichbarer Fahraufgaben, welche als Referenz oder Ausgangslage für die im Rahmen dieser Arbeit zu entwickelnde automatisierte Rangierfunktion dienlich sind.

3.1 Exteroperceptive Sensorik

Aus der Beobachtung der bisher vorgestellten Fahrzeugprototypen für autonomes Fahren lässt sich in der Automobilindustrie ein klarer Trend hinsichtlich der Sensorauswahl und -anordnung zur Umfelderfassung ableiten [5]. Um die Sicherheit eines autonomen Fahrzeugs im Straßenverkehr zu gewährleisten, wird hierbei auf Redundanz oder Prinzipverschiedenheit bei der eingesetzten Sensorik geachtet. Perception ist die kritischste Softwareschicht, da auf ihr die Entscheidungsfindung in einem autonomen System basiert [5, 18]. Zur Bereitstellung der Rohdaten für die Perception-Task haben sich in der Automobilindustrie hauptsächlich drei Sensortypen mit ihrer jeweiligen Eignung für bestimmte Anwendungsfälle herauskristallisiert [39, 40]. Sie werden in einem Multi-Sensor-Ansatz in unterschiedlichsten Ausführungen und Stückzahlen kombiniert, um ein optimales Ergebnis bei der Umfelderfassung unter allen Bedingungen zu erhalten.

Kamera: Farbbild-, Graubild-, als auch Wärmebildkameras (IR-Kameras) zählen zu den passiven und optischen Sensoren. Sie werden je nach ausgewähltem Sichtfeld und erforderlicher Auflösung für die Objekterkennung und Analyse der Fahrbahn eingesetzt [5]. Bei Tageslicht sind Kameras, welche elektromagnetische Wellen im sichtbaren Wellenlängenbereich einfangen, zu bevorzugen, wohingegen Nachtfahrten oder Situationen mit spärlicher Beleuchtung durch IR-Kameras abgedeckt werden [41]. Spezifische Einsatzzwecke sind laut Liu et al. (2019) unter anderem die Erkennung von Fahrbahnmarkierungen, Ampeln und Fußgängern in Echtzeit [42]. Üblicherweise werden mindestens acht Kameras über das Fahrzeug verteilt verbaut [ebd]. Falls die zweidimensionalen Daten einer einzelnen Kamera nicht genügen und insbesondere die Abstandsinformationen entscheidend sind, finden Stereo-Kameras als kostengünstigeres Pendant zu den nachfolgend vorgestellten LiDAR-Sensoren Anwendung [18].

LiDAR: Unter diesem Akronym verbirgt sich die englische Bezeichnung ***Li**ght **D**etection **A**nd **R**anging*. LiDAR-Sensoren senden Laserstrahlen im nicht sichtbaren Wellenlängenbereich aus und empfangen die von Objekten reflektierten Lichtimpulse wieder, wodurch sie als aktive optische Sensoren zu klassifizieren sind. Über die Laufzeitmessung der Signale kann nicht nur der Abstand zu einem Objekt, sondern auch indirekt die relative Geschwindigkeit eines erfassten Gegenstands bestimmt werden [42]. Die 3D-Punktwolke dieser Sensoren wird hauptsächlich zur Erstellung einer digitalen Karte der Umgebung, zur Lokalisierung des Ego-Fahrzeugs und zur Kollisionsvermeidung genutzt [ebd][18].

Radar: Das Messprinzip eines Radar-Sensors ist vergleichbar mit den LiDAR-Sensoren mit der Ausnahme, dass mit elektromagnetischen Wellen im GHz-Frequenzbereich (Ra-

diowellen) gearbeitet wird. Anhand frequenzmodulierter Signale erfolgt nicht nur die Objekterkennung, sondern auch eine direkte Erfassung von Abständen und relativen Geschwindigkeiten durch Ausnutzung des Doppler-Effekts. Es ist zu beachten, dass Radar-Sensoren einen Trade-Off zwischen Reichweite und horizontalem Öffnungswinkel besitzen und sie sich somit in Fernfeld- und Nahfeld-Radare untergliedern [18, 5]. Da die bereitgestellten Daten keiner aufwendigen Aufbereitung und Verarbeitung bedürfen, werden sie in einem autonomen Fahrzeug meist an der Front- und dem Heck verbaut, um eine zusätzliche Absicherung der Kollisionsvermeidung zu gewährleisten [42].
Ergänzt wird das mögliche Sensor-Setup eines autonomen Fahrzeugs durch weitere Sensoren, wie zum Beispiel:

- Global Navigation Satellite System (GNSS) zur absoluten Positionsbestimmung des Ego-Fahrzeugs [42]
- Ultraschall-Sensoren zur Nahfeldüberwachung
- Inertial Measurement Unit (IMU) zur Erfassung der Odometriedaten [42]
- u.v.m

Für die Lokalisierung des Ego-Fahrzeugs innerhalb seiner Umgebung sind hochaufgelöste digitale Karten (engl.: HD maps) erforderlich, um die kombinierten Sensorsignale mit dem virtuellen Umweltmodell unter Zuhilfenahme eines Partikelfilters abzugleichen [42]. Alternativ kann diese Task über die Methode des Simultaneous Localization And Mapping (SLAM) erfüllt werden und somit auf vorab erstellte Karten verzichtet werden [43, S.97].

Im Gegensatz zu führenden Herstellern von autonomen Fahrzeugsystemen, wie das aus Google hervorgegangene Unternehmen Waymo (Kalifornien, USA) oder die NVIDIA Corp. (Kalifornien, USA), verfolgt der Autohersteller Tesla, Inc. (Texas, USA) einen disruptiven Technologieansatz. Mit dem System "Tesla Vision" wird eine rein kamerabasierte Perception bei autonomen Fahrzeugen forciert, wodurch nicht nur Kosten sondern auch komplexe und rechenintensive Sensorfusion-Algorithmen eingespart werden sollen [8]. Aus der ursprünglichen Sensor Suite für den Tesla Auto Pilot (SAE Level 2) bestehend aus Kameras, Radar und Ultraschallsensoren wurde zuerst der Radar entfernt und zusammen mit der Einführung der Beta-Version der Software für autonomes Fahren ("Fully Self Driving", FSD) das Ende der Ultraschallsensoren im Model 3 und Model Y im Jahr 2021, gefolgt von Model S und X im Jahr 2023 verkündet [44, 45]. Vor allem durch den Entfall der HD Maps zur Positionsbestimmung wird eine höhere Generalisierungsfähigkeit des Umfelderkennungssystems erwartet [8].
In Abbildung 3.1 werden die in diesem Kapitel thematisierten Ansätze zur digitalen Erfassung der Fahrzeugumgebung gegenübergestellt.

Jedoch kursieren seit Anfang Dezember 2022 im Internet Gerüchte, dass Tesla entgegen der bisher verfolgten Strategie in zukünftigen Modellen wieder Radar-Sensoren verbauen würde. Demnach habe das Unternehmen "der US-Funkbehörde FCC mitgeteilt, dass ab Mitte Januar 2023 ein Fahrzeug mit einer neuen Radar-Technologie vermarktet werden soll" [46].

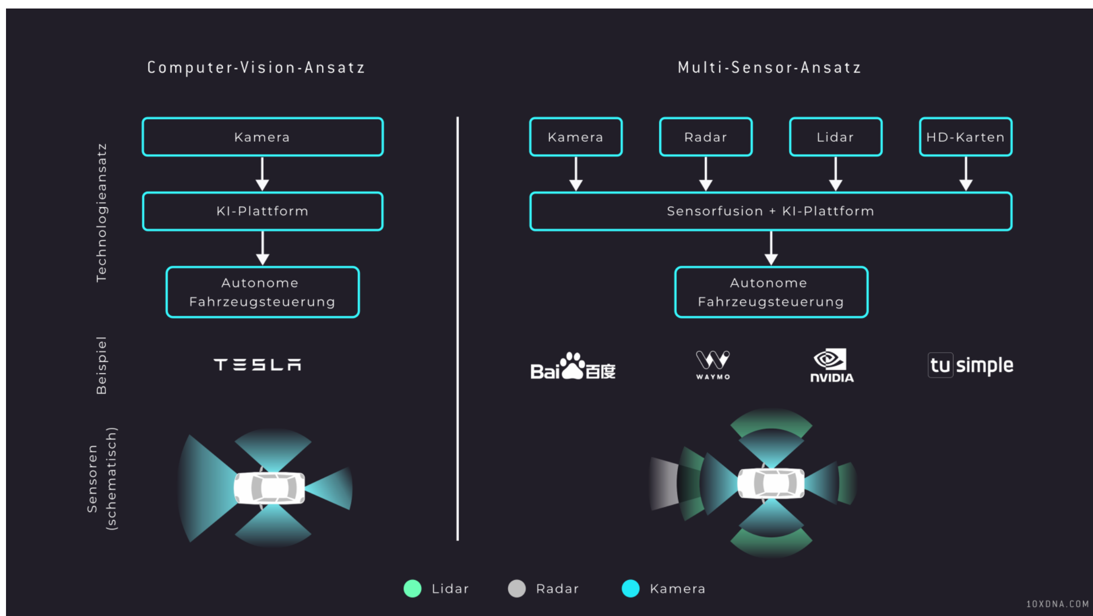


Abb. 3.1: Gegenüberstellung unterschiedlicher Ansätze zur Umfelderkennung [8]

3.2 Prinzipverwandte automatisierte Fahrfunktionen

Dieses Kapitel bietet eine Übersicht zu verwandten Arbeiten, Entwicklungs- und Forschungsergebnisse eingegrenzt auf Anwendungsfälle mit ähnlichen wie in dieser Abhandlung vorhandenen Rahmenbedingungen und Problemstellungen. Auf Basis dieser Literaturrecherche erfolgt schließlich die Entwicklung der automatisierten Rangierfunktion für autonomes Fahren.

3.2.1 Spurhalteassistenz und Lane Following

Als Fahrerassistenzsysteme finden Spurhalte- und Spurwechselassistenten in Ausführung als aktives (eingreifendes) oder passives (warnendes) System seit Jahren bereits erfolgreich Anwendung in Serienfahrzeugen [47, S. 62-63]. Sie mindern das Risiko des unbeabsichtigten Verlassens der eigenen Fahrspur oder unterstützen den Fahrer bei der Durchführung eines Fahrspurwechsels. Bei einem autonom agierenden Fahrzeug wird nicht nur das Halten der Fahrspur gefordert, sondern obendrein die essentielle Fähigkeit, dieser selbständig zu folgen - sogenanntes Lane Following. Da die Information über die nutzbare Fahrspur bei der heutigen Infrastruktur hauptsächlich über den visuellen Kanal überbracht wird durch farbig abgesetzte Markierungen, sind hierfür vor allem Computer Vision - Ansätze mit Videokameras ausgerichtet auf das frontale Umfeld des Wagens von Interesse [48].
Die Spurhaltung und das Lane Following haben die notwendigen Verarbeitungsschritte in Form einer Datenaufbereitung und der anschließenden Anwendung eines Algorithmus zur Fahrspurerkennung gemeinsam. Um die Steuerung eines autonomen Fahrzeugs entsprechend des Verlaufs der Fahrspur zu realisieren, ist ein weiterer dritter Schritt von Bedeutung - das Tracking der Fahrbahnmarkierungen [48].
Der Fokus wird im Folgenden auf aktuell angewendete und in Forschung befindliche Methoden zur Detektion und Verfolgung von Fahrbahnmarkierung gelegt.

Vor der eigentlichen Behandlung der Kamerabilder werden meist einige Vorverarbeitungsschritte durchlaufen, welche unter anderem die Erzeugung eines Grauwertbildes, das Entfernen von Bildrauschen, das Ausschneiden des relevanten Bildbereichs (Region of Interest, RoI) oder eine vorab durchgeführte Kantendetektion mithilfe des Canny - Algorithmus beinhalten können [49]. Chetan et al. (2020) teilen die bekannten Herangehensweisen zur Fahrspurerkennung in zwei Lager auf: Der klassische Weg über Bildverarbeitungsalgorithmen und der Einsatz neuer fortschrittlicher Machine Learning-Techniken [48].

Einen geeigneten und bereits erfolgreich implementierten Vertreter des Computer Vision-Ansatzes stellt die projektive Transformation der Bildpunkte in eine Vogelperspektive (Inverse Perspective Mapping, IPM) gefolgt von einer Hough-Transformation zur Extraktion der markanten Kanten der Streifen als Geraden dar [50, 48]. Weiterhin lassen sich die Kanten der Markierungen über die Untersuchung der Helligkeitsgradienten mit dem sogenannten Histogram of Gradients (HOG) in einem Grauwertbild erfassen [51, 52]. Der meist hohe Kontrast zwischen den hellen Streifen und der geteerten Straße resultiert in zahlenmäßig hohen Gradienten, welche ein Hinweis auf eine mögliche Kante eines Markierungsstreifens sein können.
Neben denen auf Bildmerkmalen beruhenden Lösungen werden in der Literatur außerdem modellbasierte Algorithmen diskutiert, wobei sich hierbei eine Spline-Fitting Methodik als besonders effizient herausstellt [52, 50, 48]. Dorj et al. (2020) ergänzen diese Methode durch ein kreisförmiges Modell, um eine korrekte Detektion der Fahrbahn auch bei Kurvenfahrt zu gewährleisten [49]. Sehr populär für die Filterung der erhaltenen Ergebnisse und Bestimmung der optimalen Modellparameter mit dem geringsten Fehler, um falsche Detektionen und Ausreißer zu unterdrücken, ist der RANdom SAmple Consensus Algorithmus (RANSAC) [50, 48].
Um die Performance auch bei schlechten Belichtungsverhältnissen oder starkem Rauschen zu verbessern, wird bei der Vorgehensweise basierend auf vordefinierten Bildmerkmalen der Einsatz lokal adaptiver Schwellwerte empfohlen [50, 49]. Zur Realisierung einer robusten Detektion auch bei verschmutzen, verdeckten oder abgefahrenen Fahrbahnmarkierungen nutzen Son et al. (2019) eine Mittelung der Pixelwerte über mehrere Frames an Stelle nur eines einzelnen Bildes pro Zeitschritt - bekannt unter dem Begriff “zeitliche Verwischung” (engl.: temporal blurring) [50]. Allgemein ist festzustellen, dass die klassischen Bildverarbeitungsalgorithmen eher ungeeignet sind bei Schlechtwetter oder geringer Ausleuchtung des zu untersuchenden Bereichs [51, 50].

In [53] wird ein Ansatz mit klassischen ML-Algorithmen zur Klassifizierung der Markierungen präsentiert, welcher die Erkennung der Fahrspuren anhand der Daten einer Frontkamera ohne Kenntnis der Fahrzeuggeschwindigkeit und ohne Gebrauch von vordefinierten Straßenmodellen absolviert. Hier wird bereits die Verknüpfung einzelner low-level Merkmale - wie Farbintensität, Bildtextur und Kanten - zu einem räumlichen Kontext vollzogen.

Die Genauigkeit und Robustheit der Fahrspurerkennung profitieren seit einigen Jahren stark von den Fortschritten in dem Bereich Deep Learning, besonders den in Kapitel 2.3 vorgestellten CNNs zur automatischen Extraktion der Bildmerkmale und der Möglichkeit, eine Ende-zu-Ende-Lösung abzubilden [51, 54]. Häufig genutzte neuronale Netze sind in diesem Bereich R-CNN, Faster R-CNN, SSD und YOLO [55, 56, 57, 58, 52, 51]. Bei den einstufigen CNNs (SSD und YOLO) lässt sich eine Klassifizierung auf Pixelebene

erzielen zusammen mit einer Abstandsschätzung für jedes Pixel [52]. Oft werden die auszuführenden Aufgaben (z.B. Klassifizierung, Regression, Segmentation) auf mehrere neuronale Netze oder Abzweigungen von einem zentralen Netz verteilt. Pizzati et al. (2019) schlagen die serielle Verknüpfung zweier CNNs vor, wobei zuerst mit dem ERFNet als Basismodell eine Bildsegmentierung zur Festlegung der Umrandungen der einzelnen Markierungsstreifen erfolgt und anschließend diese Informationen genutzt werden, um die erkannten Markierungen einer Klasse (z.B. durchgezogene oder gestrichelte Linie) zuzuordnen [54]. Eine ebenfalls interessante Entwicklung ist unter dem Begriff "Region Fully Convolutional Network" bekannt, welches ohne die typischen letzten vollvernetzten eindimensionalen Schichten zur Bildung der Ausgangsgrößen in Form eines Zeilenvektors auskommt [52]. Zang et al. (2018) legen ein solches Vorgehen mit FCN zur ebenfalls pixelweisen End-to-End-Detektion von Fahrbahnmarkierungen im Gegensatz zur üblichen Bestimmung der Ecken von rechteckigen Bounding Boxen bei der Objekterkennung offen [52].
2017 präsentierten Lee et al. das VPGNet zusammen mit einer 20.000 Bilder umfassenden Trainingsdatensammlung für die Fahrspurerkennung [51]. Neben der geometrischen Lage der Markierungen im Bild (rasterbasierte Klassifizierung) wird das Netz ebenfalls dazu befähigt, die Lage des Fluchtpunkts im Bild vorherzusagen. Hierdurch wird die Tatsache genutzt, dass es sich bei Fahrbahnmarkierungen zumeist um parallele Linien handelt, welche sich laut der Theorie der projektiven Geometrie in einem gemeinsamen Punkt schneiden. Ist dieser "Punkt im Unendlichen" bekannt, wird die Ableitung eines globalen geometrischen Zusammenhangs ermöglicht [51].
Anstatt jedes Bild einer Videokamera einzeln zu betrachten, erlauben Rekurrente Neuronale Netze (RNN) die Untersuchung zusammenhängender Videosequenzen durch eine interne Speicherstruktur. Dies schafft die Möglichkeit der Überbrückung fehlender Daten oder der Rekonstruktion der Fahrbahnlinien bei Verschmutzung oder kurzzeitig falscher Detektion [48]. Auch die Kombination aus einem CNN zur automatisierten Extraktion der relevanten Bildmerkmale und einem RNN, welches diese Informationen nutzt, um schließlich die Fahrbahnmarkierung zu erfassen, ist bereits untersucht worden [51].
Weiterhin können vortrainierte CNNs auch nur für die automatische Merkmalsextraktion genutzt werden und darauf basierend die Stützstellen bei einer parametrisierten kubischen Spline-Fitting Methode der Markierungsstreifen der Ego-Fahrspur vorhergesagt werden. Neben der mithilfe von Deep Learning bestimmbaren Modellparameter wird bei diesem Ansatz nicht nur die Erfassung der vorausliegenden Fahrspur fokussiert, sondern auch die umliegenden Bahnen berücksichtigt, um eine diskrete Spurzuordnung und vor allem die Planung von Spurwechseln zu eröffnen. Durch Anwendung einer konventionellen Encoder-Decoder CNN-Architektur lassen sich die befahrbaren Fahrspuren in einer Draufsicht klassifizieren. Schmidt et al. (2019) untersuchen in ihrem Beitrag die Eignung verschiedener Basis-Netzstrukturen (AlexNet und VGG16) zusammen mit mehreren Modifikationen (Extremely Randomized Trees, Fully connected layers und ein Multiscale-Vorgehen) zur Regression von Spline-Stützstellen [59].
Die große Herausforderung bei den lernfähigen Algorithmen stellen die zugrunde gelegten Daten dar. Das Angebot an öffentlich zugänglicher Datensammlungen ist häufig begrenzt, die Datenmenge ist zu gering oder sie besitzen eine unzureichende Vielfältigkeit [51]. Eine beliebte Basis für Benchmark in der Fahrspurerkennung ist das Caltech Lanes Dataset mit 1225 Bildern [51, 50].

Für das Lane Tracking sind vor allem der Partikelfilter als auch der Kalmanfilter unter Zuhilfenahme eines Bewegungsmodells mit angenommener konstanter Fahrzeuggeschwin-

digkeit oder -beschleunigung sehr weit verbreitet [53, 48, 50]. Alternativen hierzu stellen der Gaussian Sum Particle Filter (GSPF) oder der Algorithmus Geometric Overture for Lane Detection by Intersections Entirety (GOLDIE) dar [52]. Um die jeweiligen Nachteile der Filter auszugleichen, finden sie häufig kombinatorisch Anwendung [48, 52].

In Serie befindliche Systeme zur Fahrspurerkennung nutzen zumeist mehrere simultane Algorithmen für eine höhere Effizienz der Objekterkennung. Durch eine Verknüpfung mit anderen im Fahrzeug befindlichen Object Detection-Systemen und exteroperceptiven Sensoren lässt sich eine hohe Robustheit des Gesamtsystems erzielen [48]. Lösungsvorschläge mit einer Fusion der Daten aus Kamera, LiDAR und GPS versprechen Genauigkeiten bei der Detektion der Fahrspuren im Dezimeter-Bereich, wobei jedoch der Einsatz von LiDAR-Sensoren in PKWs als eher wirtschaftlich unattraktiv angesehen wird [48, 54]. Am vielversprechendsten scheint aus heutiger Sicht ein Lane Following-Konzept für autonomes Fahren basierend auf einer Sensorfusion von mehreren Kameras, Radaren und Ultraschallsensoren [ebd]. Eine zufriedenstellend funktionierende kamerabasierte Fahrspurerkennung bietet das Potenzial, die kostenintensiven hochgenauen digitalen Karten der Umgebung zur Lokalisierung des Fahrzeugs auf der Straße zu substituieren [59].

3.2.2 Autonomes Parken

Der Parkvorgang eines Fahrzeugs verfügt über starke Parallelen und Korrelationen zu der im Rahmen dieser Arbeit zu entwickelnden Rangierfunktion.
Einzigst bei den Ansprüchen an die Positionierungsgenauigkeit dürften sich gewisse Differenzen ergeben. Bei der aktuellen Gestaltung der Parkplätze kann eine Abweichung von 10 bis 15 cm zwischen der vom System erfassten Lage des anvisierten Parkplatzes und der tatsächlichen Position noch als akzeptabel angesehen werden. Mit Blick auf ein Straßenbild dominiert durch selbstfahrende Fahrzeuge, ist im Zuge einer möglichen Steigerung der Parkflächenausnutzung eine Verschärfung der Toleranz auf maximal 5 cm durchaus denkbar [27].

Ein besonderes Augenmerk ist auf die Fortschritte bei der Erkennung von freien Parkgelegenheiten durch fahrzeugeigene Sensorik zu legen, da dieser elementäre Schritt von großer Bedeutung für das anschließend autonom durchgeführte Fahrmanöver ist.
Wie bereits bei der Fahrspurerkennung in Kapitel 3.2.1 behandelt, bieten sich hier ebenfalls optische Detektionssysteme vorrangig mit Gebrauch von mehreren Kameras an. Für die Sensorauswahl in Frage kommen außerdem einzeln oder in Kombination arbeitende Ultraschallsensoren, LiDAR-Sensoren und Radare. Die Ultraschallsensoren, welche vor allem durch ihre langjährig erprobte Verwendung bei der Park Distance Control (PDC) in Serienfahrzeugen bekannt sind, eignen sich nur bedingt zur Detektion von freien Parkplätzen, solange keine Objekte als Referenz vorhanden sind [60, 61, 27].Um bei der Detektion von möglichen Parkgelegenheiten nicht abhängig davon zu sein, dass diese durch benachbarte abgestellte Fahrzeuge oder sonstige Objekte festgelegt werden, stehen hauptsächlich optische Systeme im Fokus der Entwicklung und Forschung [60]. Durch ein kamerabasiertes System sind höhere Genauigkeiten bei der Fahrzeugpositionierung innerhalb der Markierungsstreifen zu erwarten, da diese ideale geometrische Referenzen darstellen im Gegensatz zu eventuell schief oder falsch eingeparkten Nachbarfahrzeugen [27].

Meist werden sogenannte Around View Monitor (AVM) Applikationen verwendet. Diese ermöglichen auf Grundlage der Daten, generiert durch vier kostengünstige Weitwinkelkameras und die anschließende Fusion und perspektivische Transformation, einen Rundumblick des nahen Umfelds des Wagens aus der Vogelperspektive. Auf dieser Datenbasis können Bildverarbeitungsalgorithmen hauptsächlich zur Kantenerkennung der Parkplatzbegrenzungsstreifen angewendet werden [62, 61, 63, 60]. In der Literatur werden bei diesen linienbasierten Verfahren ähnliche Algorithmen wie bei der Fahrspurerkennung behandelt, weshalb jene hier nicht wiederholt erörtert werden.
Ein valides und plausibles Ergebnis liegt vor, wenn bei gradientenbasierten Verfahren eine steigende und eine fallende Flanke in einem Abstand unterhalb eines vorgegebenen Schwellwertes erkannt werden. Ein möglicher freier Parkplatz wird letztlich durch Kreuzungspunkte von vertikalen und horizontalen Begrenzungsstreifen definiert. Nicht nur die Kombination aus Linien und Ecken im Kamerabild lässt Rückschlüsse auf eine ausgewiesene Parklücke zu [61]. Ebenso eine Suche nach Ecken als rein punktbasierte Herangehensweise im Abbild der Umgebung unter Anwendung des Harris-Kantendetektors wird in der Literatur vorgestellt [63, 60].
Diese klassischen Computer Vision - Ansätze leiden jedoch auch hier unter dem starken Einfluss von veränderlichen Umgebungsbedingungen und dem Zustand der Markierungen, wie bereits im vorangegangenen Kapitel 3.2.1 bei der Fahrspurdetektion erläutert [61]. Ergänzend dazu erweitern Bildsegmentierungsalgorithmen die Objekterkennung und in Kombination mit Stereo Vision-Algorithmen zur Bestimmung von Abständen lassen sich Kollisionen mit Hindernissen vermeiden [62].

Die Belegung des Parkplatzes kann über Techniken des optischen Flusses aufgedeckt werden. Zwischen zwei aufeinanderfolgenden Kameraframes wird in [61] die Bewegung von entweder allen Bildpixeln oder von ausgewählten Pixeln basierend auf der Homographie und unter Berücksichtigung des Skalierungsfaktor bei dem Zentralprojektions-Kameramodell anhand einer “Perspectivity ratio map” aus der Kamerakalibrierung zeilenweise mathematisch beschrieben. Die notwendige Kalibrierung der Kamera wird an einem planaren Marker auf dem Boden mit bekanntem Abstand durchgeführt. Als Resultat dieser Bewegungssegmentation steht eine binäre Bildmaske zur Verfügung, welche Aufschlüsse über in oder vor der Parklücke befindliche Objekte gibt. Der vorgeschlagene Algorithmus zeigt eine hohe Anpassungsfähigkeit an variierende Situationen und eine vergleichbare Performance wie die der im weiteren Verlauf vorgestellten neuronalen Netze zur Objekterkennung [61].

Werden die optischen Sensoren durch weitere Ultraschallsensoren unterstützt, laufen folgende Berechnungsschritte in Reihe ab: Visuelle Detektion der Parkplatzstreifen, Untersuchung der Parkplatzbelegung mithilfe von Ultraschall und letztlich die Verfolgung der Begrenzungen während des Fahrmanövers. Hierfür wird das Directional Chamfer Matching (DCM) zur Fusionierung der Sensordaten in [62] genutzt.
Für eine ausführlichere Aufbereitung aller bekannten Smart Parking-Architekturen wird an dieser Stelle auf [62] verwiesen.

Auch bei der Automatisierung des Fahrzeugparkvorgangs hat das Deep Learning Einzug gehalten. Neben End-to-End-Lösungen, welche den Lenkwinkel auf Basis von Bilddaten vorgeben [16], finden sich allgemeine Ansätze zur Objekterkennung durch CNNs in der einschlägigen Literatur wieder [61]. Eine Vielzahl an Untersuchungen bezüglich der Parklückenerkennung durch Anwendung von CNNs auf 360°-Kamerabilder wurden

bereits vorgestellt, wobei sich auch hier ähnliche Netzarchitekturen wie bei der Fahrspurerkennung als besonders geeignet herausstellen. Li et al. (2020) präsentieren eine neue Herangehensweise zur visuellen Erschließung des Parkbereichs mithilfe von Deep Learning. Nach dem Durchlaufen des CNNs stehen hier die Art, die Position, die Länge und vor allem die Richtung der Frontlinie des Parkplatzes, die sogenannte “Eingangslinie”, zur Verfügung [63]. Häufig werden nur die Eckpunkte der Parkfläche detektiert aufgrund der dynamischen Erscheinungsform der Begrenzungsstreifen je nach relativer Position der Kamera zu der Fläche und unter Verwendung von a-priori Wissen bezüglich der Geometrie vervollständigt [63].
Angenommen die Länge der Begrenzungsstreifen ist vorab bekannt, so kann die Detektionsaufgabe als Suche nach einem geordneten Punktepaar der vier Eckpunkte des Parkplatzes formuliert werden. Bei dieser Vorgehensweise können übliche Objekterkennungsmethoden nicht ohne weitere Anpassungen eingesetzt werden [60]. Huang et al. (2019) stellen aus diesem Grund zur Regression der Markierungspunkte eine CNN-basiertes Modell namens “Directional Marking-Point Regression model (DMPR)” vor, welches neben der Bildkoordinaten der erkannten Punkte auch einen Formfaktor, die Orientierung und die Wahrscheinlichkeit in einem Vektor bestehend aus insgesamt sechs Elementen ausgibt. Anschließend werden die Punkte gefiltert, um falsche Detektionen zu eliminieren und die verbleibenden Zwischenergebnisse werden zu validen Punktepaaren gematcht, welche letztlich die Begrenzungen des Parkplatzes definieren [60].
Für das Benchmark im Bereich der Automatisierung des Parkvorgangs wird oft das veröffentliche Datenset “ps2.0” herangezogen [60, 63].

Vor allem das punktbasierte Vorgehen bei der Parkplatzerkennung unter Zuhilfenahme von Deep Learning-Methoden hat starke Ähnlichkeit zu der sogenannten “Landmark Detection”, welche bei der Gesichterkennungs oder auch in der Biologie eine entscheidende Rolle spielt. Hierbei werden neuronale Netze darauf trainiert, bestimmte Punkte, welche meist nicht direkt über geometrische Formen oder Zusammenhänge manuell extrahiert werden können, vorherzusagen [64, 65].

4 Entwicklung der Funktionsbausteine

Nach der Bereitstellung und Erläuterung der Ausgangslage für die Verfolgung der Entwicklungsziele dieser Arbeit wird dieses Kapitel dem Entwicklungsprozess der einzelnen funktionalen Softwarekomponenten von der Konzepterstellung bis zur Umsetzung mit dem Fahrroboter gewidmet.

4.1 Perception und Objekterkennung

Die Aufgabe der Umfeldwahrnehmung beschränkt sich bei dem spezifizierten Anwendungsfall rein auf die Erfassung der direkten frontalen Umgebung des Fahrzeugs, ähnlich wie bei der Detektion von Fahrspurmarkierungen, vergleiche Kapitel 3.2.1. Da von Geschwindigkeiten kleiner als 20 km/h entsprechend der Spezifikation des *U-Shift* auszugehen ist, entfällt die Notwendigkeit einer vorausschauenden Fernfeldüberwachung wie bei der Fahrspurdetektion. Voraussichtlich bewegt sich die Geschwindigkeit während des Rangiermanövers in dem auf Parkflächen oft vorgegebenen Schritttempo von ca. 5 bis 15 km/h. Vielmehr ist die unmittelbare Umwelt vor dem Fahrzeug von großer Bedeutung für eine einwandfreie automatische Abwicklung der Fahraufgabe. Dieser Aspekt ähnelt mehr dem Vorgehen bei autonomen Parkfunktionen, welche bei der visuellen Erkennung freier Parkmöglichkeiten stark auf die Nahfeldüberwachung durch mehrere Weitwinkelkameras setzen, siehe Kapitel 3.2.2.
Die Verschmelzung dieser beiden verwandten Fahrfunktionen führt schließlich zur Verwendung einer monokularen Kamera mit Weitwinkelobjektiv, welche auf das frontale Fahrzeugumfeld ausgerichtet ist. Bei dem verwendeten Fahrroboter zur Funktionsvalidierung ist die Kamera außerdem ca. 75 ° entgegen der Horizontalen geneigt, um die “blinde” Zone direkt vor dem Fahrzeug möglichst klein zu halten und somit gegebenenfalls Hindernisse mit sehr kurzer Distanz zum Ego-Fahrzeug noch durch die Kamera erfassen zu können.

4.1.1 Eignung klassischer Computer Vision-Algorithmen

Da die vorgesehenen Markierungen zur Festlegung des Zielbereichs, vergleiche Kapitel 1.2, durch zwei charakteristische Merkmale - die Farbe Gelb und die rechteckige Form - hervorstechen, werden im Folgenden Ansätze für eine Objekterkennung basierend auf der gezielten Extraktion dieser Größen diskutiert. Bevor ein hoher Aufwand zur Erstellung komplexer Modelle zur Objektdetektion getätigt wird, sind zuerst einfache Bildverarbeitungsalgorithmen auf ihre Eignung zu untersuchen. Trotz des kürzlich großen Erfolgs von Deep Learning-Anwendungen erfährt dieses Vorgehen durchaus Akzeptanz in der Fachwelt, wie aus Kapitel 3.2 hervorgeht.

Bildsegmentierung anhand von Farbwerten:
Für die Definition des relevanten Farbbereichs wird auf den in der Computergrafik vermehrt angewendeten und wahrnehmungsorientierten HSV-Farbraum zurückgegriffen [20]. Verwendet wird, wie bereits innerhalb Grundlagen zu Computer Vision in Kapitel 2.2 vorgestellt, eine Farbtiefe von 8 Bit, wodurch die Farbe Gelb im BGR-Farbraum mit dem Zeilenvektor oder auch Tensor [0,255,255] repräsentiert wird. Eine Umrechnung des Farbvektors in dem vorhandenen Farbraum (RGB, BGR, etc.) ergibt die Werte für reines Gelb im HSV-Raum mit [30,255,255]. Sie stehen für den Farbton (engl.: **H**ue), die Farbsätti-

gung (engl.: **S**aturation) und die Helligkeit (engl.: **V**alue).
Um eine robuste Objekterkennung zu realisieren, ist ein Wertebereich für alle drei Größen des Vektors anhand von Toleranzvorgaben bereitzustellen. Diese jeweiligen Bereiche sind so klein wie möglich, aber gleichzeitig so groß wie nötig zu gestalten, um die gelben Markierungen mit hoher Sicherheit zu erkennen, aber dennoch die Gefahr von falschen Detektionen zu mindern. Die Farbwahrnehmung besitzt eine starke Abhängigkeit von den vorhandenen Lichtverhältnissen und der Umgebung. Somit erfordert dieses Vorgehen ein adaptives Verhalten des Farbfilters zur Anpassung an die Umgebungsbedingungen. Als Resultat steht eine segmentierte Schwarz-Weiß-Bildmaske zur Verfügung, wobei alle Pixel mit dem Farbwert Weiß dem Objekt zugeordnet sind und die restlichen Pixel in der Farbe Schwarz als Hintergrund zählen.
Da diese pixelweise Bildsegmentierung über Farbwerte verrauschte Ergebnisse erwarten lässt, sodass die binäre Klassifizierung keine eindeutigen Objektabgrenzungen und somit die Lage dieser im Bild hervorbringt, ist die erhaltene Bildmaske weiter zu verarbeiten. Ein geeigneter Algorithmus sucht gezielt nach Konturen innerhalb des Schwarz-Weiß-Bildes. Nur die beiden Konturen mit den größten eingeschlossenen Flächen, entsprechend den zwei zu erkennenden Markierungen, werden für die weitere Auswertung herangezogen. Berücksichtigung findet außerdem das Wissen über die Form der Markierungen, indem anstelle beliebig ausgebildeter Konturen Rechtecke eingepasst werden. Zu beachten ist, dass keine Bounding Boxen mit bildachsenparallelen Kanten angewendet werden, um die Umrandungen der Streifen möglichst gut anzunähern.

Es folgt eine Diskussion der Vor- und Nachteile dieser manuellen Extraktion von Bildmerkmalen zur Objekterkennung anhand der anschaulichen Ergebnisse in Abbildung 4.1. Hierbei wurde der vorgestellte Algorithmus auf einen kleinen Testdatensatz, aufgenommen mit der unkalibrierten Kamera des Fahrroboters (siehe Kapitel 2.4), angewendet. Die 200 Bilder umfassende Sammlung repräsentiert Szenen an einem bewölkten Tag im Außenbereich eines Hauseingangs. Weiterhin Beachtung finden die beiden zu Beginn dieser Arbeit vorgestellten Fälle der möglichen Platzierung der Markierungen im Raum, vergleiche Abbildung 1.3. Neben der Umwandlung der Farbdarstellung von BGR zu HSV erfahren die Bilder keine weitere Vorbehandlung, um die Leistungsfähigkeit des vorgestellten Algorithmus unabhängig beurteilen zu können. Abbildung 4.1 stellt ausgewählte Auszüge der erhaltenen Resultate dar. Nach manueller Optimierung der Funktionsparameter anhand einer Zufallssuche sind passende Wertebereiche für die zu erkennenden Farbabstufung festgelegt durch den jeweiligen minimalen als auch maximalen Wert: H = [15, 45], S = [155, 255] und V = [135, 255].

Die Beispiele in Abbildung 4.1 a) und 4.1 d) stellen zufriedenstellende Ergebnisse dieser konturnahen Objekterkennung und -lokalisation dar. Die Umrandungen der beiden Markierungsstreifen werden durch den Algorithmus sehr gut angenähert. In Abbildung 4.1 b) erkennt der Algorithmus nur einzelne Bereiche der Markierungen im Bild als zusammenhängende Flächensegmente. Die Approximation der Umrisse der Markierungen anhand der Annahme einer rechteckigen Form scheitert ebenfalls unter dem Einfluss projektiver Gesetzmäßigkeiten in einem 2D-Abbild des dreidimensionalen Raums. Ursprünglich parallele Linien erscheinen auf einem 2D-Bild nicht mehr parallel und schneiden sich in einem “Punkt im Unendlichen”, oft referenziert als Horizont (vergleiche Kapitel 2.2). Zu sehen ist dieses Phänomen besonders in Abbildung 4.1 c). Sind die Markierungen am Boden platziert, werden sie trapezförmig dargestellt und sie können nicht mehr gut durch

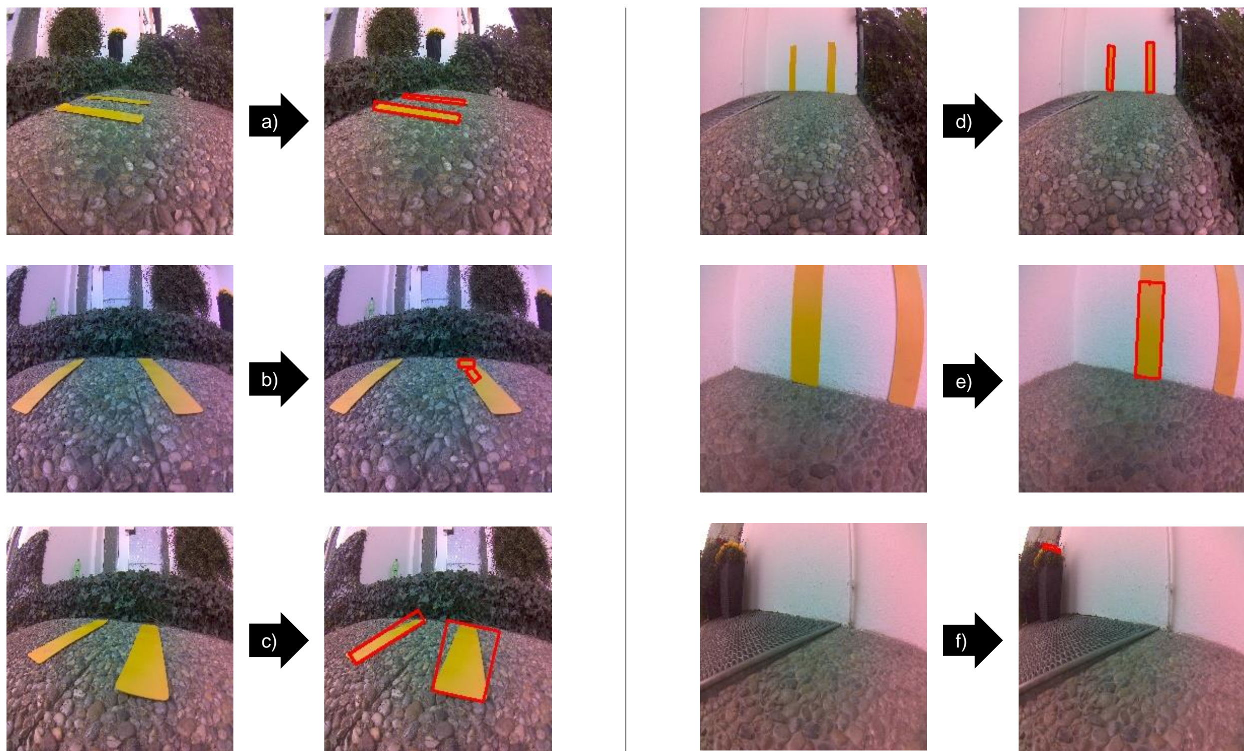


Abb. 4.1: Ausgewählte Beispiele der Resultate einer Objektdetektion basierend auf Pixelfarbwerten [eigene Darstellung]

Rechtecke im Bild repräsentiert werden. Da die stark ausgeprägten nichtlinearen Abbildungserscheinungen der Weitwinkelkamera vorab nicht bereinigt wurden, zeigt Abbildung 4.1 e) den Einfluss einer tonnenförmigen rotationssymmetrischen Verzeichnung, wie in Kapitel 2.2 vorgestellt, deutlich. Der Streifen am rechten Bildrand wird stark gekrümmt abgebildet. Außerdem verändert sich die Farbdarstellung radial ausgehend vom Zentrum zum Bildrand, wodurch der linke Streifen nahe des Bildzentrums gut erkannt wird, jedoch der am Rand befindliche Streifen außerhalb des festgelegten Farbspektrums liegt und somit nicht als Objekt klassifiziert wird. Einen großen Störfaktor für diesen einfachen Algorithmus stellen nicht zu detektierende Objekte mit einer Farbgebung im gesuchten Wertebereich dar. So werden, wie in Abbildung 4.1 f) zu erkennen, zufällig die gelb blühenden Pflanzen als Objekte registriert. Dieses Risiko einer fälschlichen Detektion steigt stark an, wenn die gesuchten Ziele nur einen kleinen Teil des Gesamtbildes einnehmen oder sich überhaupt nicht innerhalb des Sichtfeldes befinden. Eine Maßnahme zur Abschwächung dieses ungewünschten Verhaltens kann die vorherige Festlegung einer RoI, wie bei der Fahrbahnmarkierungsdetektion in Kapitel 3.2.1, sein.

Die vorangegangene Erörterung zeigt auf, dass sich ein einfacher Bildverarbeitungsalgorithmus durchaus eignet, um die eingesetzten Markierungsstreifen möglichst exakt zu erkennen und innerhalb des Bildes einer Videokamera zu lokalisieren. Über geeignete Vorverarbeitungsschritte besitzt das Vorgehen durchaus Optimierungspotenzial. Jedoch unterliegt das Verfahren einem starken Einfluss der Kameraperspektive und der Dynamik der Umgebung. Vor allem könnte die korrekte Funktionsfähigkeit der Objekterkennung bereits durch andere gelbe Objekte gestört werden. Die erforderliche Robustheit ist nicht gegeben, da bereits ein blühender Löwenzahn am Straßenrand zu einer Fehlinterpretation der Umgebung führen könnte. Diese Erkenntnisse führen dazu, dass ein derartiger

Bildverarbeitungsalgorithmus als ungeeignet für die hier erforderliche Objektdetektion einzustufen ist.

Kantenbasiertes Verfahren - Canny-Algorithmus und Hough-Transformation:
Ein weiterer vielversprechender Ansatz fokussiert sich auf die Feststellung von Kanten innerhalb eines Bildes. Vorgestellt wird im Rahmen dieses Abschnitts eine Kombination aus dem Canny-Kantendetektor und einer anschließenden Hough-Transformation, welche sich ebenfalls bei der Lösung der Objekterkennung verwandter Fahraufgaben bewährt haben, siehe Kapitel 3.2. Zunächst werden durch Filterung des Bildes mithilfe des Sobel-Operators in horizontaler und vertikaler Richtung die Intensitätsgradienten und deren Richtung bestimmt. Liegt ein lokales Maximum vor, wird senkrecht zu der Richtung des Gradienten eine mögliche Kante für den nächsten Schritt vorgeschlagen. Zur Unterdrückung von falschen Kantendetektionen ist die binäre Bildmaske mit zwei Schwellwerten zu beaufschlagen. Besitzt die ausschlaggebende Ableitung einen Wert über dem oberen Schwellwert, so wird das entsprechende Pixel als sicheres Element einer Kante gesetzt. Alle Pixel unterhalb des minimalen Schwellwerts werden wiederum verworfen. Eine gewisse Hysterese zwischen beiden Schwellwerten ermöglicht die Berücksichtigung von nicht eindeutigen Fällen. So können bestimmte Pixel zunächst als unsicheres Ergebnis deklariert werden, aber durch Betrachtung des Kontexts über eine Kantenverfolgung eventuell doch einer vorhandenen Kante zugewiesen werden, wenn sich in unmittelbarer Nachbarschaft Pixel mit sicherer Kantenzugehörigkeit befinden. Ist dies nicht der Fall, werden diese Pixel ebenfalls unterdrückt. Ein wichtiger Vorverarbeitungsschritt bei diesem Algorithmus ist die Reduzierung von Bildrauschen durch vorherige Anwendung von linearen Filtern, wie auch in der online verfügbaren *OpenCV*-Dokumentation beschrieben [66].
Für die Erkennung der rechteckigen Markierungen sind jedoch nur die geradlinigen Kanten von Interesse, da die Linearität in der projektiven Geometrie als invariant gilt. Eine Gerade in der realen Welt wird bei der Reduktion auf ein zweidimensionales Abbild ebenfalls als solche dargestellt. Ausnahmen ergeben sich durch vorhandene Nichtlinearitäten während des Abbildungsprozess, wie vor allem bei Weitwinkelkameras zu beobachten, vergleiche Kapitel 2.2. Mit dem mathematischen Modell in Form einer Geradengleichung werden möglichst viele Pixel gesucht, welche auf einer gemeinsamen Geraden liegen. Angewendet wird hier die probabilistische Hough-Transformation. Statt der Ausgabe der Modellparameter als Polarkoordinaten (Winkel und Abstand zum Ursprung) werden die Start- und Endpunkte der Linien im kartesischen Koordinatensystem bereitgestellt. Variable Parameter dieses modellbasierten Algorithmus sind die Abstands- und Winkelauflösung, ein Schwellwert, die minimal erforderliche Länge der Linien und ein maximal zulässiger Abstand zwischen den Punkten auf einer Geraden [67].

Ebenso wie bei der farbbasierten Bildsegmentierung wird dieses Verfahren bei dem vorliegenden Datensatz ohne vorherige Bildbearbeitungsschritte angewendet. Für eine gute Vergleichbarkeit der beiden unterschiedlichen Herangehensweisen zur Detektion solcher farblich abgesetzter Markierungen werden die gleichen Beispiele wie in Abbildung 4.1 herangezogen. Abbildung 4.2 visualisiert die Ergebnisse einer sequentiellen Abfolge eines Canny-Algorithmus mit abschließender Erfassung aller geradlinigen Kanten durch eine Hough-Transformation. Die Schwellwerte des Canny-Algorithmus werden manuell mit den relativ hohen Zahlenwerten 200 und 250 festgesetzt, um möglichst nur die dominanten Kanten im Bild herauszufiltern. Zufriedenstellende Ergebnisse werden bei der

Hough-Transformation, auf diesen speziellen Datensatz bezogen, nach mehreren gezielten Versuchen mit einer Abstandsauflösung von einem Pixel, einer Winkelauflösung von 1°, mindestens 20 Pixeln pro Linie, der minimalen Länge einer Linie von 50 Pixeln und einer maximalen Lücke zwischen Punkten auf einer Linie von 10 Pixeln erzielt.

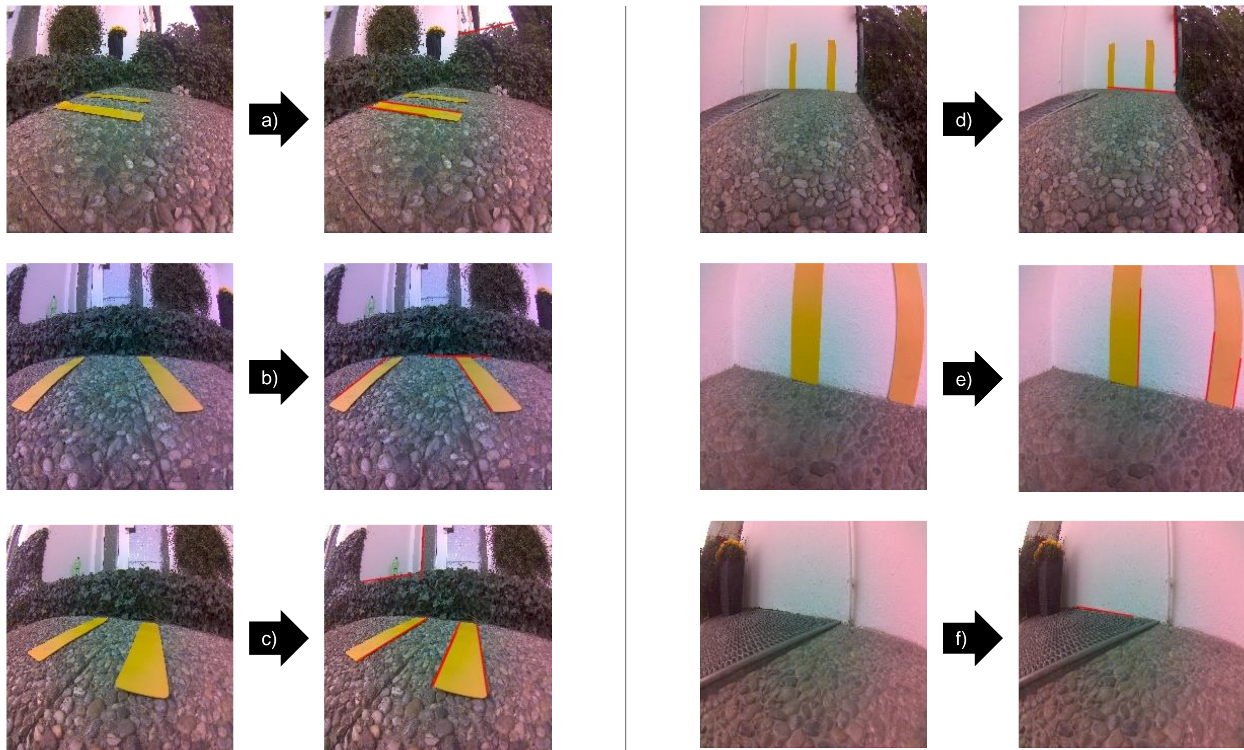


Abb. 4.2: Ausgewählte Beispiele der Resultate einer Objektdetektion durch Anwendung des Canny-Kantendetektors und einer probabilistischen Hough-Transformation [eigene Darstellung]

An dieser Stelle sind die Vorzüge als auch die Herausforderungen bei der Implementierung eines solchen kantenbasierten Verfahrens zur Objekterkennung zu erörtern. Zunächst ist bei der Betrachtung der Ergebnisse festzustellen, dass häufig nur die hochkantigen Begrenzungslinien der Markierungen erfasst werden, vergleiche Abbildung 4.2 a), b), c) und e). Die Vorgabe einer minimalen Linienlänge bei der Hough-Transformation ist ausschlaggebend hierfür. Durch diese Parameterwahl werden die kurzen Kanten der Rechtecke unterdrückt. Wird jedoch keine Einschränkung vorgenommen, werden auch viele weitere Linien im Hintergrund als Ergebnisse aufgefasst. Im Gegensatz zu der Annäherung der Konturen durch Rechtecke lassen sich die trapezförmigen Formen hier durch Geraden viel besser annähern, siehe Abbildung 4.2 c) und b). Sollen die vollständige Form und die Position der Markierungen im Bild aus den detektierten Linien abgeleitet werden, müssen alle Abgrenzungslinien zwischen Objekt und Hintergrund bekannt sein. Alle Beispiele in Abbildung 4.2 beweisen, dass dies hier nicht zuverlässig gewährleistet werden kann ohne einen viel umfangreicheren Ergebnisraum in Kauf nehmen zu müssen. In Abbildung 4.2 d) werden die Umrandungen der Markierungen überhaupt nicht erfasst. Die Hough-Transformation bietet ebenfalls keine zufriedenstellenden Resultate bei der Detektion von verzerrten Abbildern der rechteckigen Objekte, wie in Abbildung 4.2 e) ersichtlich. Eine Abhilfe würde hier der Einsatz von Curve-Fitting-Methoden, wie auch oft in der Modellierung von Fahrbahnmarkierung (siehe Kapitel 3.2.1) verwendet, schaffen, um auch nicht lineare Formen besser erfassen zu können. Einen Vorteil gegenüber der farb-

basierten Segmentierung besitzt das vorgestellte Verfahren hinsichtlich der ungewollten Detektion von Hintergrundobjekten. Insofern diese keine geradlinigen Konturen besitzen, werden sie nicht fälschlicherweise erkannt, siehe Abbildung 4.2 f).
Eine rein kantenbasierte Objektdetektion und -lokalisierung ohne Extraktion zusätzlicher Bildmerkmale wird von der Herausforderung begleitet, alleinig die Begrenzungslinien der Markierungen als verwertbare Ergebnisse zu erhalten. Wie Abbildung 4.2 zeigt, werden selbst bei sorgfältiger Auswahl der Funktionsparameter noch falsch erkannte Linien angezeigt. Auch hier ergibt sich die Anforderung an den Algorithmus der Anpassungsfähigkeit und Unempfindlichkeit gegenüber Störobjekten, um möglichst in allen denkbaren Situationen robust zu agieren. Für eine Weiterverfolgung im Rahmen dieser Arbeit ist dieser Ansatz ebenfalls nicht zielführend.

Ein kombinierter Einsatz der beiden in diesem Kapitel aufgeführten Computer Vision-Algorithmen würde die Leistungsfähigkeit der Objekterkennung in einem Farbbild deutlich verbessern, da wichtige Bildmerkmale nicht ungenutzt bleiben und die Stärken der jeweiligen Verfahren sich gegenseitig ergänzen. Zwar lassen sich diese Algorithmen schnell implementieren und liefern ohne großen Aufwand erste brauchbare Ergebnisse, jedoch ist die Güte der Resultate stark situationsabhängig und anwendungsspezifisch. Vor allem die relativ hohe Störanfälligkeit erfordert die Bereitstellung eines kontrollierten Umfelds mit konstanten Rahmenbedingungen, was in dem vorliegenden Anwendungsfall nicht gewährleistet werden kann.
Die bisher vorgestellten Ansätze werden nach einer ersten Untersuchung hinsichtlich ihrer Eignung aufgrund ihrer ungenügenden Robustheit und Genauigkeit im weiteren Verlauf dieser Arbeit nicht weiter verfolgt. Entsprechend dem Stand der Technik wird der Fokus bei der Realisierung einer möglichst exakten Detektion der Markierungen in den Farbbildern einer Weitwinkel-Videokamera auf die Applikation von DL-Modellen gelegt.

4.1.2 Landmark Detektion mit Deep Learning

Nachfolgend wird ein Konzept für die essentielle Funktionskomponente *Perception* des zu automatisierenden Rangiermanövers vorgestellt, ausgearbeitet und verifiziert.

Konzept:
Angelehnt an die visuelle Erschließung von Parklücken über die Detektion der Eckpunkte, wie bereits in vorherigen Arbeiten behandelt, siehe Kapitel 3.2.2, wird im Folgenden eine Abänderung und Adaption dieses Vorgehens auf den vorliegenden Anwendungsfall vorgenommen. Das daraus hervorgehende Konzept benötigt nicht zwingend rechteckige Markierungsstreifen zur Vorgabe der Ground Truth. Die Referenzpunkte müssen nicht zuerst über die Mittellinien der Kennzeichnungen oder deren Schnittpunkte definiert werden. Dies ermöglicht den Einsatz beliebiger optischer Trigger, insofern diese direkt vier nutzbare Punkte im Raum definieren. Diese vier Punkte legen nicht nur die zu befahrende Zielfläche für die Rangierfunktion fest, sondern sie werden auch später zur Positionsbestimmung des Fahrroboters relativ zu diesem Bereich genutzt, siehe Kapitel 4.3.
Zusammengefasst wird statt einer reinen Objektdetektion als Ziel für die Entwicklung dieses Funktionsbausteins die Regression von vier Bildpunktkoordinaten unter Nutzung der vorhandenen Bildmerkmale und der räumlichen Zusammenhänge verfolgt.
In beiden Fällen der Konfiguration der Markierungen, wie in Kapitel 1.2 beschrieben, werden jene Eckpunkte der Markierungen als Landmarks ausgewählt, welche die einge-

schlossene Fläche festlegen. Anschließend erhalten diese eine Nummerierung entgegen dem Uhrzeigersinn. Abbildung 4.3 ergänzt die Erläuterungen des Konzepts durch eine Prinzipskizze unter Berücksichtigung der spezifizierten Markierungsanordnungen.

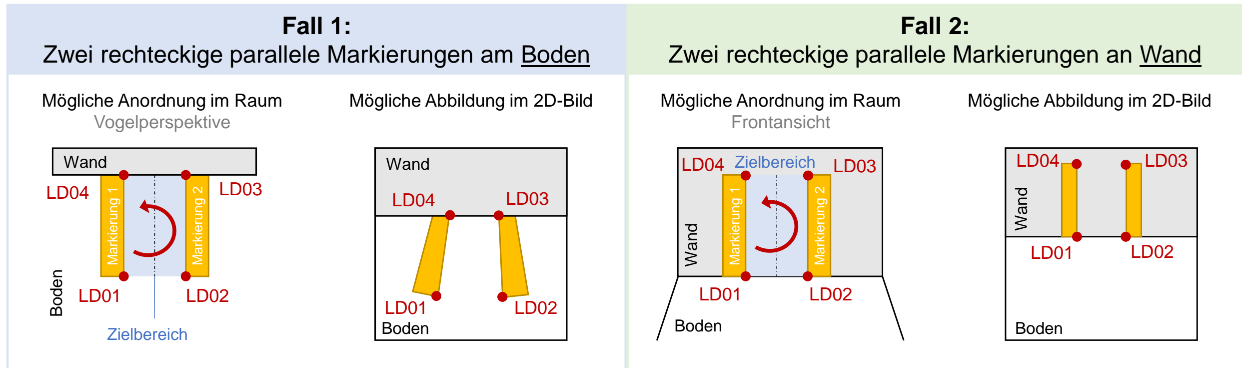
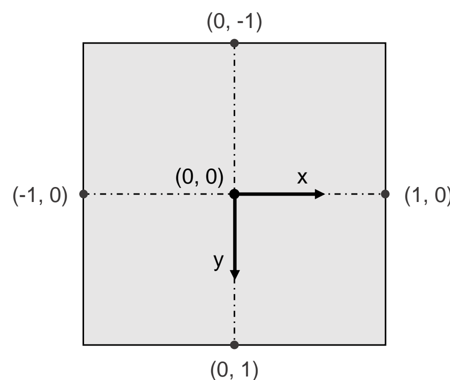


Abb. 4.3: Prinzipskizze des punktbasierten Konzepts zur visuellen Detektion des Zielbereichs der Rangierfunktion [eigene Darstellung]

Modellarchitekturen:
Zur Vorhersage der Lage aller vier Punkte in jedem Frame der Videokamera findet ein neuronales Netz Anwendung. Ergänzt werden die erwarteten Ausgabewerte eines solchen Modells durch eine Angabe zu der Objektwahrscheinlichkeit p im Bild. Hierbei ist bereits ein binärer Ergebnisraum ausreichend, welcher die Zahl Null dem Ereignis “kein Objekt im Bild” und die Zahl Eins der Aussage “Objekt im Bild erkannt” zuordnet. Die zweidimensionalen Koordinaten (x,y) der Landmarks sind auf ein kartesisches Koordinatensystem mit dem Ursprung im Bildmittelpunkt bezogen. Außerdem erfolgt eine Normierung der Koordinaten auf die halbe Bildbreite bzw. die halbe Bildhöhe, sodass diese reellen Zahlen in dem Wertebereich [-1, 1] liegen. Es wird eine Unabhängigkeit von der verwendeten Bildgröße erreicht. Das hiermit festgehaltene Bildkoordinatensystem als auch der vorgesehene Ergebnisvektor für die Weiterverarbeitung durch die anderen Funktionsbausteine sind in Abbildung 4.4 enthalten.



Output = $[\, p,\ x_1,\ y_1,\ x_2,\ y_2,\ x_3,\ y_3,\ x_4,\ y_4 \,]$

Abb. 4.4: Festlegung der Modell-Outputs und des verwendeten Bildkoordinatensystems [eigene Darstellung]

Unter dem Bestreben, den Entwicklungsaufwand bei diesen datengetriebenen und komplexen Modellen zu reduzieren, ist die Entwicklungsmethodik des Transfer Learnings zu bevorzugen. Die daraus hervorgehenden DL-Modelle setzen sich aus einem festen, vortrainierten Backbone und den letzten individuell anpassbaren und meist voll vernetzten Modellschichten, dem sogenannten Detektionskopf, zusammen. Üblicherweise werden als Backbone-Struktur zweistufige Netzwerke, speziell entworfen für die Objektklassifizierung, gewählt [68, 69]. Wie bereits in Kapitel 2.3 aufgegriffen, folgt der Aufbau dieser Netze fast immer einem ähnlichen Schema. Die ersten Netzschichten erzeugen aus dem mehrdimensionalen Dateninput aussagekräftige Merkmalsbeschreibungen und reduzieren gleichzeitig die Datendimensionen durch eine Serienschaltung mehrerer Convolution und Pooling Layer. Die finalen Schichten stellen schließlich das Mapping zwischen den bereitgestellten Informationen über die im Bild enthaltenen Merkmale und den Ausgabewerten dar. Einem Modell die Extraktion der wichtigen im Bild enthaltenen Informationen von Grund auf anzulernen, erfordert einen enormen Trainingsaufwand und eine große Datenmenge. Die Übernahme der für die Merkmalsentnahme zuständigen Netzschichten zusammen mit bereits vortrainierten Kantengewichten ermöglicht eine effiziente Applikation der Modelle auf verschiedenste Anwendungsfälle, da die durchzuführende Entwicklungsarbeit auf das überschaubare Teilmodell in Form des Detektionskopfes konzentriert wird. Diese abschließenden Layer lassen sich individuell abändern, ergänzen oder auch vollständig ersetzen.
Zu unterscheiden ist bei der Anwendung des Transfer Learnings zwischen zwei grundlegenden Vorgehensweisen, wie bereits in Kapitel 2.3 angeschnitten:

1. Einfrieren der vortrainierten Kantengewichte des Backbones und Optimierung der Parameter des applizierten “Detection heads” im Training mit dem eigenen Datensatz.
2. Initialisierung der Parameter der Backbone-Struktur anhand der verfügbaren Kantengewichte und Training beider Teilnetze mit dem vorliegenden Datensatz.

Beide Methoden werden im weiteren Verlauf angewendet und schließlich evaluiert. Hierfür werden zwei unterschiedliche CNNs als Backbone bestimmt und jeweils mit mehreren Variationen der Detektionsköpfe versehen. In einem weiteren Schritt sind zusätzlich auch die Hyperparameter miteinbezogen und innerhalb eines diskretisierten Raums variiert. Diese rasterbasierte Optimierung fördert besonders das Verständnis für die Auswirkungen bestimmter Anpassungsmaßnahmen auf das Endergebnis und legt die Kontrolle über die vorgenommen Adaptionen in die Hände des Entwicklers.

Aus den online verfügbaren PyTorch-Modellen mit vortrainierten Parametersätzen für die Bildklassifizierung werden das ResNet50 und das EfficientNetB0 als Backbone-Netze festgelegt. Die ResNet-Modellfamilie wird von He et. al (2015) in [70] vorgestellt, wobei das ResNet50 mit insgesamt 25,6 Mio. Parametern das mittelschwere Modell der Familie darstellt. Die Einführung der ResNet-Modelle mit den typischen “skip connections” ohne zusätzliche Parameter verspricht bessere Trainingsergebnisse bei sehr tiefen Netzen [ebd]. Da sowohl der Speicherbedarf als auch die Anzahl an Rechenoperationen bei mobilen Echtzeit-Anwendungen von großer Bedeutung sind, wird zusätzlich das leichtere EfficientNetB0 von Tan und Le (2019) mit nur 5,3 Mio. Parametern in Betracht gezogen [71]. Diese Netzarchitektur geht aus einer Pareto-Optimierung mit den zwei Größen der

Modellgenauigkeit und der Gleitkommazahlberechnungen pro Sekunde (FLOPS) hervor und dient als Basismodell für weitere mehrdimensional skalierte Modelle der EfficientNet-Familie [ebd].

Beide künstlichen neuronalen Netze wurden mit dem Benchmark-Datensatz “ImageNet-1K_V1” vortrainiert und sie besitzen somit als Ausgabeschicht ein lineares eindimensionales Layer, welches 1000 Ausgabewerte entsprechend den zu differenzierenden Klassen bereitstellt. Die Anpassung an die hier notwendige Regression der neuen Modellausgänge, wie der Abbildung 4.4 zu entnehmen, wird an Stelle dieser Ausgabeschicht in der Netzstruktur in unterschiedlichen Umfängen vorgenommen. Die dadurch definierten Konfigurationen und deren im weiteren Verlauf verwendeten Bezeichnungen fasst Tabelle 4.1 zusammen.

Die V11 und V21-Modelle stellen die Varianten der ursprünglichen CNNs mit den minimal invasiven Modifikationen dar. Lediglich die Größe der Ausgabeschicht wird an die hier erforderlichen neun Modellausgänge adaptiert. Bei den Varianten V12 und V22 findet eine Erweiterung des Detektionskopfes statt, indem die Linearkombination aller Eingänge eines Neurons innerhalb einer *Fully Connected (FC)* linearen Schicht als Eingabe einer Tangens Hyperbolicus Aktivierungsfunktion dient und vor der Übergabe an die Ausgabeneuronen ein Dropout-Layer platziert wird. Die Tanh-Funktion scheint für den vorliegenden Anwendungsfall als besonders geeignet, da sie einen begrenzten Wertebereich zwischen [-1, 1] sicherstellt, ein nichtlineares Verhalten induziert und die Funktion eine Punktsymmetrie besitzt, wodurch sowohl negative als auch positive Eingabewerte gleichermaßen aktiviert werden. Eine weitere Vertiefung der Netzstruktur des Detektionskopfes weisen die Modelle V13 und V23 auf. Durch eine sequentielle Abfolge von voll vernetzten Schichten mit Tanh-Aktivierungsfunktionen, zwei Dropout-Layern mit unterschiedlichen Dropout-Wahrscheinlichkeiten und einer Halbierung der Neuronen innerhalb jeder Schicht bezogen auf die Vorgängerschicht wird vor allem eine Reduzierung der Parameter im Vergleich zu den Versionen V12 bzw. V22 bei gleichzeitig größerer Tiefe des Detektionskopfes angestrebt. Trotz einer höheren Komplexität des Detektionskopfes können somit nicht nur die zu trainierenden Parameter deutlich reduziert werden, sondern auch der notwendige Speicherplatz für die Netzstruktur und die Kantengewichte, vergleiche Tabelle 4.1. Da der Tangens Hyperbolicus mit den zu bestimmenden normierten Bildkoordinaten den gleichen Wertebereich teilt, wird dessen Eignung in der Praxis als Aktivierungsfunktion in der letzten Schicht durch den Aufbau der Modelle V14 und V24 untersucht. Eine Validierung der Leistungsfähigkeit der eingesetzten Dropout-Layer in den Versionen V12, V13, V22 und V23 soll durch die nahezu baugleichen Konfigurationen V15, V16, V25 und V26 abzüglich aller Dropout-Layer ermöglicht werden.

Tabelle 4.1: DL-Modellkonfigurationen für die Detektion der Landmarks

Transfer Learning Methode	Detektionskopf	Name	Parameter		Speicherbedarf
			total	trainierbar	
Vortrainierte Backbone Parameter eingefroren	FC Linear (2048, 9) / (1280, 9)	ResNet_V11	23,5 Mio.	18441	260 MB
		EffNet_V11	4,0 Mio.	11529	23 MB
	FC Linear (2048, 2048) / (1280, 1280); Tanh (2048) / (1280); Dropout (0.2) / (0.2); FC Linear (2048, 9) / (1280, 9)	ResNet_V12	27,7 Mio	4,2 Mio.	253 MB
		EffNet_V12	5,7 Mio.	1,7 Mio.	29 MB
	FC Linear (2048, 1024) / (1280, 640); Tanh (1024) / (640); Dropout (0.5) / (0.5); FC Linear (1024, 512) / (640, 320); Dropout (0.2) / (0.2); Tanh (512) / (320); FC Linear (512,9) / (320, 9)	ResNet_V13	26,1 Mio	2,6 Mio	247 MB
		EffNet_V13	5,0 Mio.	1,0 Mio.	27 MB
	ResNet_V11 / EffNet_V11 + Tanh (9)	ResNet_V14	23,5 Mio	18441	237 MB
		EffNet_V14	4,0 Mio.	11529	23 MB
	ResNet_V12 / EffNet_V12 o. Dropout	ResNet_V15	27,7 Mio.	4,2 Mio.	253 MB
		EffNet_V15	5,7 Mio.	1,7 Mio.	29 MB
	ResNet_V13 / EffNet_V13 o. Dropout	ResNet_V16	26,1 Mio.	2,6 Mio.	247 MB
		EffNet_V16	5,0 Mio.	1,0 Mio.	27 MB
Vortrainierte Backbone Parameter als Initialisierung	s. ResNet_V11 s. EffNet_V11	ResNet_V21	23,5 Mio.	23,5 Mio.	260 MB
		EffNet_V21	4,0 Mio.	4,0 Mio.	23 MB
	s. ResNet_V12 s. EffNet_V12	ResNet_V22	27,7 Mio.	27,7 Mio.	253 MB
		EffNet_V22	5,7 Mio.	5,7 Mio.	29 MB
	s. ResNet_V13 s. EffNet_V13	ResNet_V23	26,1 Mio.	26,1 Mio.	247 MB
		EffNet_V23	5,0 Mio.	5,0 Mio.	27 MB
	s. ResNet_V14 s. EffNet_V14	ResNet_V24	23,5 Mio.	23,5 Mio.	237 MB
		EffNet_V24	4,0 Mio.	4,0 Mio.	23 MB
	s. ResNet_V15 s. EffNet_V15	ResNet_V25	27,7 Mio.	27,7 Mio.	253 MB
		EffNet_V25	5,7 Mio.	5,7 Mio.	29 MB
	s. ResNet_V16 s. EffNet_V16	ResNet_V26	26,1 Mio.	26,1 Mio.	247 MB
		EffNet_V26	5,0 Mio.	5,0 Mio.	27 MB

Datensatz:
Wie aus den Erläuterungen in Kapitel 2.3 hervorgeht, ist vor allem das überwachte Lernen als Trainingsmethode bei CNNs sehr weit verbreitet. Erforderlich ist hierfür eine möglichst große beschriftete bzw. “gelabelte” Bildersammlung. Mit dem Bestreben, die Vorverarbeitungsschritte der Rohdaten der Videokamera auf ein Minimum zu reduzieren, soll das später eingesetzte neuronale Netz direkt aus den verzeichneten und unkorrigierten Bildern der Weitwinkelkamera die Koordinaten der Landmarks zusammen mit der Objektwahrscheinlichkeit vorhersagen können. Es ist zu empfehlen, die Trainingsdaten direkt mit der im späteren Betrieb vorgesehenen Kamera aufzunehmen, um auf die Parametrisierung von zusätzlichen Kameramodellen zu verzichten. Bei der Generierung der Bilddaten wird die Möglichkeit der Fernsteuerung des Roboters über einen am Host-PC angeschlossenen Controller genutzt. Für die erste Trainingsphase zur Identifikation einer geeigneten Modellarchitektur werden kontrolliert insgesamt 1298 Farbbilder mit einer Auflösung von 224 x 224 Pixeln aufgenommen. Der Datensatz für die Evaluierung herkömmlicher Bildverarbeitungsalgorithmen in dem vorangegangenen Kapitel erfährt eine Erweiterung durch Aufnahmen vor allem in der Büroumgebung. Bei der Erstellung der Trainingsdaten liegt das Augenmerk besonders auf den folgenden Begebenheiten:

- Platzierung der Markierungen auf dem Boden als auch an der Wand (Fall 1 und Fall 2)
- Parallelität der Markierungen mit unterschiedlichen Abständen (d = 15 cm, 25 cm und 35 cm)
- Unterschiedliche Beleuchtungssituationen und Hintergründe
- Möglichst viele Kameraperspektiven und Variationen des Abstandes zwischen der Kamera und den Zielobjekten
- Gezielte Platzierung von gelben rechteckigen Störobjekten im Sichtfeld der Kamera (z.B. gelbe Klebezettel, weitere gelbe Markierungen)
- Ausgewogenheit zwischen Positiv- und Negativbeispielen (Objektwahrscheinlichkeit = 0 oder 1)

Der Labeling-Prozess dieser Bilder wird durch ein eigens angefertigtes Pythonskript unterstützt. Die Bilder werden automatisch aus dem vorgegebenen Speicherort eingelesen, geöffnet und der Benutzer beurteilt bei jedem Bild zunächst, ob beide Markierungen eindeutig im Bild zu sehen sind oder nicht. Ist die Antwort auf diese Frage positiv, werden über vier Mausklicks entsprechend der Reihenfolge, festgelegt in Abbildung 4.3, die Landmarks gesetzt. Erfolgt keine Angabe von genau vier Punkten informiert darüber eine Fehlermeldung. Falsch gesetzte Landmarks können über einen Rechtsklick in der Bildfläche wieder gelöscht werden. Durch Drücken einer beliebigen Taste bestätigt der Benutzer die Eingabe und das Bild wird im Zielordner mit einem neuen Namen abgespeichert. In dem Fall, dass die Markierungen im Bild nicht ausreichend zu identifizieren sind, kann das Bild sofort übersprungen werden und die Objektwahrscheinlichkeit als auch alle Bildkoordinaten der Landmarks werden automatisch zu Null gesetzt.
Die gesetzten Label sind in dem neuen Dateinamen der Bilder enthalten, wodurch das Einlesen der Bilder als auch deren Label während des Trainings simultan erfolgen kann. Bei dem hier behandelten Modelloutput mit den neun Ausgabegrößen ist dieses Vorgehen noch praktikabel, da die Länge der Labels überschaubar ist. Bei größeren Label-

dimensionen ist auf eine separate Abspeicherung dieser Informationen gegebenenfalls zurückzugreifen. Wichtig bei der Darstellung der Label ist die Position der Sonderzeichen “#” und “&”. Sie dienen als Anker für das korrekte Einlesen der Labeldaten.
Um die Vielfalt der Trainingsdaten zu verbessern, findet eine bestimmte Methode der Data Augmentation Anwendung. Über ein dreimaliges zufälliges *Color Jitter* aller 1298 Bilder entstehen 3894 weitere Trainingsbilder, wodurch der finale Datensatz insgesamt 5192 Bilder umfasst mit einer Speichergröße von ca. 75 MB. Definiert wird die Variation der Farbhelligkeit mit max. $\pm$ 0.3, des Kontrasts mit max. $\pm$ 0.3, der Sättigung ebenfalls mit max. $\pm$ 0.3 und des Farbwerts mit max. $\pm$ 0.05.
Einen kurzen Auszug aus diesem ersten grundlegenden Datensatz zusammen mit der Beschreibung der Bildlabel in Form des Dateinamens enthält Abbildung 4.5.

Label in Dateiname

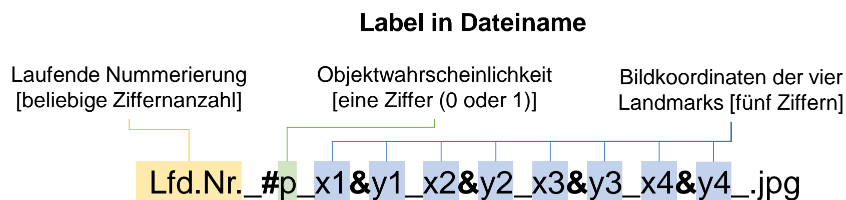


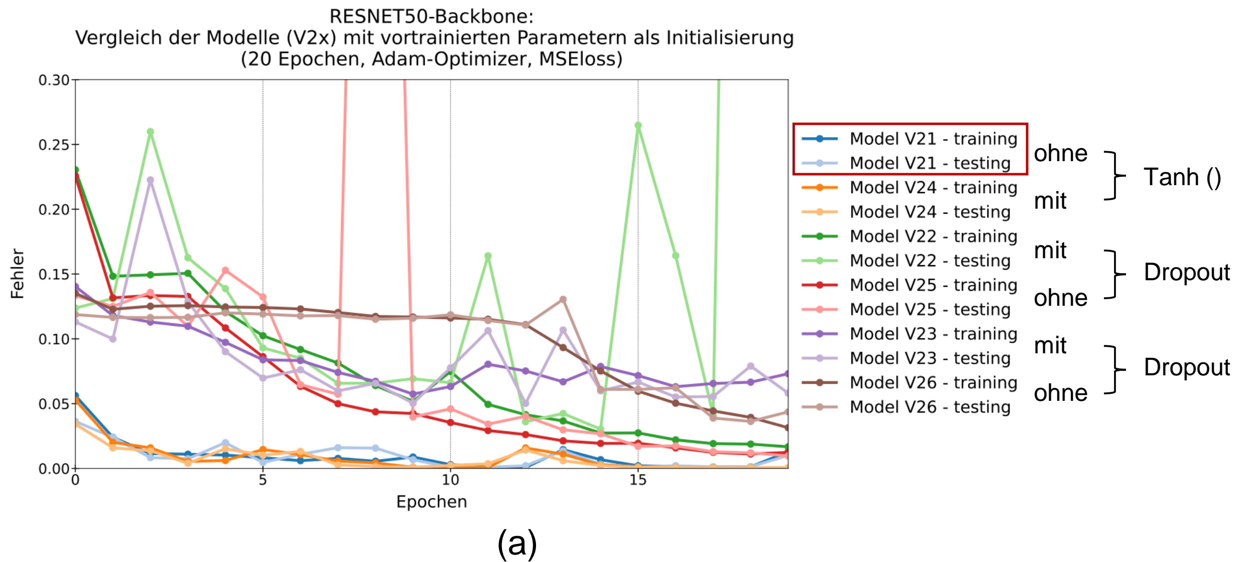
Abb. 4.5: Auszug aus dem ersten Trainingsdatensatz zur DL-basierten Landmark Detektion und Speichersystematik der Labelinformationen [eigene Darstellung]

Wie auch online auf der PyTorch-Website unter den Modellen mit vortrainierten Parametern stets angegeben, akzeptieren diese Modelle nur korrekt formatierte Bildinputs. Dementsprechend ist sowohl für das Training der Modelle als auch die Implementierung eine Preprocessing-Funktion erforderlich. Üblicherweise werden die Bilder im PIL.Image-Format eingelesen und sie liegen als HWC-Bildmatrix mit RGB-Farbraum vor. Zunächst wird die Matrix in ein Tensorobjekt mit Zahlenwerten zwischen [0,1] überführt, um ein effizientes Handling der Daten auf der GPU zu ermöglichen. Falls die Bilder nicht die richtige Größe besitzen, ist ein Resizing auf die von dem ResNet50 als auch dem EfficientNetB0-Backbone geforderte Bildgröße von 224 x 224 Pixeln durchzuführen. Die Modelle können die Bilddaten außerdem nur im CHW-Speicherlayout verarbeiten, wobei auch eine Dimensionserweiterung erlaubt ist, um mehrere Bilder gleichzeitig in einer Batch an das Modell zu übergeben (BCHW-Layout). Eine ausführliche Beschreibung der Speicherlayouts ist dem Grundlagenkapitel 2.2 zu entnehmen. Schließlich wird der RGB-Farbraum in den BGR-Farbraum überführt.

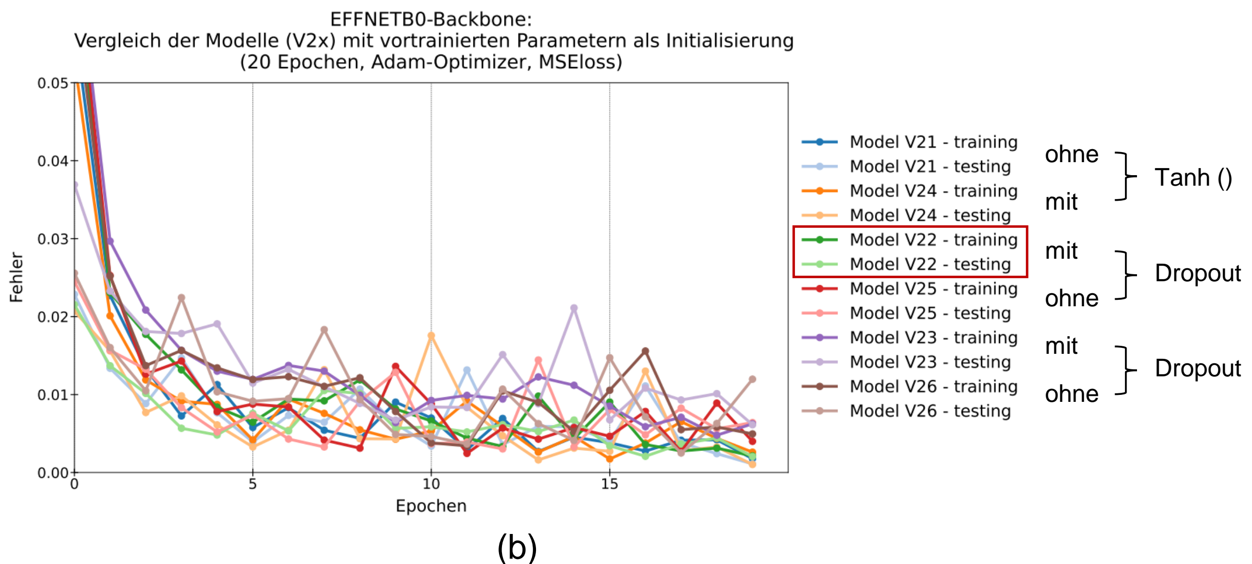
Erstes Training:
Der vorhandene Datensatz wird zufällig in dem Verhältnis 80/20 in Bilder für die Trainings- und die Testphasen unterteilt. Innerhalb dieser beiden Gruppen erfolgt zusätzlich ein Mischen der Bilder und anschließend die Zusammenfassung zu Batches mit einer Größe von 20 Bildern pro Batch. Als Hyperparameter für das erste Training festgelegt sind der Adam-Optimizer mit einer festen Lernrate von lr = 0.001 und die Verlustfunktion (engl.: loss function) MSEloss, welche den quadratischen Fehler zwischen dem Modell-Output und dem Label gemittelt über eine Batch berechnet. Bei den ResNet50-Modellen ist die Batchgröße auf 10 zu reduzieren bei Berücksichtigung aller Modellparameter während des Trainings.
Nach Analyse des Verlaufs der Verlustkurven aus dem ersten Training kann bereits eine Vorauswahl bezüglich der Transfer Learning-Methode getroffen werden. Die Modelle mit nicht trainierbaren Parametern des Backbones (V1x) zeigen zwar eine schnelle Konvergenz der Loss-Kurven, verfügen allerdings über einen größeren bleibenden Fehler als deren Pendants (V2x) mit einer Optimierung aller Modellparameter. Eine weitere Untersuchung der ersten und ressourcenschonenden Möglichkeit zur Applikation des Transfer Learnings wird demnach frühzeitig ausgeschlossen.

Zur Festlegung einer Modellarchitektur als Ausgangsbasis für eine weiterführende Hyperparameteroptimierung des Trainings der Landmark Detektion werden ausschließlich die V2x-Modelle herangezogen, vergleiche Tabelle 4.1. Die Verläufe der Verlustkurven während dieses ersten Trainings stellt Abbildung 4.6 jeweils für das ResNet50 (4.6 (a)) als auch das EfficientNetB0 (4.6 (b)) als Backbone dar. Die Modelle basierend auf dem EfficientNetB0 weisen bei allen Variationen des Detektionskopfes eine ausgeprägte Konvergenz der Trainingskurven auf und unterscheiden sich kaum untereinander. Im Gegensatz dazu sind bei der Verwendung des ResNet50 als Backbone deutliche Auswirkungen auf die Verläufe der Lernkurven bei Konfiguration der letzten Schichten zu erkennen. Festzuhalten ist, dass vor allem eine größere Tiefe des Detektionskopfes die Tendenz zum Overfitting erhöht. Wie in Abbildung 4.6 zu erkennen ist, divergieren bei der Modellvariante V22 die Kurven während des Trainings- und der Testphasen stark trotz des Einsatzes eines Dropout-Layers. Ohne Dropout tritt dieses Phänomen bereits früher auf, vergleiche Variante V25. Zwar scheinen die Varianten V23 und V26 der ResNet50-Modelle keine so ausgeprägte Tendenz zum Overfitten zu besitzen, jedoch weisen sie auch keine zufriedenstellende Lernkurve auf.

Bei den ResNet50-Modellen wird für weitere Untersuchungen die Modellvariante V21 mit einer einfachen Anpassung der Ausgabeschicht an die neun Modellausgänge festgehalten. Ausschlaggebend für diese Wahl ist die stabile Konvergenz der Lernkurve ab 15 Epochen. Die Variante V24 mit dem Tangens Hyperbolicus in der Ausgabeschicht liefert einen ähnlichen Kurvenverlauf. Die exakte Wahl der Aktivierungsfunktion ist jedoch Bestandteil der zweiten Trainingsrunde zur Hyperparameteroptimierung. Die Annahme einer stabilen Konvergenz aufgrund dieser ersten Trainingsergebnisse ist ebenfalls durch ein Training mit deutlich mehr Epochen zu verifizieren.
Die Identifikation eines geeigneten Modells für weitere Optimierungen bei der EfficientNetB0-Modellfamilie ist weniger eindeutig. Allerdings besitzen die Kurven der Variante V22 die beste Konvergenz im Vergleich zu allen anderen Trainingsverläufen und diese Konfiguration wird somit als bestes EfficientNetB0-Modell der ersten Trainingsrunde festgehalten.



(a)



(b)

Abb. 4.6: Vergleich der ResNet50- (a) und EffNetB0-Modellkonfigurationen (b) zur Landmark Detektion mit Fokus auf die Nutzung der vortrainierten Backbone-Parameter als Initialisierungsschritt (Modellvarianten V2x) [eigene Darstellung]

Zweites Training mit Hyperparameteroptimierung:
Um die Stabilität der Lernkurven besser beurteilen zu können, wird für die zweite Trainingsrunde die Anzahl der Epochen verdoppelt auf 40. Der zugrunde gelegte Trainingsdatensatz ändert sich nicht. Nach der in dem vorangegangenen Abschnitt offengelegten Vorauswahl der ursprünglich 12 Modellvarianten stehen nun zwei Modelle als Grundlage für detailliertere Optimierungsschritte zur Verfügung, vergleiche Tabelle 4.1:

- ResNet50-Modellvariante V21: ResNet_V21
- EfficientNetB0-Modellvariante V22: EffNet_V22

Dieser Abschnitt beschreibt die Ergebnisse der zweiten Trainingsrunde, welche der Hyperparameteroptimierung nach dem Rasterverfahren (engl.: Grid Search) dient. Als zu

optimierende Parameter definiert sind die Netzarchitektur (insbesondere die Wahl der Aktivierungsfunktion), der Optimizer, die Lernrate und die Verlustfunktion. Für jeden dieser Parameter sind diskrete Werte vorgegeben, welche variabel gesetzt werden, während alle anderen Parameter konstant auf ihrem Basiswert verbleiben. Die optimale Endlösung entsteht somit durch eine Kombination der optimalen Parameterwerte für jede Stellschraube. Auf eine Darstellung aller Zwischenergebnisse ist zu verzichten, da dies den Rahmen dieser Arbeit überschreiten würde. Der Fokus der folgenden Abhandlung liegt auf der Diskussion der erhaltenen Resultate.

Abbildung 4.7 zeigt für beide Ausgangsmodelle mit unterschiedlichem Backbone den optimalen Pfad bei der Einstellung der Hyperparameter des Trainings auf. Zu beachten ist, dass der Ergebnisraum hier durch die manuelle Vorgabe von diskreten Parameterwerten statt eines Wertebereichs stark begrenzt ist. Dementsprechend kann eine Existenz besserer Lösungen außerhalb des hier betrachteten diskreten Raums nicht ausgeschlossen werden. Vor einer Implementierung eines DL-Modells im Realfahrzeug ist ein automatisiertes Feintuning der Hyperparameter vorzunehmen, um die Performance des neuronalen Netzes auf ein bestmögliches Niveau anzuheben. Für die prototypischen Untersuchungen mithilfe des Fahrroboters ist diese grobe manuell geführte Optimierung des Trainings der Detektionsmodelle jedoch ausreichend.

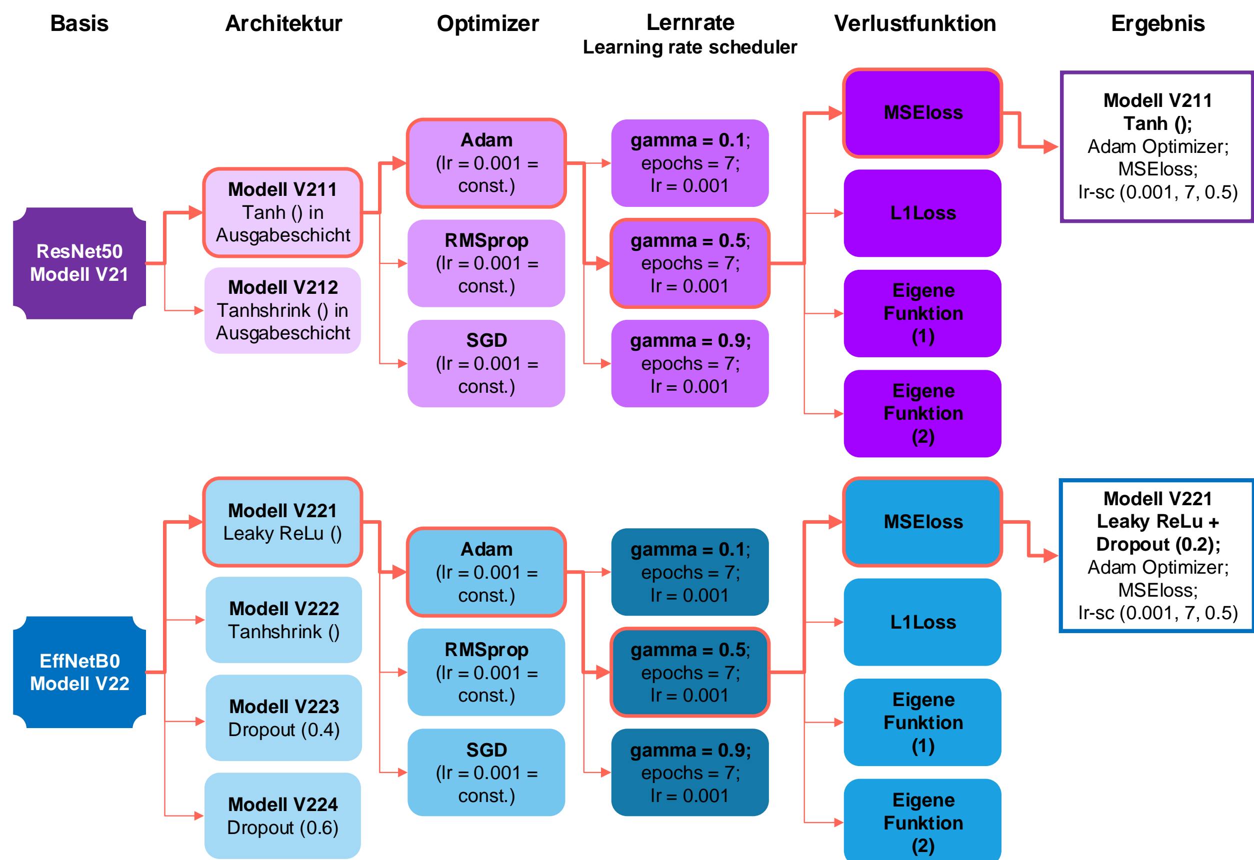


Abb. 4.7: Manuelle Hyperparameteroptimierung der besten Modellvarianten aus erster Trainingsrunde [eigene Darstellung]

Betrachteten werden jeweils drei Optimizer (Adam, RMSprob und SGD) und vier unterschiedliche Verlustfunktionen. Als Alternative zu dem gemittelten quadratischen Fehler,

welcher vor allem großen Abweichungen zwischen der Ground Truth und dem Modelloutput Rechnung trägt, bietet sich der gemittelte absolute Fehler durch Anwendung der L1Loss-Funktion an. Ergänzt werden diese standardisierten Funktionen durch zwei eigenständig entwickelte Verlustfunktionen. Die erste Version dieser Funktion bestimmt den über eine Batch gemittelten absoluten Fehler der Objektwahrscheinlichkeit und bei den Bildkoordinaten die Summe der euklidischen Abstände jedes Markierungspunkts ($\Delta d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$) ebenfalls über eine Batch gemittelt. Schließlich folgt eine Addition der beiden gemittelten Werte. Version 2 unterscheidet sich hiervon durch die Berechnung des gemittelten quadrierten Fehlers bei der Objektwahrscheinlichkeit statt des absoluten Fehlers. Die Lernkurven mit den eigenen Loss-Funktionen zeigen jedoch, dass die Netze nicht ordnungsgemäß trainieren, also kaum eine Verringerung des Fehlers im Trainingsverlauf aufweisen, wodurch von einer weiteren Verwendung dieser Funktionen abgesehen wird.
Einen sehr einflussreichen Trainingsparameter stellt die Lernrate dar. Sie legt fest, in welchem Ausmaß die Modellparameter in jeder Trainingsepoche verändert werden. An Stelle der Variation von konstanten Lernraten, wird bei dieser Optimierung auf einen Lernraten-Scheduler zurückgegriffen, welcher eine ausgeprägtere Konvergenz der Lernkurven durch Unterdrückung von Ausreißern erwarten lässt. So wird die anfängliche Lernrate (“lr”) nach einer definierten Anzahl abgeschlossener Trainingsepochen (“epochs”) mit einem Vorfaktor (“gamma”), welcher typischerweise zwischen Null und Eins liegt, multipliziert und schrittweise reduziert. Wird die Lernrate jedoch zu früh oder zu stark während des Trainings eingeschränkt, kann die vorhandene Lernfähigkeit des Netzes nicht vollständig genutzt werden. Die hier festgelegte Auswahl der Funktionswerte für den Lernraten-Scheduler basiert auf Erfahrungswerten und Beispielen aus der Literatur oder anderen veröffentlichten Projekten.
Nachdem die erste Trainingsrunde eher den Fokus auf die Definition einer geeigneten groben Netzarchitektur legt, ist das Feintuning dieser ersten Modelle Inhalt der zweiten Trainingsrunde. Bei dem ResNet_V21 erfolgt ein Vergleich zwischen der Tanh() und der Tanhshrink() Funktion als Aktivierung der Ausgabeschicht. Das Modell EffNet_V22 mit dem mehrschichtigen Detektionskopf bietet hier mehr Freiheitsgrade. Für eine Analyse ausgewählt wurde als Alternative zu dem Tangens Hyperbolicus als Aktivierungsfunktion ebenfalls Tanhshrink() und Leaky ReLu(). Weiterhin wird die Dropout-Rate in 0.2-Schritten variiert von 0.2, auf 0.4 und schließlich auf 0.6.

Bei beiden Basismodellen kristallisieren sich die MSEloss-Funktion, der Adam-Optimizer als auch der Lernraten-Scheduler mit gamma = 0.5 als optimale Hyperparameter für das überwachte Lernen heraus. Das aus dieser Optimierung hervorgehende und trainierte ResNet50-Modell wird als Version V211 referenziert (ResNet_V211), vergleiche Abbildung 4.7. Von der Tanh()-Aktivierungsfunktion profitiert das EffNetB0-Modell allerdings nicht. Hier zeigen sich bessere Trainingsergebnisse bei der Verwendung der Leaky ReLu() Funktion zusammen mit einer Dropout-Rate von 0.2 im Detektionskopf. Dieses Modell erhält die Bezeichnung V221 (EffNet_V221). Als vorläufiges Ergebnis stehen zwei unterschiedliche trainierte Modelle zur Erfüllung der Landmark Detektion basierend auf einzelnen Bildern einer Kamera mit Fish-Eye-Linse bereit. Den größten Unterschied dieser beiden vorgestellten Modelle stellt die Größe des Parametersatzes in Form der Kantengewichte dar. Dieser muss später im Fahrzeug nicht nur in einem nicht flüchtigen Programmspeicher abgelegt werden, sondern ist auch entscheidend für die benötigte Rechenzeit des Modells. Besonders bei Echtzeitanwendungen ist die Ausführungszeit sol-

cher komplexer Modelle mit mehreren Millionen Parametern ein hochsensibles Kriterium. Dieser Aspekt könnte schließlich auch ausschlaggebend für die Wahl zwischen dem eher schweren ResNet_V211 und dem leichten EffNet_V221 sein.
Die Lernkurven der erarbeiteten Modelle sind in Abbildung 4.8 gegenübergestellt. Trotz der unterschiedlichen Parameterumfänge und Netzarchitekturen weisen beide nahezu identische Trainingskurven mit einer äußerst stabilen Konvergenz unterhalb eines Fehlers von 0,005 bereits nach 15 Epochen auf. Das ResNet_V211 benötigt für das Training auf dem 5192 Bilder großen Datensatz und unter Berücksichtigung aller Modellparameter bei durchgehend fast voller Ausnutzung der verfügbaren GPU (NVIDIA T1000) 64 min. Abgespeichert werden die Kantengewichte in Form einer .pth-Datei des Parametersatzes, welcher während einer Testphase innerhalb des Trainings den minimalen Fehler (hier: 0.000070) produziert. Im Gegenzug dazu beansprucht das EffNet_V221 nur ungefähr die halbe Zeit mit 37 min Trainingsdauer und einer ebenfalls etwas geringeren durchgängig konstanten Auslastung der GPU von ca. 70 %. Die besten Kantengewichte werden bei dem minimalen Testfehler von 0,002 festgehalten.

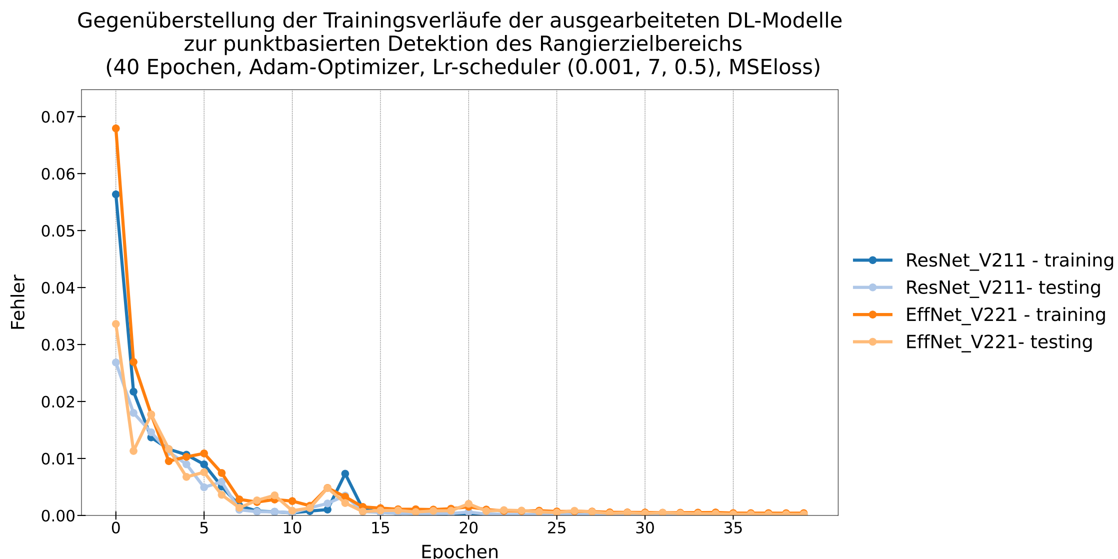


Abb. 4.8: Gegenüberstellung der Lernkurven der optimierten Modelle basierend auf dem ResNet50- und dem EfficientNetB0-Backbone [eigene Darstellung]

Evaluation und Verifikation der Ergebnisse:
Nach ersten Tests der Detektionsmodelle unter Realbedingungen auf der Hardware des Fahrroboters werden bei den trainierten DL-Modelle vor allem Defizite bei der korrekten Lokalisierung der Landmarks im Kamerabild bei größeren Distanzen (> 30 cm) festgestellt. Dieses Verhalten lässt darauf schließen, dass der bisher verwendete Trainingsdatensatz zu wenige Beispiele der Markierungen, betrachtet aus der Ferne, aufweist. Um die Vielfalt des Datensatzes zu erhöhen und im Zuge dessen auch den Einfluss der Qualität der bereitgestellten Daten auf die Leistungsfähigkeit der hier vorgestellten neuronalen Netze zu evaluieren, erfolgt eine Erweiterung des Datensatzes mit Fokus auf die Abbildung von Szenen mit größeren Entfernungen. Hierzu werden die 1298 ursprünglichen Bilder des ersten Datensatzes mit weiteren 2013 neu aufgenommenen Bildern in einem zweiten umfangreicheren Trainingsdatensatz vereint. Auch hier erfolgt eine einfache Aufweitung der Daten durch dreimalige Anwendung eines *Color Jitter* auf alle 3311 Bilder. Somit stehen nun mehr als doppelt so viele Trainingsbilder (insgesamt 13244 Bilder) für

das Transfer Learning der DL-Modelle zur Verfügung. Um beide Datensätze unterscheiden zu können, erhalten sie eine Namensgebung nach ihrem finalen Erstellungsdatum. Der im vorangegangenen Abschnitt zugrunde gelegte Datensatz wird im Folgenden als "Datensatz 13.11.2022" und die zu einem späteren Zeitpunkt erweiterte Bildersammlung als "Datensatz 07.12.2022" ausgewiesen.
Wie in Kapitel 2.3 bereits angesprochen, sind der Umfang als auch die Vielfalt an gelabelten Daten für das überwachte Lernen von tiefen neuronalen Netzen entscheidende Attribute der bereitgestellten Datensätze. Obwohl die Sammlung der Bilder eher nach dem Zufallsprinzip und ohne vordefinierte Muster erfolgt, besitzen beide Datensätze eine nahezu ausgewogene Verteilung von Positiv- und Negativbeispielen (Objektwahrscheinlichkeit 1 oder 0). Ein weiterer wichtiger Aspekt ist die Verteilung der Positionen der Landmarks über dem Bildbereich, welche durch die Diagramme in Abbildung 4.9 veranschaulicht wird. Zur Wahrung der Übersichtlichkeit werden hierzu nicht alle vier Punkte für die Auswertung herangezogen. Anhand der in den Labeln enthaltenen Bildkoordinaten wird der Mittelpunkt der Fläche, definiert durch die vier Landmarks, berechnet und die absolute Häufigkeit dessen Lage im Bild bei beide Datensätze evaluiert. Entgegen der Vermutung einer weitläufigeren Streuung der Mittelpunkte über die gesamte Bildebene unterstreicht der Datensatz vom 07.12.2022 die Tendenz zur Agglomeration der Punkte vor allem zentral in der oberen Bildhälfte. Es sind bei beiden Diagrammen zwei dominante Streifen zu sehen. Sie resultieren aus der Berücksichtigung beider Fälle zur Anbringung der gelben Markierungen. Der obere Streifen repräsentiert somit die Mittelpunkte der Zielflächen, dargelegt durch senkrecht an einem feststehenden Objekt angebrachte Markierungen. Hingegen führt die Platzierung der Markierungen auf dem Boden zu der Verteilung der Mittelpunkt wie in dem unteren Streifen zu erkennen. Die Krümmung zum Bildrand hin ist auf die Nichtlinearitäten des Abbildungsprozesses der Kamera mit Fish-Eye-Linse zurückzuführen. Die Konzentration der Mittelpunkte vor allem in der oberen Bildhälfte ist durch die um 75° zur Bodenfläche geneigter Kamera zu erklären. Der Fluchtpunkt aller parallelen Linien befindet sich somit auch in der oberen Bildhälfte und eine gleichmäßige Verteilung der resultierenden Mittelpunkte über der gesamten Bildebene ist kaum zu erzielen. Besonders durch die Vorgabe während des Labeling-Prozesses, dass die Markierungen vollständig im Bild zu sehen sein müssen, um die vier markanten Punkte zu bestimmen, ist eine vermehrte Häufung der Mittelpunktlage in der unteren Bildhälfte nicht erreichbar.

Als Bewertungsmaßstab der favorisierten Modelle wird eine Konfusionsmatrix herangezogen. Jedoch ist für eine Verifikation der Ergebnisse nicht nur die korrekte Erkennung der Markierung als Objekte im Bild von Bedeutung, sondern vordergründig die Genauigkeit der Lokalisierung der Landmarks. Es wird in dieser Arbeit spezifiziert, dass ein Modell für die anwendungsspezifische Objektdetektion den Anforderungen an die Perception-Task genügt, wenn der aus den vier Landmarks gebildete Flächenmittelpunkt einen euklidischen Abstand zwischen Vorhersage und Ground Truth von weniger als zwei Pixeln im Bild in mehr als 90 % der Fälle aufweist. Hierfür werden die Modelle auf Grundlage der Testdaten, welche zufällig gewählte 20 % der Trainingsdaten darstellen, analysiert und verglichen. Die Resultate dieser Modellevaluierung fasst Tabelle 4.2 zusammen. Bei beiden Modellausführungen ist zu erkennen, dass der deutlich größere Trainingsdatensatz ("07.12.2022") zu einer Verbesserung der Modellvorhersagen führt. Sowohl die Abweichungen bei den einzelnen Landmarks als auch der mittlere absolute Fehler bei der Ausgabe der Objektwahrscheinlichkeit nehmen ab. Weiterhin ist der Tabelle zu entnehmen, dass die Lage des Flächenmittelpunktes durch die Modelle genauer approximiert werden

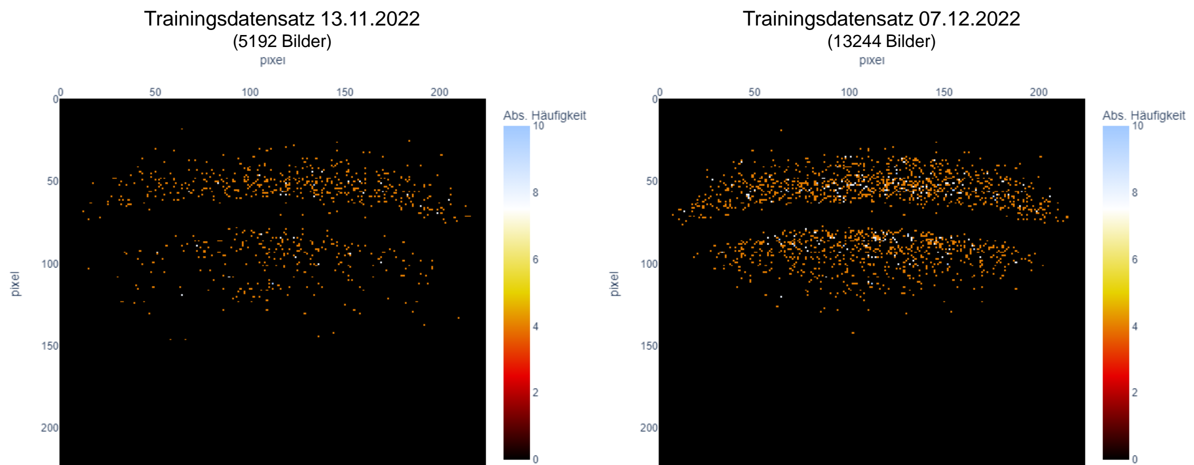


Abb. 4.9: Absolute Häufigkeiten der Lage der Flächenmittelpunkte in den Trainingsdaten, jeweils festgelegt durch die vier Landmarks [eigene Darstellung]

kann als die Koordinaten der einzelnen Landmarks. Da dieser Mittelpunkt selbst eine Mittelung der x- und y-Koordinaten der Landmarks darstellt, dämpft er in gewissem Maße Ausreißer bei der Regression der Landmarks ab.

Als verifiziertes Artefakt dieser Entwicklungsphase ist schließlich das Modell V211 mit dem ResNet50-Backbone hervorzuheben. Es erfüllt die Vorgabe einer möglichst genauen Ausgabe des Flächenmittelpunkts in mindestens 90 % der wahrheitsgemäßen Objektdetektionen (TP i.O.). Durch das Training mit dem umfangreicheren Datensatz lässt sich dieses Ergebnis noch weiter auf ein "i.O.-Ergebnis" in 97,4 % der TP-Fälle verbessern, vergleiche Tabelle 4.2. Im Gegensatz dazu ist für die in Kapitel 5 durchzuführende Integration der Funktionsbausteine das auf dem EfficientNetB0 basierende Modell EffNet_V221 als ungeeignet für die hochgenaue Landmark Detektion zu kategorisieren.

Zusammenfassend steht nach der Erarbeitung einer Lösung für die essentielle Perception-Task, welche das Fundament für alle folgenden Funktionsbausteine darstellt, ein unter Zuhilfenahme von Transfer Learning überwacht trainiertes als auch verifiziertes DL-Modell mit einem ResNet50-Backbone und einer einfachen Ausgabeschicht aktiviert mit der Tanh()-Funktion zur Verfügung. Es erfüllt unter ersten statischen Tests die Anforderungen an eine hochgenaue Detektion und Lokalisierung von vier Landmarks in den Farbbildern einer unkalibrierten Weitwinkelkamera. Eine Validierung dieses Modells ist jedoch erst zu einem späteren Zeitpunkt im Zusammenspiel mit den weiteren Funktionsbausteinen durchführbar.

Tabelle 4.2: Evaluation und Verifikation der entwickelten DL-Modelle zur Landmark Detektion basierend auf den jeweiligen Testdatensätzen der Trainingsdatensammlung

Bewertungs-kriterium	ResNet_V211 (13.11.2022)	ResNet_V211 (07.12.2022)	EffNet_V221 (13.11.2022)	EffNet_V221 (07.12.2022)
True positive (TP)	613	1456	613	1455
True negative (TN)	425	1192	425	1191
False positive (FP)	0	0	0	1
False negative (FN)	0	0	0	1
Mittlerer abs. Fehler Objektwahrscheinlichkeit	0.0013	0.0004	0.0059	0.0045
Mittlere Abweichung LD01 [Pixel]	0.8984	0.5640	1.1984	1.0545
Mittlere Abweichung LD02 [Pixel]	0.8988	0.5796	1.1881	1.0732
Mittlere Abweichung LD03 [Pixel]	0.7738	0.5963	1.1493	1.0486
Mittlere Abweichung LD04 [Pixel]	0.7904	0.5488	1.2592	1.0209
Mittlere Abweichung Mittelpunkt [Pixel]	0.6714	0.4206	0.8244	0.7280
TP i.O.	554 (90 %)	1418 (97,4 %)	497 (81 %)	1210 (83 %)
TP n.i.O.	59 (10 %)	38 (2,6 %)	116 (19 %)	245 (17 %)

4.2 Steuerung

Da die Entwicklung geeigneter Regelungs- und Steueralgorithmen eine eigenständige Fachdisziplin im Ingenieurbereich darstellt, ist der Anspruch dieses Kapitels nicht in der Erarbeitung eines vollständig neuen Steuerungskonzeptes für die Automatisierung der Fahraufgabe zu sehen. Für die Funktionsentwicklung mithilfe des miniaturisierten Fahrroboters sind bereits einfache und weit verbreitete Algorithmen ausreichend, um die Steuerungskomponente in der gesamten Funktionssoftware abzubilden.

Zunächst ist festzuhalten, welche Größen innerhalb des verwendeten Fahrroboters zugänglich und somit beeinflussbar sind. Der Roboter verfügt neben der exteroperceptiven Sensorik in Form der Weitwinkelkamera über keine weiteren Sensoren, abgesehen von der Sensorik innerhalb des Entwicklerboards, wie z.B. der Temperatursensor. Im Gegensatz zu realen Straßenfahrzeugen stehen bei dem Fahrroboter keinerlei Odometriedaten zur Verfügung, welche eine Zustandsschätzung des dynamischen abgeschlossenen Systems erlauben. Aufgrund dieser hardwaretechnischen Einschränkungen sind in erster Linie Steuerungskonzepte mit ihrem offenen Wirkungsablauf vor allem bei der Antriebsregelung in den Fokus zu rücken. Eine direkte Regelung interner Größen des Roboters ist ausgeschlossen, da hierzu schlicht die Erfassung der dedizierten Regelgrößen über eine Sensorik zur Gegenkopplung fehlt.

Neben der Regelung und Steuerung der Subsysteme des Fahrzeugs ist der allgemeine Fahrvorgang ebenfalls als Regelkreis modellierbar. Hierbei fungiert traditionell der menschliche Fahrer als kontinuierlicher Regler im Gesamtsystem. Dieser nimmt Einfluss auf das dynamische Fahrzeugverhalten, welches Umwelteinflüssen und anderen Störgrößen ausgesetzt ist. Eine Rückmeldung über den tatsächlichen Zustand, in dem sich das Fahrzeug befindet, erfolgt zumeist über die visuellen, akustischen oder haptischen Sinneskanäle des Menschen. Innerhalb von Sekundenbruchteilen verarbeitet das menschliche Gehirn diese Informationen und vergleicht sie mit dem gewünschten Fahrzustand, um schließlich die Einstellung der Stellgrößen zu korrigieren.
Wird die Betrachtung der Fahrzeugdynamik rein auf die Längs- und Querdynamik beschränkt, ergeben sich grundsätzlich das Antriebs- bzw. Bremsmoment an den Rädern und der Lenkwinkel als wichtige Stellgrößen. Über das Lenkrad und die Pedale kann der Mensch in heutigen Fahrzeugen in das Geschehen eingreifen. Mit zunehmender Automatisierung des Fahrvorgangs ist ein Regelungssystem vorzusehen, welches den menschlichen Fahrer ablöst.
Auch auf die Gestaltung der Regelung der übergeordneten Fahraufgabe hat die Limitierung bei der verfügbaren Hardware des Fahrroboters, wie in Kapitel 2.4 aufgeführt, gewisse Einflüsse. Da die beiden angetriebenen Räder beide einen eigenen radnahen Gleichstrommotor besitzen, lässt sich ein Lenkwinkel durch unterschiedliche Ansteuerung des linken und des rechten Motors einstellen. Für eine reine Geradeausfahrt sind für beide Motoren gleiche Drehzahlen vorzugeben. Durch eine einfache elektronische Brückenschaltung ist eine Umpolung der Versorgungsspannung der E-Motoren möglich, wodurch eine Änderung der Drehrichtung jederzeit möglich ist.

Vor allem im Zusammenhang mit der Trajektorienplanung und -verfolgung sind geeignete Regelungsalgorithmen gefragt, welche eine selbstständige Kurshaltung gewährleisten.
In der Literatur werden überwiegend modellbasierte Fahrzustands- und Positionsregelungen für autonome Fahrzeuge diskutiert [72, S. 102-104] [73, 74]. Durch eine modellprädiktive Regelung (engl.: Model Predictive Control, MPC) lassen sich optimale Stellgrößen zum Folgen einer Referenztrajektorie für einen gewissen vorausliegenden Zeithorizont berechnen und durch Rückkopplungen optimieren. Je komplexere Fahrdynamikmodelle Verwendung finden, desto höher ist der Rechenaufwand dieser Regelstrategie. Zusätzlich ist die Parametrisierung der eingesetzten Modelle durchzuführen und deren Validität zu gewährleisten. Dai et. al (2020) schlagen für die Regelung des Lenkwinkels eine Gleitregimeregelung mit Fuzzy-Logik (engl.: Fuzzy Sliding Mode Control, F-SMC) vor [73]. Da eine Modellierung des dynamischen Verhaltens des Fahrroboters nicht zielführend für die prototypischen Untersuchungen ist, werden im Folgenden einfache und in der Praxis weit verbreitete stetige lineare Regler erörtert. Dazu zählen der P-Regler, der PD-Regler und schließlich der PID-Regler als Universalregler. Nach Schwarting et. al (2018) eignet sich ein einfacher PID-Regler bereits zur Trajektorienverfolgung [75].

Auf dem Rapid Prototyping-Tool können mit einem einfachen P-Regler bereits Strategien zur Objektverfolgung umgesetzt werden. In dem öffentlich zugänglichen Beispiel “Object Following” wird auf Basis eines DL-Modells zur Objektdetektion eine solche Steuerstrategie aufgezeigt. Als Regelgröße wird die x-Koordinate des Zentrums der Bounding-Box des Zielobjekts definiert. Soll dieses Ziel zentral angesteuert werden, muss sich die x-Koordinate möglichst im Bereich der vertikalen Bildmittellinie bewegen. Wird der Ursprung des Bildkoordinatensystems in den Bildmittelpunkt gelegt und die Koordinaten auf

die Bildbreite (respektive die Bildhöhe) normiert, gibt das Detektionsmodell bereits die Regelabweichung der x-Koordinate des Objektzentrums von der Führungsgröße x = 0 aus. Nach Multiplikation mit einem kontinuierlich einstellbaren Verstärkungsfaktor werden die Drehzahlen der beiden Motoren entsprechend angepasst. Die Verwendung eines konstanten Reglerparameters ermöglicht ein gewünschtes progressives Verhalten. Bei kleinen Regelabweichungen fällt der Eingriff des Reglers gering aus. Große Abweichung führen automatisch zu einer stärkeren Reaktion des Reglers zusammen mit der Tendenz zu ausgeprägten Überkompensationen bzw. Überschwingern. Abhängig von der Aktualisierungsrate der Kamera erfolgt eine Anpassung der Steuergröße der Motoren in quantisierten Zeitschritten und nicht kontinuierlich. Das vorgestellte Steuerungskonzept wird anhand Abbildung 4.10 verdeutlicht. Dessen Eignung für den vorliegenden Anwendungsfall ist im Rahmen des Aufbaus der Version 1 der Rangierfunktion zu überprüfen, vergleiche Kapitel 5.1.

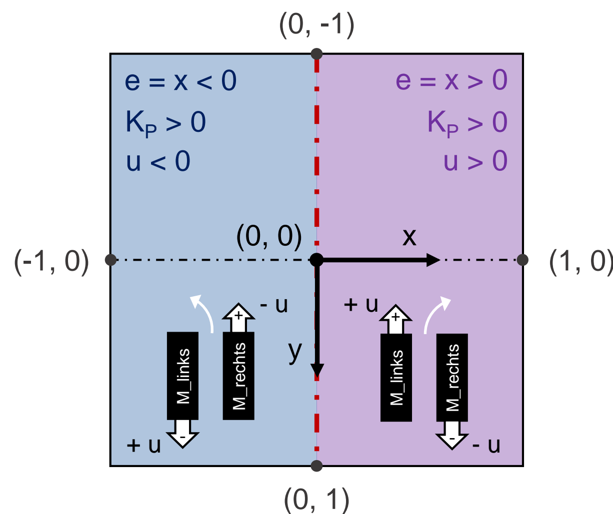


Abb. 4.10: Einfache proportionale Antriebssteuerung des Fahrroboters basierend auf der horizontalen Abweichung des Zielobjekts von der Bildmittellinie (x = 0) [eigene Darstellung]

Eine Parallelschaltung des P-Gliedes mit einem Differentialanteil (D-Glied) sorgt für Verbesserungen während des dynamischen Einschwingvorgangs. Sowohl die Überschwingweite als auch die benötigte Ausregelzeit können bei Verwendung eines PD-Reglers deutlich reduziert werden. Jedoch verbleibt im stationären Zustand bei diesem Reglertyp theoretisch ebenfalls eine gewisse Regelabweichung [9].
Verwendung findet dieser Reglertyp ebenfalls in einem Beispiel-Notebook, welches auf dem SDK-Image des JetBots enthalten ist. In der Live-Demo der beispielhaften automatisierten Fahraufgabe “Road Following” wird die konstante Vorwärtsgeschwindigkeit beider Motoren über einen Schieberegler eingestellt. Ein neuronales Netz wird darauf trainiert, einen Bildpunkt als Zielobjekt mit seinen beiden Koordinaten in der Bildebene anhand der Interpretation der Umgebung zu bestimmen. Aus der bekannten Position dieses Referenzpunktes im Bild ist der eingeschlossene Winkel mit der vertikalen Bildmittellinie über den elementweisen Arkustangens (in Python: *numpy.arctan2 (x,y)*) abzuleiten. Der Winkel wechselt dadurch sein Vorzeichen bei Übergang zwischen beiden Bildhälften, ähnlich wie der proportionalen Regelung der Abweichung in x-Richtung, vergleiche hierzu Abbildung 4.10. Durch die Festlegung des Winkels als Regelgröße wird hier auch der Abstand

zur Bildhorizontalen in y-Richtung miteinbezogen. Ist der Referenzpunkt gedanklich in y-Richtung mit einem gewissen Betrag und mit x = const. verschoben, verkleinert sich der eingeschlossene Winkel bei steigendem Abstand Δy zum Bildmittelpunkt. Eine Festhaltung der y-Koordinate bei wachsender Abweichung in x-Richtung hat genau den gegenteiligen Effekt. Allgemein formuliert führen Referenzpunkte nahe der vertikalen Bildränder zu großen Lenkvorgaben, wodurch eine Progressivlenkung realisiert werden kann. Außerdem rufen Punkte nahe der Bildhorizontalen tendenziell stärkere Reglereingriffe hervor als weiter entfernte Punkte in y-Richtung. Die Applikation eines PD-Regelverhaltens ist ebenfalls bei der Integration der Funktionsbausteine in Betracht zu ziehen, siehe Kapitel 5.

Eine Erweiterung des PD-Reglers mit einem integrierenden Glied (I-Glied) ist als universaler PID-Regler bekannt. Das I-Glied führt zu einer weiteren Optimierung der Überschwingweite. Vor allem ist mit dem PID-Regler im Gegensatz zu den bisher vorgestellten Regelungskonzepten eine Ausregelung der Regelgröße auf den stationären Sollwert möglich [9].
Aufgrund der hohen geforderten Positioniergenauigkeit zur Erfüllung der hier behandelten Fahraufgabe wird besonders eine Implementierung eines PID-Reglers im Rahmen der Version 2 der Fahrfunktion zur Ansteuerung der Aktuatorik angestrebt, vergleiche Kapitel 5.2. Vor allem bei Vorgabe einer Referenztrajektorie ermöglicht dieser Regelungsalgorithmus theoretisch eine vollständige Ausregelung der Abweichung von dieser Bahn. Auch die relativ geringe Überschwingtendenz ist im Hinblick auf ein möglichst stabiles Fahrverhalten und einen hohen Komfort vorteilhaft. Bei Bedarf lassen sich auch die beiden anderen Reglertypen umsetzen durch Ab- oder Zuschalten des D- oder I-Glieds.
Abbildung 4.11 rundet die vorangegangene Diskussion der in Betracht gezogenen Reglertypen durch eine visuelle Gegenüberstellung des dynamischen und statischen Systemverhaltens eines beispielhaften Regelkreises ab. Die Kurven der relevanten Regler (P-, PD- und PID-Regler) sind farbig hervorgehoben.

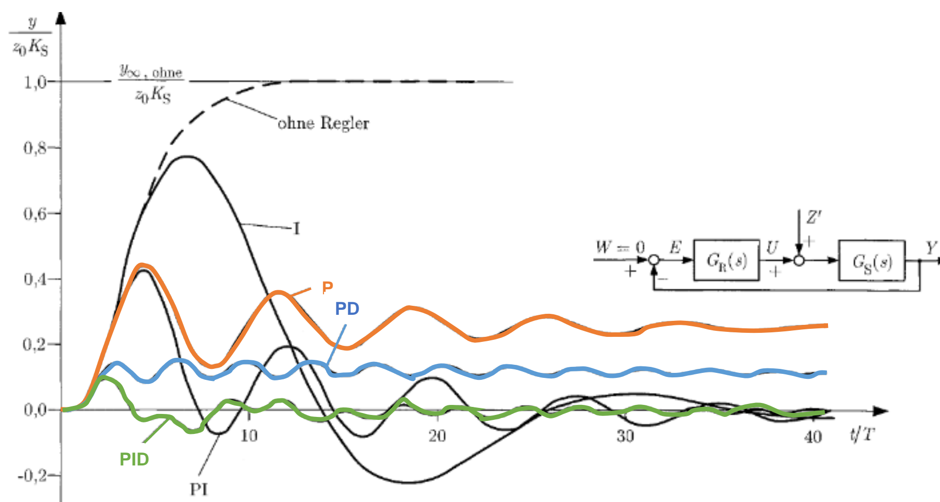


Abb. 4.11: Normierte Regelgröße eines Regelkreises mit sprungförmiger Störung unter Variation der eingesetzten Reglertypen bei gleichbleibender Regelstrecke mit der Übertragungsfunktion $G_S(s) = 1/(1+T_s)^4$ [9, S. 129]

Die bisherige Erläuterung der unterschiedlichen Reglertypen ist eher allgemein gehalten, da sich je nach Version der Rangierfunktion die Regelgröße und somit auch die Berech-

nung der Regelabweichung unterscheiden kann. Beide im Rahmen dieser Arbeit erstellten Versionen (vergleiche Kapitel 5) benötigen als Ausgangsgröße des Reglers einen Wert, welcher eine Drehzahldifferenz der Motoren beschreibt. Diese vorzeichenbehaftete Zahl wird je Motor entweder addiert oder subtrahiert. Es ist außerdem zu empfehlen ein Steuer-Bias einzuführen, um mögliche ungleiche Ansprechcharakteristiken der E-Motoren oder Bodenhaftungen der Räder auszugleichen.

Zu betrachten ist abschließend ein Konzept zur Regelung von Mehrgrößensystemen. Prinzipiell eignen sich bei dem Fahrroboter als Regelgrößen die Fahrzeugposition in der Ebene (ohne Vertikaldynamikbetrachtungen), die Geschwindigkeit als auch die Kameraorientierung. Mit der begrenzten Sensorik in Form der frontal angebrachten Weitwinkelkamera lassen sich diese Größen über visuelle Odometrie rückkoppeln und somit regeln. Die Ausarbeitung der visuellen Odometrie basierend auf Bildern einer monokularen Kamera wird in dem anschließenden Kapitel behandelt.
Ist es erforderlich, mehrere Größen in die Regelung und die daraus resultierende Ansteuerung der Motoren einzubeziehen, wird folgendes Konzept vorgeschlagen: Jede Größe besitzt ihre eigene Führungsgröße mit dieser sich unter Differenzbildung eine Regelabweichung ergibt. Eine Addition dieser einzelnen Regelabweichung noch vor dem Regler ist unbedingt zu vermeiden. Entstehen beispielhaft bei einem Zweigrößensystem zwei betragsmäßig gleiche Regelabweichungen mit gegensätzlichen Vorzeichen würden sich diese gegenseitig auslöschen und der Regler erhält als Eingang eine Regelabweichung e = 0, was jedoch nicht der Realität entspricht, da beide Regelgrößen offensichtlich eine Abweichung von ihrem Sollwert aufweisen. Aus dieser Überlegung resultiert als Lösung eine parallele Verarbeitung der Regelabweichungen e_n im Mehrgrößensystem und eine Linearkombination der einzelnen Reglerausgänge unter Einbezug von Wichtungsfaktoren g_n, da für die Motoren eine eindimensionale Stellgröße u gefragt ist. Verdeutlicht wird dieses Vorgehen durch das Blockschaltbild in Abbildung 4.12.

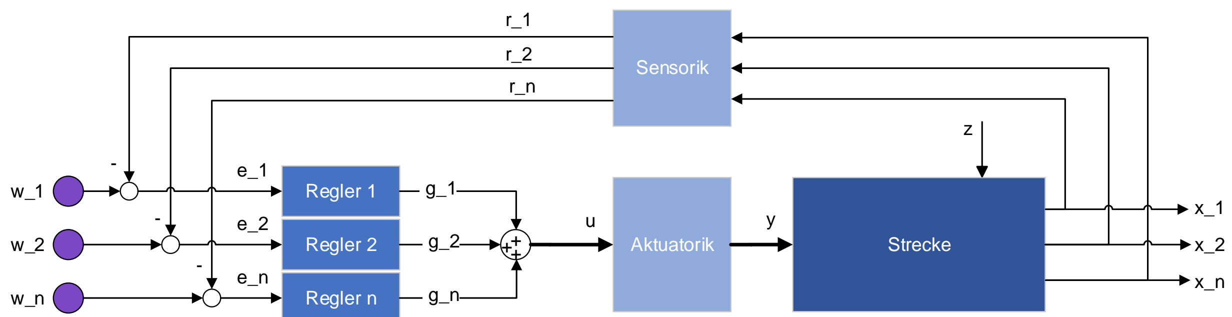


Abb. 4.12: Schematische Darstellung eines Regelungskonzepts bei Mehrgrößensystemen [eigene Darstellung]

Ein in der Praxis üblicher Reglerentwurf zur Definition der Reglerparameter über Stabilitätsuntersuchungen des Regelkreises ist bei der Implementierung der Funktion auf dem Fahrroboter nicht möglich, da hierzu ein dynamisches Modell der Regelstrecke fehlt und die Verhaltensmodellierung nicht Inhalt dieser Arbeit ist. Somit ist bei der Parameterwahl auf Erfahrungswerte und gezieltes, versuchsweises Setzen der Parameter zurückzugreifen.

4.3 Lokalisierung

Für die exakte Ausführung und Steuerung der Fahrzeugbewegung auf Basis der erkannten Referenzpunkte des Zielbereichs ist die Lage des Roboters relativ zu der im Raum markierten Fläche zu bestimmen. Dieses Kapitel behandelt Konzepte zur Lokalisierung des Ego-Fahrzeugs in einem räumlichen Kontext anhand rein visueller Informationsquellen. Zu Beginn wird eine Lokalisierungsstrategie aufgeführt, welche sich alleinig auf die geometrische Anordnung der vier Referenzpunkte (Landmarks) in einem Kamerabild stützt. Die Entwicklung und das Training von DL-Modellen zur Regression dieser Bildpunkte wurden in Kapitel 4.1.2 erörtert. Als Basis der folgenden Erläuterung ist somit davon auszugehen, dass durch vorheriges Anwenden eines solchen Modells bei jedem neuen Kamerabild die vier Eckpunkte des zu befahrenden Bereichs mit ihren jeweiligen Bildkoordinaten vorliegen.
Abschließend wird in Kapitel 4.3.2 die Lösung der Lokalisierungsaufgabe mithilfe der Ableitung visueller Odometriedaten vorgestellt. Dieses Vorgehen ermöglicht schließlich auch die gewünschte Positionsschätzung des Fahrroboters im dreidimensionalen Welt-KS.

4.3.1 Kontexterfassung aus Bildpunktepositionen

Als Motivation für diesen Abschnitt dient die effiziente Nutzung bereitgestellter Informationen der Perception-Softwareschicht bei gleichzeitig möglichst geringer Komplexität der eingesetzten Algorithmen. Somit wird zunächst angestrebt, ein Lokalisierungskonzept nur auf Basis der innerhalb der Automatisierungssoftware zur Verfügung gestellten Größen aufzubauen.
Grundsätzlich lassen sich anhand der Lage der vier Landmarks im Bild weitere Punkte ableiten. Von Interesse ist neben dem Mittelpunkt der definierten Fläche ebenso der Mittelpunkt der “vorderen” Begrenzungslinie, welche den Einfahrtsbereich festlegt. Wie in Kapitel 4.1.2 beschrieben, sind die Referenzpunkte entgegen dem Uhrzeigersinn beginnend in der unteren linken Ecke durchnummeriert. Eine koordinatenweise Erschließung des Flächenmittelpunkts mittelt die Koordinaten jeweils zweier benachbarter Eckpunkte, um die Kantenmittelpunkte zu erhalten. Eine weitere Mittelung zweier gegenüberliegender Kantenmittelpunkte ergibt die Position des Zielpunkts im zweidimensionalen Bild.
Die Bildkoordinaten dieses zentralen Punkts eignen sich bereits als Regelgrößen. Ähnlich der Implementierung in dem Beispielprogramm des Roboters zur Objektverfolgung, vergleiche Kapitel 4.2, lässt sich über die kontinuierliche x-Koordinate oder auch die diskrete Bildkoordinate u des Flächenmittelpunkts im Kamerabild ein möglicher lateraler Versatz zwischen der Mittellinie des Zielbereichs und des Roboters ableiten. Unter der Annahme, dass sich der Fahrroboter in beliebigem Abstand vor den Markierungen befindet und beide Mittellinien parallel zueinander stehen, lässt die horizontale Bildkoordinate mit dem Koordinatenursprung im Bildmittelpunkt qualitativ Rückschlüsse auf den tatsächlichen Abstand zwischen den Mittellinien zu. Für die Regelung des Lenkwinkels kann diese normierte Koordinate entweder direkt als Steuergröße genutzt oder zuvor über den Verstärkungsfaktor eines P-Reglers angepasst werden. Das Ziel einer solchen Regelung ist, diesen lateralen Versatz zu eliminieren ohne Kenntnis über den tatsächlich vorhandenen Abstand im dreidimensionalen Raum.
Ohne a-priori Wissen über die tatsächliche Größe der eingesetzten Markierungen mit einzubeziehen ist eine Bestimmung des euklidischen Abstandes im Raum zwischen der Kamera des Roboters und den Markierungen anhand eines Bildes nicht möglich, vergleiche hierzu Kapitel 2.2. Auch hier lassen sich über die y-Koordinate (oder auch v-Koordinate)

des Flächenmittelpunktes qualitative Aussagen treffen. Je weiter sich der Punkt der oberen Bildkante annähert, desto größer ist die Distanz im Welt-KS. Dieser Umstand lässt eine Regelung der Vorwärtsgeschwindigkeit des Roboters äquivalent zu dem Vorgehen bei der Lenkungsregelung zu. Bei großer Entfernung soll mit höherer Geschwindigkeit auf den Zielbereich zugefahren werden. Befindet sich der Mittelpunkt auf der Bildhorizontalen (y = 0 bzw. v = 0), hält der Roboter an. Über Vorgabe eines Offsets kann dieser Anhaltepunkt variiert werden.

Die bisherigen Aufführungen lassen sich allerdings nicht auf den allgemeinen Fall übertragen. Dieser berücksichtigt neben den translatorischen Versätzen in der Raumebene zusätzlich eine mögliche Verdrehung zwischen den Mittellinien des Fahrzeugs und der Zielfläche. Dementsprechend kann bei Lage des Flächenmittelpunkts exakt im Bildmittelpunkt nicht pauschal die Aussage getroffen werden, dass sich der Roboter in gewissem Abstand zentral vor der Einfahrtslinie befindet. Die beiden Koordinaten eines zentralen Punktes reichen nicht aus, um die drei entscheidenden Freiheitsgrade (2 x translatorisch, 1 x rotatorisch) der Kamerabewegung in der Ebene qualitativ einzuschränken. Die drei weiteren Freiheitsgrade des dreidimensionalen Raums sind für das hier vorgestellte Konzept zur Lokalisierung trivial. Sie sind nur mit einzubeziehen, wenn die Vertikaldynamik (Hub-, Nick- und Wankbewegung) des Fahrzeugs nicht vernachlässigbar ist.
Als Ansatz für die Abschätzung des Blickwinkels auf die parallelen Markierungen wird die Berechnung eines Referenzwinkels zwischen den beiden vorderen Landmarks (LD01 und LD02) verfolgt. Das Verfahren ist unabhängig davon, ob die Markierungen an einer Wand angebracht sind oder auf dem Boden liegen. Da bei schiefer Betrachtung dieser beiden Punkte in unterschiedlichem Abstand zum Kameraursprung besitzen, befinden sie sich im Abbild auch nicht auf gleicher Höhe (y-Koordinate). Nach Berechnung der Differenzen Δu und Δv erhält man schließlich über den Arkustanges den auf die konstante Kreiszahl Pi normierten Winkel α im Bogenmaß. Diese Berechnungsschritte werden durch die Formeln 4.1, 4.2 und 4.3 wiedergegeben. Bevorzugt zu verwenden sind hierbei die diskreten Bildkoordinaten u und v mit ihrem Ursprung im oberen linken Bildeck. Dadurch wird sichergestellt, dass die Differenz Δu stets positiv ist und nur der Abstand in v-Richtung über das resultierende Vorzeichen des Winkels α entscheidet.

$$\Delta u = u(LD02) - u(LD01) \tag{4.1}$$

$$\Delta v = v(LD02) - v(LD01) \tag{4.2}$$

$$\alpha = \frac{atan\left(\frac{\Delta v}{\Delta u}\right)}{\pi}; \quad [\alpha] = rad \tag{4.3}$$

Zur Verdeutlichung der Korrelation zwischen dem Vorzeichen dieses Referenzwinkels und einer ungefähren qualitativen Positionsbestimmung des Roboters dient Abbildung 4.13.

Befindet sich aus der Vogelperspekte betrachtet mit der Einfahrtslinie des Zielbereichs nach unten weisend die Kamera links von der referenzierten Mittellinie, nimmt der Winkel α ein negatives Vorzeichen an. Ein Vorzeichenwechsel findet bei Überschreiten der Mittellinie des markierten Bereichs statt. Der normierte Winkel bewegt sich in dem Wertebereich (- 0,5 ... 0.5). Die Extremwerte des realen Winkels entsprechen somit - 90 ° und 90 °. Eine zahlenmäßige Positionsschätzung ist bei diesem Vorgehen zur Lokalisierung nicht möglich, solange angenommen wird, dass keine Angabe zu der tatsächlichen

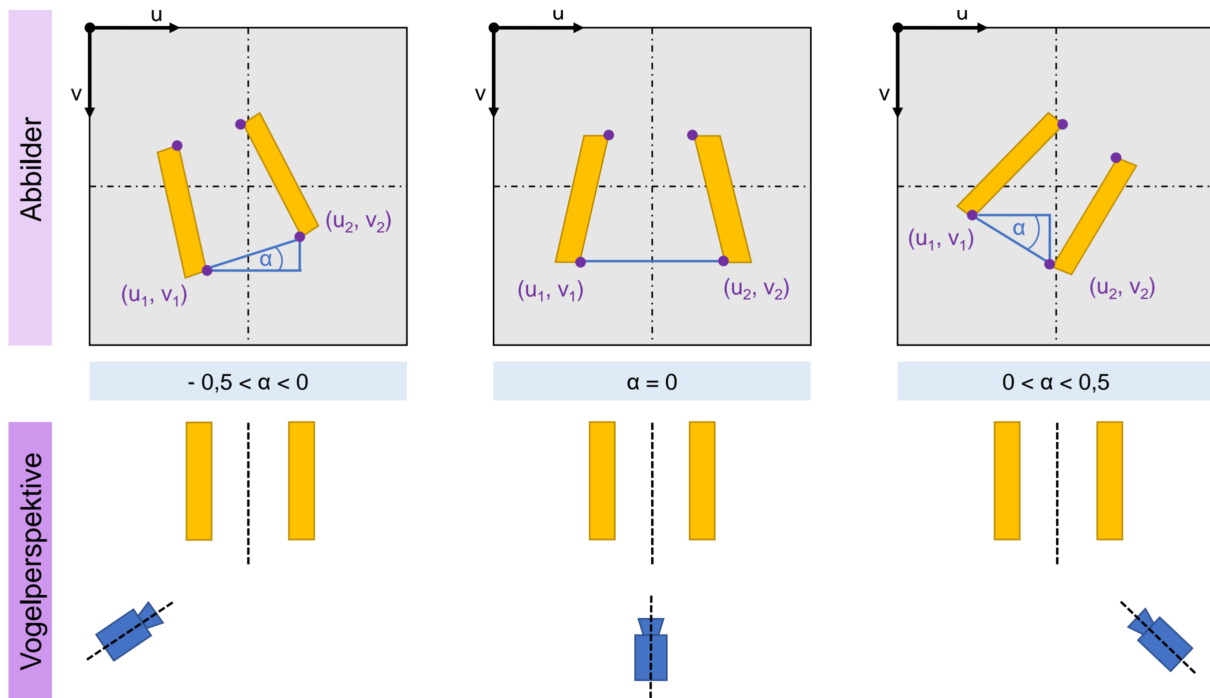


Abb. 4.13: Konzept zur relativen Positionsbestimmung basierend auf geometrischen Bildpunktezusammenhängen (qualitativ) [eigene Darstellung]

Größe der Markierungen vorhanden ist. Aber bereits diese einfache qualitative Auffassung reicht aus, um darauf aufbauend eine Funktion zu entwickeln, welche eine gezielte Ansteuerung eines definierten Zielpunkts erlaubt. Integriert wird dieses Lokalisierungskonzept zusammen mit einem Proportional-Regler und dem fundamentalen Landmark Detektions-Modell zu der Version 1 der Rangierfunktion, siehe Kapitel 5.1.

Um nicht nur einen zentralen Punkt anzusteuern, sondern auch eine Trajektorienverfolgung zu ermöglichen, liefert die visuelle Odometrie die hierzu notwendigen Regelgrößen quantitativ. Sie wird im weiteren Verlauf der Arbeit als vordergründige Strategie zur kamerabasierten Lokalisierung des Ego-Fahrzeugs forciert.

4.3.2 Visuelle Odometrie

Werden im Fahrzeug Odometriedaten durch eine IMU bereitgestellt, kann die zurückgelegte Bahn anhand der zeitlichen Integration relevanter sensorisch beobachteter Größen des Fahrzustands (z.B. Raddrehzahlen, Gierrate, Längs- und Querbeschleunigung) rekonstruiert werden [76]. Anstelle der messtechnischen Erfassung ermöglicht auch bereits das stark vereinfachte lineare Einspurmodell von Riekert und Schunck (1940) als kinematisches Fahrzeugmodell innerhalb des Gültigkeitsbereichs mit seinen lediglich sieben Modellparametern eine Abschätzung der ebenen Fahrzeugbewegung basierend auf dem Lenkwinkel. Der erhaltene Schwimmwinkel liefert eine Aussage zur momentanen Kursrichtung des Fahrzeugs, wohingegen die Gierrate die Winkelbeschleunigung der Drehung um die Fahrzeughochachse darstellt. Durch die Berechnung der lateralen und longitudinalen Komponenten des Geschwindigkeitsvektors im Massenschwerpunkt und Bestimmung des Gierwinkels (entspricht der Fahrzeugorientierung) ist die Bewegung des Fahrzeugs

in der Ebene vollständig definiert. In beiden Fällen ist jedoch eine vorherige Bestimmung der Anfangsposition als Initialisierungsschritt notwendig, um den zurückgelegten Kurvenverlauf mathematisch zu beschreiben.
Einen prinzipverwandten Ansatz zur Positionsbestimmung stellt die in einschlägiger Literatur diskutierte visuelle Odometrie (VO) dar [77, 41, 29].
Allgemein beruht dieses Vorgehen statt auf bereitgestellten physikalischen Größen einer Messeinheit oder eines Fahrdynamikmodells auf dem Zentralprojektionsmodell einer am Fahrzeug verbauten exteroperceptiven Kamera, vergleiche hierzu Kapitel 2.2. Über bekannte Punktkorrespondenzen zwischen der realen Welt und der Bildebene lassen sich die Modellparameter bestimmen. Wie aus der Herleitung der Projektionsmatrix in Kapitel 2.2 hervorgeht, ist diese schließlich in eine intrinsische und extrinsische Matrix zerlegbar. Von Interesse ist besonders die extrinsische Matrix, da sie die Koordinatentransformation zwischen dem Welt-KS und dem Kamera-KS beinhaltet und alle sechs Freiheitsgrade im Raum einschränkt. Ist zusätzlich die Lage des fahrzeugfesten Koordinatensystems relativ zum Ursprung des Kamera-KS bekannt, lässt sich die Position als auch die Orientierung des Fahrzeugs im Welt-KS vollständig abschätzen.
Bei den meisten bekannten VO-Techniken wird nach einer ersten Positionsbestimmung als Initialisierung in allen weiteren Berechnungsschritten nur noch die relative Änderung der Position über die Relation zwischen festgelegten Bildpunkten in aufeinanderfolgenden Frames ermittelt. Diese 2D-2D-Punktbeziehungen stellen einen Spezialfall der Transformationsmatrix (extrinsische Matrix), auch als Homographie bezeichnet, dar, vergleiche Kapitel 2.2. Eine große Herausforderung besteht jedoch in der zu berücksichtigenden Fehlerfortpflanzung (Drift). Durch schlichte Addition der jeweiligen mit Ungenauigkeiten behafteten Bewegungsschätzungen können bereits nach wenigen Zeitschritten inakzeptable Abweichungen von der tatsächlichen Bahnkurve beobachtet werden. Bisherige Herangehensweisen lösen diesen Konflikt überwiegend durch eine iterative Rückrechnung und anschließendes Justieren der bereits bestimmten Fahrstrecke mithilfe von Reprojektionen der Bildpunkte im dreidimensionalen Raum und einem Abgleich mit den tatsächlichen 3D-Punktkoordinaten. Eine ähnliche Arbeitsweise besitzen SLAM-Algorithmen, welche sich ebenfalls zur Unterdrückung eines solchen Drifts eignen [41].

Basierend auf den vorangegangenen Erläuterung wird im Folgenden ein Ansatz zur Bereitstellung visueller Odometriedaten für die Erfüllung des Rangiermanövers durch den eingesetzten Fahrroboter vorgestellt. Im Gegensatz zu bisherigen Konzepten, welche oftmals beliebige, durch einen Detektionsalgorithmus erkannte, dreidimensionale Punkte als sogenannte "Keypoints" [77] nutzen, liegen bei dem hier betrachteten Anwendungsfall bereits vier Landmarks des statischen Zielbereichs vor. Diese Bildpunkte als Ausgangsgrößen des in Kapitel 4.1.2 vorgestellten DL-Modells zur Objektdetektion werden im weiteren Verlauf als Referenz für den Algorithmus zur visuellen Odometrie verwendet. Weiterhin wird die Begebenheit genutzt, dass alle vier 3D-Punkte koplanar sind. Somit können die Punktkorrespondenzen über das 2D-2D-Mapping der Homographie erschlossen werden. Um schließlich den Translationsvektor als auch die Rotationsmatrix daraus ableiten zu können, muss zuvor eine einmalige Kalibrierung der Kamera durchgeführt werden. Diese legt die invarianten internen Kameraparameter als auch die Verzeichnungskoeffizienten offen.
Da die Positionsbestimmung allein auf diese vier ebenen Punkte ausgerichtet ist und somit nur ein Ergebnis geliefert werden kann, wenn diese sich im Sichtfeld der Kamera befinden und detektiert werden, ist von einer integrativen Berechnung der zurückgelegten Bahn des Roboters abzusehen. Durch eine Neuberechnung der momentanen Lage

der Kamera im Raum in jedem Zeitschritt stellt sich außerdem die Problematik des zu erwartenden Drifts nicht. Der Berechnungsaufwand ist äquivalent zu den oben erörterten Konzepten, da es stets eine Homographie zu lösen gilt - wie auch bei der Bestimmung der Transformationsmatrix zwischen zwei aufeinanderfolgenden Kamerabildern. Angewendet werden kann dieses Vorgehen sowohl bei auf dem Boden platzierten als auch an einer festen Wand befestigten Markierungen. Einzige einzuhaltende Voraussetzung ist die Kenntnis von mindestens vier Punktkorrespondenzen (Koordinaten im Bild-KS als auch im Welt-KS), wobei die Koplanarität dieser Punkte sicherzustellen ist.

Zusammenfassend sind folgende Größen vorzubereiten oder anzugeben, um den nachfolgend vorgestellten Algorithmus zur visuellen Odometrie implementieren zu können:

- Mindestens vier Punkte mit ihren Koordinaten im Welt-KS, welche alle in einer gemeinsamen Ebene liegen
- Bildkoordinaten dieser vier Punkte in jedem Frame
- Interne Kameraparameter (intrinsische Matrix) durch vorherige Kalibrierung (z.B. automatische Kalibrierung mit der Checkerboard-Methode, vergleiche Kapitel 2.2)
- Optional die Koeffizienten der Nichtlinearitäten (Verzeichnungen) ebenfalls aus der Kamerakalibrierung

Als Ausgangspunkt des Berechnungsalgorithmus steht die zahlenmäßige Bestimmung der Homographie-Matrix H, sodass die aus Formel 2.7 abgeleitete Gleichung 4.4 mit Z_W = 0 erfüllt ist. Diese Gleichung ist für beide Fälle der Positionierung der Markierungen (Boden oder Wand) gültig, indem das Welt-KS entsprechend festgesetzt wird. Die ursprüngliche 3 x 4 - Matrix Q vereinfacht sich zu der 3 x 3 - Matrix H.

$$\tilde{p}_{u,v} = \begin{bmatrix} U \\ V \\ S \end{bmatrix} = \mathbf{I}\,\mathbf{E}\,\tilde{P}_W = \mathbf{H}\,\tilde{P}_W = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \cdot \begin{bmatrix} X_W \\ Y_W \\ 1 \end{bmatrix} \tag{4.4}$$

Aus einer Linksmultiplikation der inversen intrinsischen Matrix I mit der Homographie-Matrix H geht schließlich die extrinsische Matrix E hervor, siehe Formel 4.5.

$$\mathbf{E} = \mathbf{I}^{-1}\,\mathbf{H} \tag{4.5}$$

Zu beachten ist, dass die aus der Kalibrierung erhaltene Kameramatrix noch manuell anzupassen ist. Die Modelloutputs der Landmark Detektion stellen normierte kontinuierliche Bildkoordinaten bezogen auf den Bildmittelpunkt dar. Um diese direkt für die VO verwenden zu können, darf der Ursprung des Bildkoordinatensystems nicht durch die intrinsische Matrix in die obere linke Ecke des Bildes verschoben werden. Dementsprechend sind die Elemente der Matrix I, welche die Verschiebungsfaktoren u_0 und v_0 enthalten manuell gleich Null zu setzen, vergleiche Formel 2.5.

Für die Implementierung der bisher aufgeführten Bestandteile des Algorithmus in Python-Code bietet sich der in *OpenCV* verfügbare Gleichungslöser für perspektivische Tranformationen an: *cv2.solvePnP()*. Als Eingabegrößen werden der Funktion die Koordinaten der realen Punkte und der korrelierenden Bildpunkte in Form von Tuples in einem eindimensionalen Array zusammen mit der Kameramatrix übergeben. Sollen die Nichtlinearitäten miteinbezogen werden, dann sind auch die Koeffizienten der Verzeichnungen anzugeben. Anhand des Funktionsparameters "flags" wird die zu verwendende Lösungsmethode definiert. Bei diesem speziellen Anwendungsfall ist die Methode *SOLVEPNP_IPPE* (flags = 6) zu wählen, welche mindestens vier koplanare Raumpunkte zur eindeutigen Lösung des Gleichungssystems voraussetzt. Für eine Übersicht aller verfügbaren Lösungsmethoden ist auf die Dokumentation von *OpenCV* zu verweisen [78].
Ausgegeben werden neben einer Statusmeldung direkt der Rodrigues Rotationsvektor und der Translationsvektor im Kamera-KS. Bei dem Rodrigues Rotationsvektor handelt es sich um eine in der Computergrafik weit verbreitete Darstellung der Rotation eines Koordinatensystems. Statt einer Rotationsmatrix, welche basierend auf den Euler-Winkeln gebildet wird, definiert dieser Vektor mithilfe von lediglich drei Parametern die Rotationsachse im Raum. Da jedoch eine Angabe der Rotation über die Euler-Winkel intuitiver und besser nachvollziehbar ist, wird eine Ableitung dieser Winkel aus dem Rodrigues Rotationsvektor in einem weiterführenden Berechnungsschritt angestrebt. Die Rotationsmatrix R lässt sich über die Funktion *cv2.Rodrigues()* identifizieren.
Um den Translationsvektor bezogen auf das kartesische Welt-KS zu erhalten, folgt eine Matrixmuliplikation der negativen Inversen der Rotationsmatrix und des ursprünglichen Translationsvektors bezogen auf das Kamera-KS. Eine Transformation der Rotationsmatrix in das Kamera-KS ist nicht zwingend erforderlich, da hiermit nur die Vorzeichen der Rotationswinkel wechseln würden. Als Zwischenergebnis steht nun die Lage und Orientierung des Kamera-KS innerhalb des zuvor beliebig festgelegten Welt-KS über die Rotationsmatrix und den Translationsvektor zur Verfügung.
Wie in Formel 2.2 ersichtlich, sind die Euler-Winkel nicht direkt analytisch anhand der bekannten Rotationsmatrix identifizierbar. Hierfür wird auf den von Slabaugh (2020) vorgestellten Pseudo-Code zurückgegriffen [10]. Dieser beschreibt eine Methode zur Lösungsfindung unter Berücksichtigung des sogenannten "Gimbal Locks". Durch die Periodizität der Winkelfunktionen existiert theoretisch eine Vielzahl an möglichen Lösungen. Der in Abbildung 4.14 aufgezeigte Pythoncode beschränkt sich auf die ersten beiden Lösungen für jeden Winkel. Die Variablen R_{ij} stellen die Elemente der Rotationsmatrix dar und Ψ den Rotationswinkel um die x-Achse, Θ um die y-Achse und Φ um die z-Achse [ebd].

Wird der gesamte Algorithmus in jedem neuen Bild der Videokamera im Anschluss an die Detektion der vier Landmarks angestoßen, liegt in gewissen Zeitschritten eine aktuelle Positionsschätzung der Kamera im Welt-KS vor. Da die Lage und Orientierung der Kamera im KS zur Einfassung des Roboters in den räumlichen Kontext bereits ausreichend ist, wird auf eine darauf aufbauende Lageschätzung des Fahrzeugmassenschwerpunkts bewusst verzichtet.
Um eine Trajektorienplanung zu ermöglichen, wird die kalkulierte Kameraposition in einem Belegungsnetz (engl.: Occupancy Grid) festgehalten. Statt hier die Lage bekannter Objekte relativ zum Fahrzeug zu dokumentieren, bildet eine Matrix die X-Y-Ebene des Raums ab mit dem Ursprung des Welt-KS als Fixpunkt und der Anordnung der Markierungsstreifen als statische Objekte. Es wird eine 500 x 500 - Bildmatrix deklariert, welche die zu betrachtende Ebene diskretisiert. Die quadratischen Pixel repräsentieren ein Flächenelement mit den Dimensionen 1 cm x 1 cm. Festgelegt wird außerdem, dass sich

```
if (R31 ≠ ±1)
   θ1 = −asin(R31)
   θ2 = π − θ1
   ψ1 = atan2(R32/cos θ1, R33/cos θ1)
   ψ2 = atan2(R32/cos θ2, R33/cos θ2)
   φ1 = atan2(R21/cos θ1, R11/cos θ1)
   φ2 = atan2(R21/cos θ2, R11/cos θ2)
else
   φ = anything; can set to 0
   if (R31 = −1)
      θ = π/2
      ψ = φ + atan2(R12, R13)
   else
      θ = −π/2
      ψ = −φ + atan2(−R12, −R13)
   end if
end if
```

Abb. 4.14: Pseudo-Code zur Ableitung der Rotationswinkel nach Euler aus einer gegebenen Rotationsmatrix [10]

der Ursprung des Welt-KS zentral am oberen Rand des Belegungsnetzes befindet und gleichzeitig den Mittelpunkt der Einfahrtslinie, definiert durch die Markierungen, darstellt. Innerhalb des Koordinatensystems dieser Bildmatrix zur Darstellung einer Vogelperspektive besitzt dieser Ursprung die diskreten ganzzahligen Koordinaten (x_origin, y_origin). Die eingesetzten Markierungen besitzen eine Länge von 37 cm, wodurch sich die Koordinaten des Ursprungs zu (250, 37) ergeben. Bei an einer senkrechten Wand befestigten Markierungen kann der Ursprung des Welt-KS zu (250,0) gesetzt werden. Durch Addition der x- und y-Komponenten des Translationsvektor mit den Ursprungskoordinaten ergibt sich der in die Ebene projizierte Ursprung des Kamera-KS. Die Ausrichtung der optischen Achse (z-Achse des Kamera-KS) wird über den Rotationswinkel Φ bestimmt. Abbildung 4.15 ergänzt die vorangegangen Erläuterungen durch eine Visualisierung des Sachverhalts. Es werden beide Fälle der möglichen Anordnung der Markierungen betrachtet.

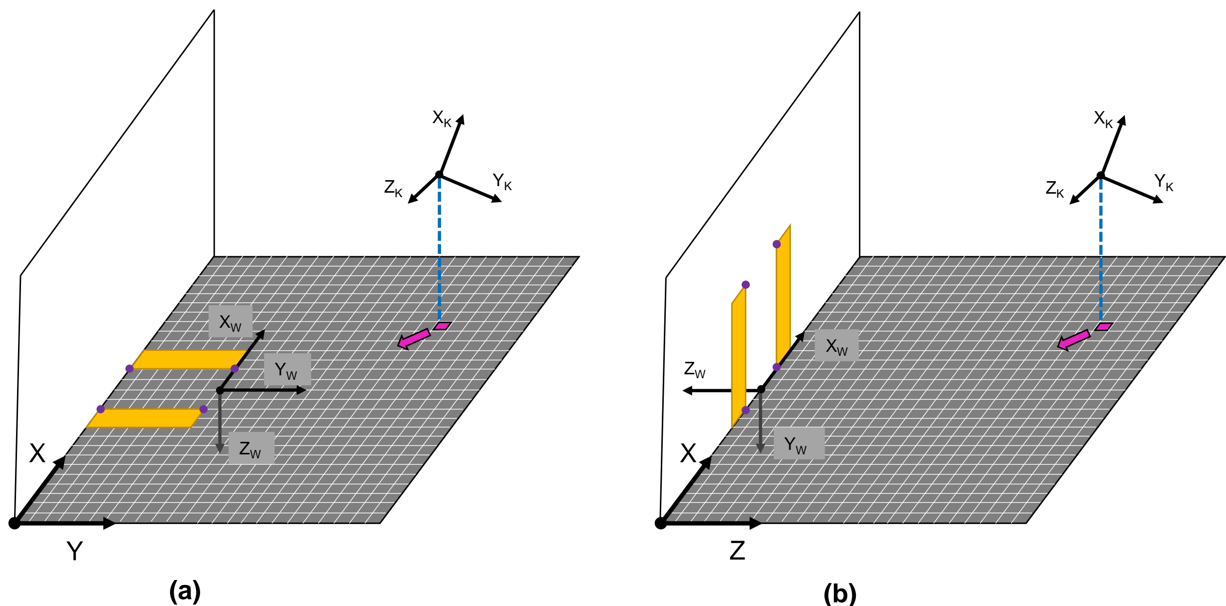


Abb. 4.15: Homographiebasierte Positionsschätzung der monokularen Kamera in der Ebene: (a) Markierungen auf Boden, (b) Markierungen an Wand [eigene Darstellung]

Erste statische Tests zeigen, dass der vorgestellte Algorithmus zur homographiebasierten VO die Position als auch die Ausrichtung der Kamera richtig abschätzt. Zahlenmäßige Untersuchungen der Genauigkeit dieses Ansatzes sind nur bedingt realisierbar, da die Ground Truth der Kameralage nicht eindeutig zu bestimmen ist. Durch das Wissen, dass sich der Kameraursprung innerhalb der Optik befindet, lässt sich die Lage und die Orientierung der Kamera im Raum in Realität auch nur durch eine ungefähre Messung abwägen. Jedoch befinden sich die erhaltenen Werte in einem plausiblen Zahlenbereich, wodurch eine verifiziertes Ergebnis dieses Entwicklungsschritts vorliegt.
An dieser Stelle lassen sich bereits Tendenzen hinsichtlich der Genauigkeit der VO ableiten. Es ist zu beobachten, dass die korrekte Bestimmung der intrinsischen Kameramatrix einen großen Einfluss auf die Qualität der Lokalisierung besitzt. Bei einer späteren Implementierung wird empfohlen, anstelle der schnellen Kalibrierung mithilfe eines ausgedruckten Schachbrettmusters auf exaktere Kalibrierungsmethoden zurückzugreifen.
Die Positionsschätzung wird umso ungenauer, je weiter entfernt sich der Roboter von den Markierungen befindet. Es ist festzustellen, dass die besten Ergebnisse bei kurzen Distanzen und einer zentralen Position vor dem Einfahrtsbereich erhalten werden. Dies liegt vor allem an der Leistungsfähigkeit des DL-Modells zur Detektion der Landmarks. Je größer hier die Abweichungen bei den vorherzusagenden Bildpunkten sind, desto schlechter funktioniert auch die darauf basierende Lokalisierung des Fahrroboters relativ zu dem Zielbereich. Eine Evaluierung der Auswirkungen dieser Modellungenauigkeiten erfolgt in Kapitel 5 bezogen auf die Gesamtfunktionalität. Ausreißer bei der Lagebestimmung lassen sich möglicherweise durch geeignete Regelstrategien oder entsprechende Maßnahmen bei der Integration aller Funktionsbausteine zu einer Gesamtlösung unterdrücken.

4.4 Trajektorienplanung

Den letzten fehlenden Baustein zur Darstellung eines automatisierten Rangiervorgangs stellt die Berechnung einer Referenztrajektorie für die durchzuführende Fahrzeugbewegung dar. Wie auch die Lokalisierung des Fahrroboters wird die Trajektorienplanung in der X-Y-Ebene des Welt-KS vorgenommen. Herangezogen wird hierfür ebenfalls das in Kapitel 4.3.2 eingeführte Belegungsnetz in Form einer 500 x 500 - Bildmatrix. Statt der üblichen kubischen Spline-Interpolation zwischen vorgegebenen Stützstellen und einer Aktualisierung dieser in festgelegten Zeitschritten, ist im Hinblick auf das stabile Zusammenspiel zwischen visueller Odometrie und der Trajektorienplanung eine statische Berechnung zu bevorzugen.
Verfolgt wird ein parabolischer Ansatz mit der Funktion $f(y) = ay^2 + by + c$ bei der Kalkulation der Referenztrajektorie, siehe die Visualisierung in Abbildung 4.16. Sie beschreibt die x-Koordinate innerhalb des Occupancy Grids abhängig von y. Die unbekannten Funktionsparameter a,b und c sind durch Bereitstellung von mindestens drei unabhängigen Gleichungen und das Lösen dieses linearen Gleichungssystems bestimmbar. Als Endpunkt der Trajektorie werden weder der Flächenmittelpunkt des Zielbereichs noch der Ursprung des Welt-KS platziert auf dem Mittelpunkt der Einfahrtslinie zwischen Landmark 1 und 2 genutzt. Durch die Definition eines Zielpunkts P1, verschoben um einen gewissen Betrag in y-Richtung relativ zu dem Ursprung des Welt-KS, kann sichergestellt werden, dass sich die Markierungen möglichst durchgehend während der Trajektorienverfolgung vollständig im Sichtfeld der Kamera befinden. Liegt dieser Endpunkt zu nahe an dem Zielareal, ragen diese über den aufgenommenen Bildrand hinaus. Um eine Po-

sitionsbestimmung durch visuelle Odometrie zu erhalten, ist jedoch die Kenntnis der vier innenliegenden Eckpunkte der beiden Markierungsstreifen von Bedeutung. Da die Modelle zur Objektdetektion so trainiert wurden, dass sie die Lage der vier Landmarks nur dann ausgeben, wenn beide Markierungen gänzlich im Bild zu sehen sind, kann dies zum Ausfall der VO führen. Der notwendige vorzusehene Abstand ist experimentell offenzulegen und entsprechend der Strecke zwischen den beiden parallelen Markierungen zu skalieren.
Als Startpunkt der Trajektorie steht die aus der Positionsschätzung bekannte Lage des Kameraursprungs P_2 mit seinen Koordinaten (x_2,y_2) innerhalb des Belegungsnetzes fest. Die dritte zu erfüllende Bedingung betrifft die Lage des Scheitelpunkts der Parabel. Dieser soll stets mit dem Endpunkt der Trajektorie zusammenfallen. Dadurch ist die Ableitung $f'(y_1) = 0$ in P_1.

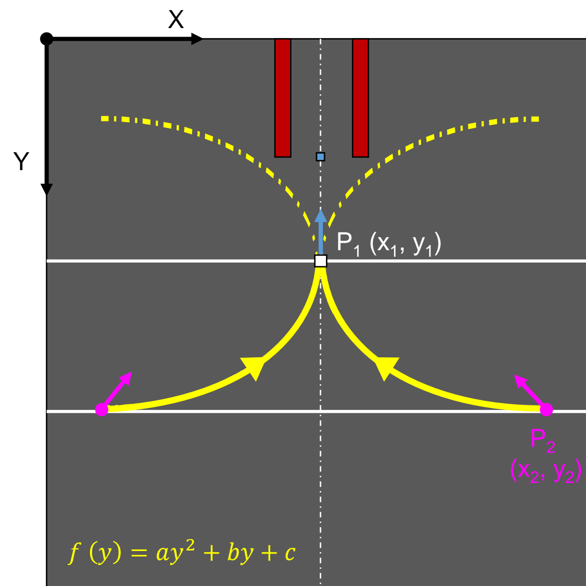


Abb. 4.16: Trajektorienplanung für die ebene Fahrzeugbewegung innerhalb festgelegter Bereichsgrenzen durch Einfügen einer parabolischen Funktion [eigene Darstellung]

Diese drei zu erfüllenden Gleichungen, siehe Formeln 4.6, 4.7 und 4.8, bilden schließlich zusammen das Gleichungssystem, in Formel 4.9 dargestellt in Matrixschreibweise mit der Koeffizientenmatrix K und dem Ergebnisvektor b. K muss als reguläre Matrix mit linear unabhängigen Elementen vorliegen, um die gesuchten Funktionsparameter durch Invertierung der Koeffizientenmatrix ableiten zu können.

$$f(y_1) = a(y_1)^2 + by_1 + c = x_1 \tag{4.6}$$

$$f(y_2) = a(y_2)^2 + by_2 + c = x_2 \tag{4.7}$$

$$f'(y_1) = 2ay_1 + b = 0 \tag{4.8}$$

$$b = \begin{bmatrix} x_1 \\ x_2 \\ 0 \end{bmatrix} = \mathbf{K} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} (y_1)^2 & y_1 & 1 \\ (y_2)^2 & y_2 & 1 \\ 2y_1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{4.9}$$

Weitere mögliche Randbedingungen zur Optimierung der berechneten Solltrajektorie werden hier nicht betrachtet aufgrund fehlender Relevanz für die prototypische Anwendung auf dem Fahrroboter. In zukünftigen Implementierungen der Rangierfunktion würden sich Optimierungen der zurückzulegenden Fahrstrecke hinsichtlich des Fahrgastkomforts oder der Energieeffizienz eignen.

Abschließend ist der Berechnungsvorgang der Solltrajektorie zu erörtern. Wie bereits eingangs erwähnt, wird eine statische Bestimmung der zu folgenden Fahrstrecke angestrebt. Dies bedeutet, dass der Fahrroboter für kurze Zeit anhält, um die Planung durchzuführen. Benötigt wird diese Maßnahme aufgrund der Abhängigkeit des Anfangspunktes P_2 von der Genauigkeit der Lokalisierung durch visuelle Odometrie.
Es ist zu beobachten, dass Beschleunigunsphasen und Vibrationen, hervorgerufen durch Bodenunebenheiten, die Leistungsfähigkeit der VO negativ beeinflussen. Die approximierten Kamerapositionen unterliegen teilweise großen Schwankungen. Diese würden bei einer dynamischen Trajektorienplanung während des Fahrbetriebs ("on the fly") auch die resultierende Trajektorie verfälschen.
Um eine möglichst exakte Referenz für die ebene Fahrzeugbewegung bereitzustellen, ist somit der Roboter für kurze Zeit anzuhalten. Die hierfür eingeplante Zeitspanne umfasst mehrere Kameraframes und bewegt sich voraussichtlich im ms-Bereich. Über diese Frames erfolgt eine Mittelung der Position, welche einen robusten Anfangswert für die Referenztrajektorie zur Verfügung stellt. Die Funktionsparameter werden einmalig im Stand identifiziert und für den weiteren Betrieb eingefroren. So kann zu jeder Kameraposition über die bekannte Funktion ein Sollwert für die x-Koordinate, welche vor allem für die laterale Bewegung entscheidend ist, ausgegeben werden. Die momentane Steigung des Kurvenverlaufs an gegebenen Punkten dient als Referenz für die Kameraorientierung. Durch die Lage des Extrempunktes der jeweilig berechneten Parabeln und der daraus entstehenden Vorgabe einer Steigung von Null in Punkt P_1 soll zusätzlich die Zentrierung des Roboters vor der Einfahrt in den Zielbereich forciert werden.
Erreicht der Fahrroboter einen zuvor festgelegten Abstand relativ zu dem zu befahrenden Bereich, ist dieses Manöver auszuführen. Somit sind auch die Bereichsgrenzen innerhalb des Belegungsnetzes definiert, innerhalb welcher eine Trajektorie vorliegt. Der Anhaltepunkt ist so zu wählen, dass eine ausreichend große Strecke zwischen Anfangs- und Endpunkt der parabelförmigen Bahn vorliegt. Hierdurch wird sichergestellt, dass keine zu großen Steigungen entlang des Kurvenverlaufs entstehen, welche zu ungewollt großen Lenkwinkeln und Querbeschleunigungen führen würden.

Die virtuelle Odometrie und die Trajektorienplanung werden in Version 2, siehe Kapitel 5.2, vereint und durch einen PID-Regelungsalgorithmus ergänzt.

5 Integration der Softwarekomponenten und Funktionsvalidierung

Die im vorangegangenen Kapitel in sich abgeschlossen entwickelten Funktionsbausteine sind zu einer auf dem Fahrroboter ausführbaren Funktion zusammenzuführen. Das Zusammenspiel aller beteiligten und verifizierten Komponenten wird durch das Rapid Prototyping in der realen Welt erfahrbar. Neben einer frühzeitigen Funktionsvalidierung lassen sich außerdem bestehende Grenzen und offene Optimierungspotenziale für die Weiterentwicklung der im Rahmen dieser Arbeit entworfenen Grundfunktionalität identifizieren.
Betrachtet werden bei dieser Funktionsintegration nur auf dem Boden liegende Markierungen, vergleiche Fall 1 in Kapitel 1.2. Markierungen an einer senkrechten Ebene angebracht bringen die Herausforderung mit sich, dass sie sich nur bei größeren Distanzen vollständig im Sichtfeld der Kamera befinden.
Die unterstützenden Softwarekomponenten für die Ausführung der Fahrfunktion auf dem Roboter, wie die Einbettung der Python-Funktionsbibliotheken, das Preprocessing der Kamerabilder oder etwaige Initialisierungsschritte sind ebenfalls in dem abgebildeten Programmcode zu Version 2 in Anhang 9.1 enthalten.

Anzumerken ist, dass bei den während des finalen Trainings auf dem Datensatz "07.12.2022" gespeicherten Kantengewichte des verwendeten Modells zur Landmark Detektion vor der Implementierung eine Laufzeitoptimierung durchgeführt wird. Hierfür wird das SDK *TensorRT*^{TM} (TRT) von NVIDIA angewendet [79]. Nach Erstellung einer Datei mit den optimierten TRT-Modellparametern ist das Modell über die Klasse *TRTModule()* der Python-Bibliothek *torch2trt* zu initialisieren.

5.1 Version 1

Nach Entwicklung des Modells zur Landmark Detektion, siehe Kapitel 4.1.2, wird durch die erste Versionierung eine Minimallösung der Fahraufgabe dargeboten. Sie bedient sich nur der Objekterkennung zusammen mit einem PD-Regler zur Positionsregelung. Basierend auf einer rein qualitativen Lokalisierung, wie in Kapitel 4.3.1 vorgestellt, soll der Roboter den Mittelpunkt der Zielfläche ansteuern und somit diesen Bereich befahren, um die Fahraufgabe zu erfüllen. Berücksichtigt werden hierbei rein die Bildkoordinaten des Flächenmittelpunkts ohne weitere Rechenschritte oder eine Ableitung anderer Größen.

Angestoßen bei jeder Aktualisierung des Kamerabildes wird der Output des Modells zur Objektdetektion erfasst. Anwendung findet das in Kapitel 4.1.2 festgelegte Modell V211 mit ResNet50-Backbone trainiert auf dem großen Datensatz "07.12.2022". Die Berechnung des Modells findet auf der GPU des NVIDIA Entwicklerboards des Fahrroboters statt, wobei der Ergebnisvektor an die CPU übergeben wird. Zur Visualisierung der erkannten Landmarks werden die in dem Modelloutput enthaltenen Koordinaten in diskrete Pixelkoordinaten bezogen auf die obere linke Ecke des Bildes transformiert. Für die eigentliche Funktionalität ist dieser Schritt nicht zwingend erforderlich.
Ist die von dem DL-Modell bestimmte Objektwahrscheinlichkeit größer als ein vorgegebener Schwellwert, wird die programmierte Fahrstrategie ausgeführt. Andernfalls verharrt der Fahrroboter im Stillstand. Die x- und y-Koordinate des Flächenmittelpunkts muss bezogen auf den Bildmittelpunkt und normiert auf die halbe Bildbreite oder Bildhöhe vorliegen, um so direkt den Abstand zum Bildzentrum wiederzugeben. Der Wert der x-

Koordinate wird als Regelabweichung der lateralen Bewegung betrachtet. Dagegen beschreibt die y-Koordinate die Regelabweichung innerhalb der Regelung der longitudinalen Roboterführung. Beide Regelgrößen (Lenkwinkel und Geschwindigkeit) werden durch getrennte PD-Regler kontrolliert. Eine reine proportionale Regelung führt zu unzulässig großen Überschwingern. Über interaktive Schieberegler lassen sich die Reglerparameter während des Betriebs variieren. Die Regelabweichungen (x- und y-Koordinate des MP) als auch die daraus resultierenden Stellgrößen für die Lenkung und die Geschwindigkeit werden ebenfalls über kontinuierliche Schieberegler ausgegeben. Zu beachten ist, dass bei der Geschwindigkeitsregelung die Stellgröße zu negieren ist, aufgrund der Definition der Ordinate in der Bildebene, vergleiche Abbildung 4.4. Ein Speed-Bias verschieden von Null erzielt eine Verschiebung des Zielpunktes entlang der Ordinate.
Zur Ansteuerung des linken Antriebsmotors werden die beiden Stellgrößen addiert. Bei dem rechten Motor muss die Stellgröße für die Lenkung mit einem negativen Vorzeichen versehen werden.

Die Reglerparameter sind so lange zu aktualisieren bis der Fahrroboter stabil und ohne große Überschwinger das Zentrum der markierten Zielfläche ansteuert. Als optimale Einstellung der Parameter wurden folgende Werte im Versuch ermittelt:

Geschwindigkeitsregelung: $K_P = 0.5; K_D = 0.2;$ Speed-Bias (Offset) $= 0.4$

Lenkungsregelung: $K_P = 0.5; K_D = 0.2;$ Steering-Bias (Offset) $= 0$

Diese erste Funktion erfüllt zwar die Vorgabe, die festgelegte Fläche ausgerichtet auf deren Mittelpunkt zu befahren, jedoch nicht wie gewünscht möglichst parallel zu den langen Kanten der Markierungsstreifen. Sie zeigt, dass die Einschränkung des dritten Freiheitsgrads der ebenen Bewegung, die Drehung um die Fahrzeughochachse (Orientierung), zwingend erforderlich ist, um ein valides Ergebnis zu erhalten. Hierfür eignet sich theoretisch auch bereits die Einführung eines Referenzwinkels berechnet aus geometrischen Bildzusammenhängen, siehe Kapitel 4.3.1. Für eine Lageregelung ist diese qualitative Aussage allerdings in der Praxis nicht brauchbar, da dieses Vorgehen nicht nur ungenaue Werte liefert, sondern auch stark von den Nichtlinearitäten im Abbildungsprozess der Weitwinkelkamera gestört wird. Dieser Ansatz der Funktionsintegration ist somit nicht zielführend und wird nicht weiter ausgearbeitet.
Für die Weiterentwicklung zu Version 2 des Funktionsprototyps wird basierend auf den gewonnenen Erkenntnissen gezielt die Anwendung der visuellen Odometrie (vergleiche Kapitel 4.3.2) zusammen mit der Vorgabe einer Referenztrajektorie (vergleiche Kapitel 4.4) in der Ebene verfolgt.

5.2 Version 2

Die in Version 2 festgehaltene Fahrfunktion besitzt eine deutlich gesteigerte Komplexität im Vergleich zu der Vorgängerversion. Sie vereint alle bisher vorgestellten Funktionsbausteine aus Kapitel 4 (Objektdetektion, Steuerung, Lokalisierung mit VO und Trajektorienplanung) in sich. Neben Strategien zur Ausübung des Rangiermanövers ist nun auch eine Suchstrategie vorhanden, um eine durchgängige Funktionalität auch bei fehlenden Markierungen innerhalb des Kamerasichtfelds darzustellen.

Die folgenden Inhalte liefern eine ausführliche Beschreibung des gesamten Programmcodes dieser Version, welcher im Anhang unter 9.1 zu finden ist.

Das Ausführungsprogramm unterteilt sich in mehrere Schichten. Über Kontrollstrukturen (if-else-Anweisungen) im Programmcode wird in jeder Schicht der momentan vorliegende Fahrzustand weiter spezifiziert, um schließlich die passende Fahrstrategie zu selektieren. Einen Überblick dieses kaskadierten Aufbaus der Software liefert Abbildung 5.1.

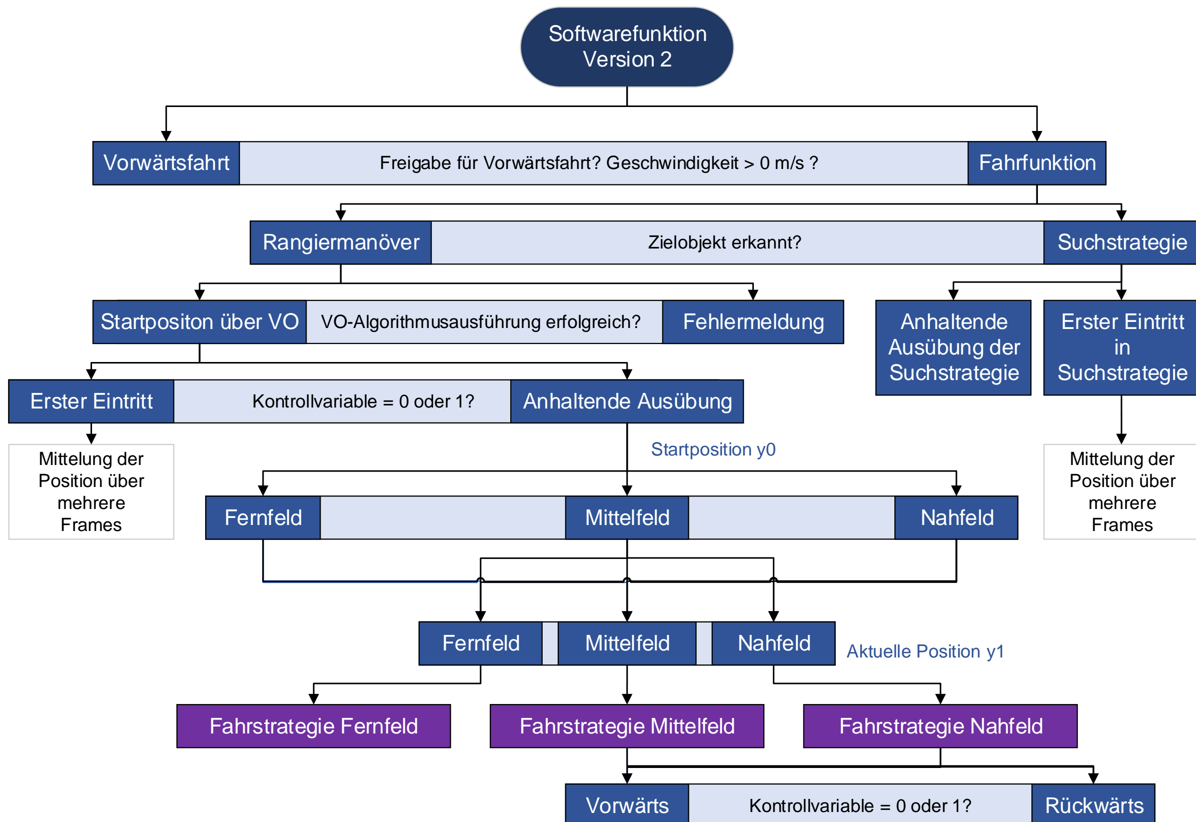


Abb. 5.1: Kontrollstruktur und schematische Darstellung der Schichten der Softwarefunktion (Version 2) [eigene Darstellung]

Da die Landmark Detektion voraussichtlich keine Ergebnisse liefert, wenn die Markierungen nicht vollumfänglich im Bild zu erkennen sind, ist für das tatsächliche Befahren des Zielareals eine separate Funktion vorzusehen, welche ohne die Landmark Detektion und die darauf basierende VO arbeitet. In der ersten Softwareebene ist zu entscheiden, ob die eigentliche Fahrfunktion auszuführen ist oder bereits eine schlichte Geraudeausfahrt die Fahraufgabe löst. Für beide Wahlmöglichkeiten muss eine Geschwindigkeit größer als Null als Voraussetzung gegeben sein. Ist zusätzlich die bool'sche Kontrollvariable *release_forward* positiv, wird die Vorwärtsfahrt angestoßen. Sie wird nach einer gewissen Anzahl an übergebener Frames beendet, da keine Abstandsinformationen vorliegen und das System geht in den Leerlaufmodus über durch automatisches Nullsetzen der Sollgeschwindigkeit. Bei erstmaliger Ausführung der Softwarefunktion wird zuerst immer der rechte Pfad durchlaufen, vergleiche Abbildung 5.1. Der initiale Wert der Kontrollvariable ist somit *False*. Um interpretieren zu können, in welchem räumlichen Zusammenhang der Roboter und das Zielobjekt zueinander stehen, muss sichergestellt werden, dass die

Startposition über die VO erkannt wird. Die Freigabe für die reine Vorwärtsfahrt wird nur dann erteilt, wenn zuvor der Fahrroboter im Nahbereich der Markierungen lokalisiert wurde und dieser gleichzeitig einen max. lateralen Versatz als auch eine max. Verdrehung relativ zu der Mittellinie des Zielobjekts innerhalb festgesetzter Grenzen aufweist. Um die Softwarefunktion anschließend neu starten zu können, erfolgt zunächst eine Initialisierung, welche alle Variablen auf ihren Anfangswert zurücksetzt.

Eine Ebene tiefer wird das Modell ResNet_V211 (07.12.2022) zur Landmark Detektion ausgeführt und anhand des ersten Output-Elements (= Objektwahrscheinlichkeit) entschieden, ob die Markierungen im Bild zu erkennen sind oder nicht. Befindet sich diese Wahrscheinlichkeit über einem bestimmten Schwellwert, wird die Ausführung der eigentlichen Rangierfunktion angestoßen. Andernfalls führt der Fahrroboter eine Suchstrategie aus, um diese in der Umgebung dennoch zu detektieren. Diese Strategie besteht aus einer Kreisfahrt im UZS mit einem sich langsam vergrößernden Radius. Wichtig zu beachten ist, dass die gesamte Softwarefunktion bei jedem neuen Kamerabild von vorne durchlaufen wird und somit auch die Bedingungen in jedem Zeitintervall neu geprüft werden. Um eine robuste Ansteuerung des Zielbereichs zu realisieren, ist vor dem Eintritt in die Suchstrategie zu gewährleisten, dass tatsächlich über mehrere Frames kein Objekt erkannt wurde und es sich nicht um einen einmaligen Aussetzer handelt. Gesteuert wird dieses Verhalten ebenfalls über eine dedizierte Kontrollvariable, welche angibt, ob gerade zum ersten Mal die Suchstrategie ausgeübt wird oder sich der Roboter schon länger im Suchmodus befindet.

Gibt der “Success”-Faktor bei Lösung des VO-Algorithmus einen wahren Wert zurück, ist die Freigabe für den nächsten Schritt gegeben. Sollte die Positionsschätzung trotz detektierter Landmarks scheitern, gibt das System eine Fehlermeldung aus. Eine Fehlerbehandlung ist in dieser Version für diesen speziellen Fall nicht vorgesehen. Auch in der darauffolgenden Schicht nach erfolgreicher Ausführung der VO ist wieder zu differenzieren, ob dieser Pfad zum ersten Mal seit Start der Funktion behandelt wird oder bereits bei vorherigen Frames durchlaufen wurde. Eine Aussage zu diesem Status liefert ebenfalls eine eigene Kontrollvariable. Sie springt von Null auf Eins, wenn bei Ersteintritt des Roboters in diesen Zustand seine Anfangsposition über eine vorgegebene Anzahl an Frames gemittelt wurde. Während dieses Vorgangs bewegt sich der Roboter nicht.
Durch die Einteilung der anfänglichen Lage in Fern-, Mittel- und Nahfeld wird entschieden, wie und unter welchen Randbedingungen die eigentlichen Fahrstrategien (siehe lila eingefärbte Felder in Abbildung 5.1) auszuführen sind. Innerhalb dieser Bereichseinteilung anhand der Anfangsposition werden schließlich nochmals Fern-, Mittel- und Nahfeld abgegrenzt, um die aktuelle Position einzuordnen.
Die Fahrstrategien erhalten ihre Verschiedenheit anhand der Festlegung der Regelgröße für die Lageregelung. Streng genommen handelt es sich somit um Steuerstrategien.
Für den Nahbereich (“y1 < boundary_2”) definiert der Programmcode, dass beide Motoren mit gleicher Geschwindigkeit rückwärts drehen falls die Lagetoleranzen nicht eingehalten sind und der Roboter somit nicht die Freigabe erhält geradeaus in die Lücke zu fahren. Hier wird außerdem ein Zähler implementiert, welcher dem Roboter die Zeitspanne mehrere Frames bereitstellt, um seine Position im Stand nachzujustieren. Angewendet wird dabei der nachfolgend vorgestellte Regelungsalgorithmus des Mittelfelds mit der Vorgabe einer Sollgeschwindigkeit von Null, aber einer relativ großen Lenkungsverstärkung.
Die Strategie für das Mittelfeld (“boundary_2 < y1 < boundary_1”) stellt das Zusammen-

spiel zwischen der Vorgabe einer Referenztrajektorie und der VO dar. Wie bereits in Kapitel 4.4 erläutert, wird die Solltrajektorie für diesen Bereich nur einmalig bei Überschreiten der Bereichsgrenze von fern zu mittel im Stillstand berechnet. Hierfür ist widerrum eine Kontrollvariable vorzusehen, welche dies überwacht. Bestimmt werden bei diesem Schritt die Parameter der parabolischen Bahnkurve. Mithilfe dieser nun volldefinierten Funktion lassen sich in den folgenden Frames für jede neue Position des Fahrroboters die Sollwerte für die x-Koordinate als auch die momentane Steigung der Kurve (entspricht der Orientierung) abhängig von dem momentan y-Wert berechnen. Da das gesamte Fahrmanöver mit einer konstanten Vorwärtsgeschwindigkeit absolviert wird, interessieren hier für die Regelung nur der laterale Versatz (x-Koordinaten) und die Verdrehung (Kurvensteigung vs. Kameraausrichtung). Bei Aufenthalt im Fernfeld ("y1 > boundary_1") ist eine Kameraausrichtung auf den Ursprung des Welt-KS anzustreben, um zu gewährleisten, dass sich die Markierungen vor Beginn der Mittelfeld-Strategie im Fokus der Kamera befinden.

Die zwei zu unterscheidenden Regelungsalgorithmen sind in separaten Funktionen ausgelagert. Diese werden im Programmaufbau an der entsprechenden Stelle aufgerufen. Aus der evolutionären Entwicklung des Softwareprototyps sind zwei Regelstrategien hervorgegangen.

Der Algorithmus *control_farfield()* regelt die Kameraorientierung anhand einer PID-Reglerstruktur, vergleiche Kapitel 4.2, um eine erste Ausrichtung relativ zu der Zielfläche zu realisieren. Als Regelabweichung wird eine Winkeldifferenz im Bogenmaß gebildet. Darüber hinaus werden sowohl die konstante Vorwärtsgeschwindigkeit als auch die resultierende Stellgröße für die Lenkung mit Vorfaktoren multipliziert, um einen Richtungswechsel in Form der Negierung dieser Werte zu erzielen. Durch die weitere Unterteilung der translatorischen Verschiebung in x-Richtung in einen Nah- und Fernbereich wird die Möglichkeit geschaffen, unterschiedliche Übertragungscharakteristiken der Regelung zu applizieren. Vorrangig hiervon betroffen ist der Verstärkungsfaktor des PID-Reglers ($K_P(near)$ und $K_P(far)$). Somit sind zwei Parameter einzustellen, welche je nach Bedarf verschieden sein können, aber nicht müssen. Den Abschluss dieser Funktion stellt die Aktualisierung der Motordrehzahlen basierend auf den erhaltenen Stellgrößen dar.
Die Funktion *control_middlefield()* berücksichtigt zwei Regelgrößen. Sie nutzt die in Kapitel 4.4 definierte Funktion zur Vorgabe von Sollwerten für die x-Koordinate und die Kameraorientierung abhängig von der geschätzten y-Koordinate der Kamera. Hier wird ebenfalls ein PID-Regler implementiert, welcher zur Aufgabe hat, die Abweichung zwischen der Referenztrajektorie und der realen Position des Fahrroboters auszuregeln. Da nun zwei zu regelnde Größen vorliegen, welche in einer skalaren Stellgröße münden sollen, wird das in Kapitel 4.2 vorgestellte und durch Abbildung 4.12 untermauerte Prinzip einer Mehrgrößenregelung umgesetzt. Statt für jede Größe eine eigene Stellgröße an die Aktuatorik weiterzugeben, werden die beiden Ausgänge des Reglers gewichtet und addiert. Besonders zu beachten bei der Bestimmung des lateralen Versatzes relativ zur Sollvorgabe ist eine Normierung dieser Größe auf einen Bezugswert. Dadurch sollte möglichst vermieden werden, dass diese Regelabweichung einen Wert > 1 annimmt. Bei einer Abweichung von 1 cm und einer ungünstigen Wahl der Reglerparameter könnten sonst die Motoren bereits mit ihrer maximalen Drehzahl (entspricht = 1) angesteuert werden. Als geeigneter Bezugswert wird der Absolutwert des Translationsvektors, welcher den Abstand in x zur Mittellinie des Zielbereichs ausgibt, festgelegt. Um ungültige Divisionen durch Null

zu verhindern, besitzt dieser eine vorgegebene untere Grenze von Eins. Alle weiteren Funktionsmerkmale sind analog zu der *control_farfield*-Regelstrategie, wie die Einstellung einer Sensitivität über zwei unterschiedliche Verstärkungsfaktoren, die Ansteuerung der Motoren als auch die Verwendung von Vorfaktoren für die Geschwindigkeits- und Lenkungsvorgabe.

Um den Inhalt dieses Kapitels abzurunden, ist der zu erwartende Bewegungsablauf des Fahrroboters anhand des entwickelten Programms unter Integration aller zuvor eigenständig entwickelten Module zu präsentieren. Hierbei wird allerdings nur auf das tatsächliche Rangiermanöver eingegangen (Annahme: Zielobjekte erkannt und Position durch VO bestimmt) und die Suchstrategie als auch die reine Vorwärtsfahrt sind außen vor.

Fahrroboter zuerst im Fernfeld beobachtet (y0 > boundary_1):
Der Roboter erfährt zunächst eine Ausrichtung auf den Ursprung des Welt-KS, bis er das Mittelfeld erreicht. Genutzt wird hierfür die Regelstrategie *control_farfield*. Ihr werden ein negierter Referenzwinkel, gebildet durch den Arkustanges des Translationsvektors in x- und in y-Richtung als Sollwert und der Euler-Winkel Φ als Regelgröße jeweils im Bogenmaß übergeben. Die Vorfaktoren für die Geschwindigkeit und die Lenkung sind beide positiv und gleich Eins. Bei Überschreiten der Bereichsgrenze wird die Fahrstrategie gewechselt. Der Roboter hält zunächst kurz an, um seine aktuelle Position zu mitteln und darauf aufbauend die Solltrajektorie innerhalb des Mittelfelds zu berechnen. Nun wird die *control_middlefield* eingesetzt mit gleichen Vorfaktoren wie zuvor im Fernfeld. Benötigte Angaben sind die x-Koordinate und die Steigung der Trajektorie für die momentane Koordinate y1 des Roboters. Zu regelnde Größen sind die x-Koordinate der Kameraposition und die Orientierung bestimmt durch Φ (Drehung um Hochachse). Die Geschwindigkeit bleibt konstant. Erwartet wird eine Fahrt des Roboters möglichst exakt entlang dieser Referenzbahnkurve, bis seine y-Koordinate die zweite Bereichsgrenze *boundary_2* unterschreitet. An dieser Stelle sollte sich der Fahrroboter idealerweise bereits zentral auf der Mittellinie des Ziels ohne Verdrehung befinden. Auch bei dieser Bereichsüberschreitung bleibt der Roboter stehen und mittelt seine Position über mehrere Frames. Während dieser Wartezeit wird er dazu befähigt, seine Position durch Drehung auf der Stelle nachzujustieren. Ist schließlich die Abweichung von der Sollposition unterhalb der oberen Grenzen, wird die Freigabe der Vorwärtsfahrt erteilt. Andernfalls wird unter anderem die Kontrollvariable zur Bestimmung der Anfangsposition auf ihren Initialwert zurückgestellt, der Roboter lokalisiert sich im Nahfeld der Markierungen und führt die Nahfeld-Strategie, wie weiter unten beschrieben, aus. Dadurch wird ein Korrekturmanöver eingeleitet.

Fahrroboter zuerst im Mittelfeld beobachtet (boundary_1 > y0 > boundary_2):
Hier führt der Roboter zu Beginn im Mittelfeld die *control_farfield* rückwärts aus mit negativen Vorfaktoren für die Geschwindigkeit als auch die Lenkung. Nach Eintritt in den Fernbereich wird die Kontrollvariable, welche über die Vorwärts- oder Rückwärtsbewegung entscheidet auf Eins gesetzt und alle weiteren Schritte sind analog zu obiger Strategie.

Fahrroboter zuerst im Nahfeld beobachtet (y0 < boundary_2):
Über die entsprechenden Kontrollvariablen wird eine Rückwärtsfahrt angeordnet, wenn die Position nicht innerhalb der vorgegebenen Grenzen liegt. Solange sich der Roboter im Nahfeld befindet, erfolgt bei dieser reinen Rückwärtsfahrt keine Regelung. Ab Errei-

chen des Mittelfelds wird die gleiche Strategie wie bei anfänglicher Positionsbestimmung im Mittelfeld angewendet.

Als geeignete Reglerparameter innerhalb der Version 2 sind einzustellen:
$K_{P,far} = 0.05; K_{P,near} = 0.6; K_D = 0.02; K_I = 0.01;$ weight_grad $= 0.6;$ weight_transl $= 0.4;$ steering gain (v = 0) $= 28$

Die Initialisierung der Funktion wird mit folgenden Parametern empfohlen:
Schwellwert Objektwahrscheinlichkeit $= 0.5;$ Min. Frame count $= 3;$ Schwellwert $\Delta x = 20\,cm;$ Schwellwert $\Delta\Phi = 20\,^{\circ}$

Bei einer konstanten Geschwindigkeit der Motoren zwischen 0,2 und 0,25 · U_max und einem Abstand der Markierungen von 18 cm werden als Bereichsgrenzen (entspricht y-Koordinate im Belegungsnetz / in Vogelperspektive) vorgeschlagen:
boundary_1 $= 97\,cm$ (Abstand zu Einfahrtslinie = 60 cm) und
boundary_2 $= 62\,cm$ (Abstand zu Einfahrtslinie = 25 cm)

Bereits in Version 2 umgesetzte Optimierungen beinhalten eine Hysterese zwischen den Bereichsgrenzen als auch die Programmierung eines “Gedächtnisses” der Funktion.
Da bei der VO die geschätzte Kameraposition starken Schwankungen unterliegt, kann der Sonderfall eintreten, dass die bestimmte Position zwischen einzelnen aufeinanderfolgenden Frames entlang einer Bereichsgrenze oszilliert. Bemerkbar ist dieses Phänomen dadurch, dass der Roboter an einer Stelle hängen bleibt, da er periodisch widersprüchliche Befehle erhält. Eine Hysterese trägt zur Lösung dieses Konflikts bei. Sie wird im Code implementiert, indem die Bereichsgrenze nach erstmaligem Überschreiten um einen konstanten Wert (z.B. 5 cm) verschoben wird. Dadurch wird die Wahrscheinlichkeit reduziert, dass in dem darauffolgenden Frame der Roboter fälschlicherweise in dem angrenzenden Bereich lokalisiert wird.
Damit außerdem die Ausführung des Rangiermanövers nicht sofort unterbrochen wird, sobald in einem Frame kein Objekt zu erkennen ist, erfolgt eine Nutzung der Variablenwerte der x-Komponente des Translationsvektors. Dieser gibt den Wert basierend auf dem zuletzt detektierten Landmarks wieder. Je nach Vorzeichen dieses Elements wird entweder eine kurze Drehung im USZ oder entgegen des UZS angestoßen, um eine Ausrichtung in die vermutete Lage des Zielbereichs einzuleiten. Eine Steigerung der Robustheit gegenüber eines frühzeitigen Abbruchs des Manövers wird außerdem erreicht, indem auch vor Übergang in den Suchmodus eine gewisse Anzahl an Frames gewartet wird. Somit müssen zum Beispiel zuerst in drei aufeinanderfolgenden Kamerabildern keine Objekte erkannt werden, bevor die Suchstrategie ausgeführt wird (vergleiche “Erster Eintritt in Suchstrategie” in Abbildung 5.1).

Die Ausführung der in diesem Abschnitt vorgestellten Fahrfunktion ist innerhalb der Zeitspanne zwischen zwei aufeinanderfolgenden Frames bei einer vorgegebenen Aktualisierungsrate von 5 FPS stets abgeschlossen. Aufgrund der zu erwartenden niedrigen Geschwindigkeiten während des Fahrmanövers ist diese Aktualisierungsrate als ausreichend zu betrachten. Es ist somit der Nachweis erbracht, dass die Ausführungszeit (engl.: Execution Time) dieser Funktion unter 200 ms liegt, wobei auch ein gewisser Rechenanteil der nicht zwingend erforderlichen Visualisierung der Modelloutputs zuzuweisen ist.

5.3 Validierung und Integrationstests

Sowohl die Version 1 (Kapitel 5.1) als auch die finale Version 2 (Kapitel 5.2) stellen valide Lösungen der in Kapitel 1.2 festgesetzten Fahraufgabe dar. Allerdings erfüllt Version 1 nicht die Randbedingung, dass der Zielbereich möglichst zentral über die Einfahrtsöffnung befahren wird. Dieses Verhalten ist unerwünscht. Die Version 1 wird keinen weiteren Tests unterzogen, aufgrund der fehlenden Eignung für den späteren möglichen Anwendungsfall innerhalb des U-Shift-Fahrzeugkonzepts.
Dieses Kapitel stellt die Basis für die spätere Evaluierung der Übertragbarkeit auf das *U-Shift* in Kapitel 6 bereit. Neben der allgemeinen Funktionsvalidierung wird Version 2 als Endergebnis dieser Arbeit gezielten Integrationstests unterzogen. Diese sollen das Zusammenspiel der Softwarekomponenten anhand der Positioniergenauigkeit im Zielpunkt und der Robustheit gegenüber Umweltänderungen oder Störobjekten bewerten.

Tests zur Positioniergenauigkeit:
Die erste Testprozedur konzentriert sich auf die resultierende Abweichung von der Zielposition (Mittelpunkt des Zielbereichs und Ausrichtung kollinear zur Mittellinie) nach erfolgreichem Abschluss des Rangiermanövers. Abbildung 5.2 enthält die Ergebnisse dieses speziellen Integrationstests. Hierfür werden 15 Testfälle geschaffen, welche unterschiedliche Anfangspositionen des Roboters relativ zu dem Target berücksichtigen. Eine Beschreibung des jeweiligen Testfalls ist ebenfalls der Abbildung 5.2 zu entnehmen. Für jeden einzelnen Test sind 10 Wiederholungen vorgesehen, um schließlich das statistische Mittel zusammen mit der Standardabweichung der betrachteten Größen abzuleiten. Dazu zählen die Anzahl an benötigten Korrekturen, der laterale Versatz in cm und die Verdrehung in ° im Endpunkt. Zu beachten ist, dass es sich hierbei um eine begrenzte statistische Probe handelt, da eine ausgiebige Testphase den Rahmen dieser Arbeit überschreiten würde. Die so gewonnenen Resultate sind somit als erste Einschätzung der Leistungsfähigkeit der Version 2 der Rangierfunktion zu verstehen.
Das Testdesign deckt alle theoretisch denkbaren Anfangspositionen durch eine Eingliederung der kontinuierlichen Werte für die x-, y-Koordinate und die Drehung um die Hochachse bei der Bewegung in der Ebene in grob definierte Wertebereiche ab. Die hier gewählten Wertegrenzen von 30 cm bei dem lateralen Versatz (x-Richtung) von der Mittellinie des Zielbereichs und 20 ° bei der Verdrehung resultieren aus Erfahrungswerten. Analog zu der Klassifizierung der Fahrstrategien in Kapitel 5.2 erfolgt eine ungefähre Abstandsvorgabe anhand der Bereiche Fern-, Mittel- und Nahfeld. Alle Tests sind unter gleichen Umweltbedingungen - Markierungen auf Boden mit konstantem Abstand (= 18 cm), Büroumgebung (dunkler Teppichboden, weißer Hintergrund) und Tageslicht - durchzuführen. Die eingestellten Funktionsparameter und Rahmenbedingungen der hier aufgeführten Tests werden zur Reproduktion tabellarisch in Anhang 9.2 bereitgestellt.

Eine Interpretation der Testergebnisse liefert die Erkenntnis, dass keine eindeutige Abhängigkeit der Positioniergenauigkeit im Zielpunkt von der anfänglichen Roboterposition vorzuliegen scheint. Die Ergebnisse lassen keine klare Tendenz erkennen, bei welchen Anfangspositionen die Fahrfunktion am genauesten arbeitet. Es ist jedoch festzustellen, dass die Leistungsfähigkeit des automatisierten Rangiermanövers stark sinkt, sobald der Abstand zur Mittellinie (lateraler Versatz) zu groß wird. Dieses Phänomen lässt sich mit der implementierten Trajektorienplanung in Version 2 erklären. Durch die statische Berechnung der Funktionsparameter eines parabolischen Ansatzes entstehen bei großen

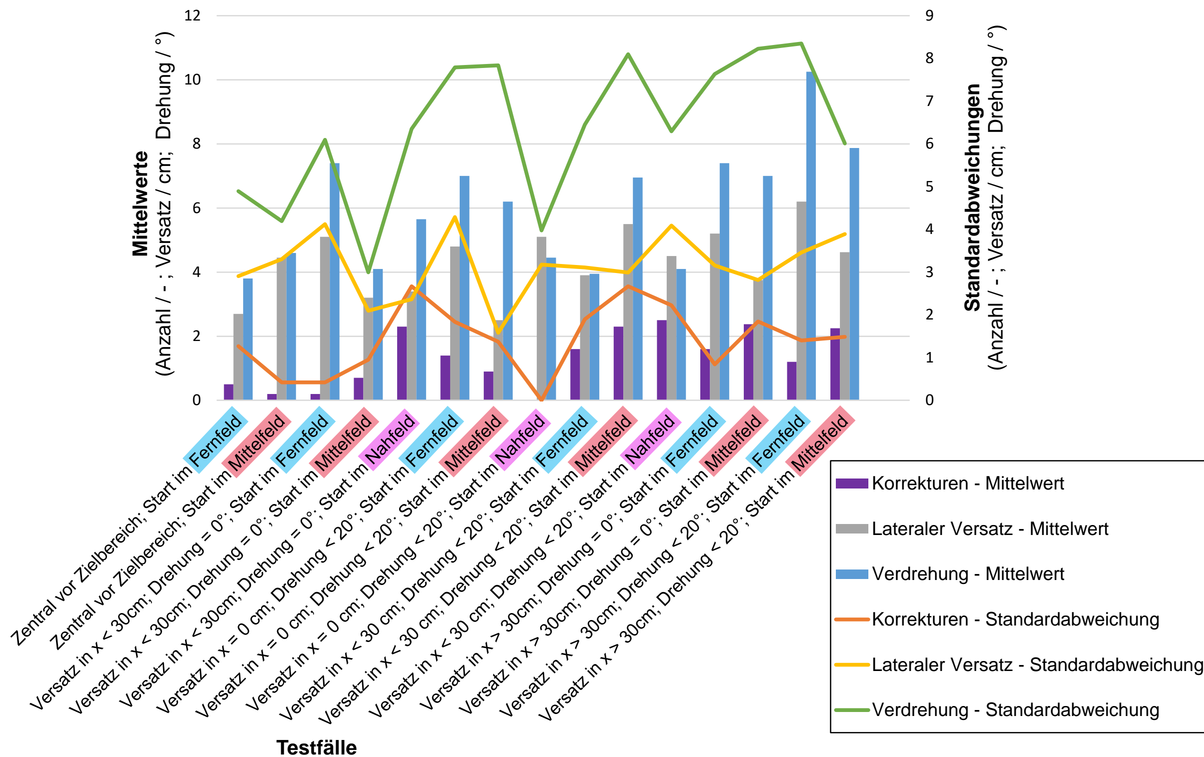


Abb. 5.2: Integrationstest zur Positioniergenauigkeit der Funktionsversion 2 mit Detektionsmodell ResNet_V211 (07.12.2022) in Büroumgebung bei Tageslicht [eigene Darstellung]

lateralen Versätzen steile Kurven, welche dazu führen, dass die Markierungen außer Sichtweite geraten. Die Version 2 besitzt zwar ein programmiertes Gedächtnis der letzten bekannten Ego-Position, entsprechend derer versucht wird, die Markierungen wieder einzufangen. Allerdings erfolgt bei der bisherigen Implementierung anschließend eine neue Schätzung der Anfangsposition und die zuvor ausgeführte Fahrstrategie wird nicht weiter behandelt. Diese Inkonsistenz, welche in vielen Korrekturfahrten resultiert, ist in zukünftigen Weiterentwicklungen der Version 2 zu beseitigen.
Weiterhin auffällig ist die große Streubreite (grüne Kurve in Abbildung 5.2) bei der Verdrehung im Zielpunkt. Aus Beobachtungen während der Testdurchführung geht hervor, dass diese enormen Schwankungen von bis zu 15 ° zumeist aus der Feinjustierung der Kameraorientierung kurz vor der Einfahrt in den Zielbereich resultieren. Die notwendige hohe Verstärkung der Steuergrößen für die Lenkung, um die Trägheit des Roboters im Stillstand zu überwinden und der stark begrenzte Zeitrahmen für die Ausführung der letzten Ausrichtung können zu starken Überschwingern führen. Da nach abgeschlossener Feinjustierung nur noch eine Geradeausfahrt möglich ist, ruft ein vorhandener Winkelfehler ebenfalls einen lateralen Versatz im Ziel hervor. Um diesem Verhalten gegenzusteuern, ist für die Feinjustierung ein längerer Zeitraum vorzusehen oder der Verstärkungsfaktor exakter einzustellen.

Als allgemeine Schlussfolgerungen dieses ersten Genauigkeitstests sind folgende Punkte festzuhalten:

- Der durchschnittliche laterale Versatz liegt bei allen Testfällen stets unter 8 cm.
- Die durchschnittliche Verdrehung ist hingegen immer kleiner als 12 °.

- In allen Testfällen wurden durchschnittlich nicht mehr als 4 Korrekturmanöver benötigt.
- Die Streuung aller drei Größen ist sehr groß.

Somit erzielt die Fahrfunktion Version 2 keine vorhersehbare und reproduzierbare Endposition, welche sich ohne größere Ausreißer innerhalb bestimmter Toleranzen bewegt. Der Unterdrückung dieser Schwankungen sollte bei einer zukünftigen Optimierungsarbeit besondere Aufmerksamkeit zuteil kommen.

Um den Einfluss der essentiellen Objektdetektion (bzw. Perception) auf die Genauigkeit des Softwareprototypen zu beurteilen, sind beide DL-Modelle zu verwenden. Die oben diskutierten Ergebnisse werden mit Version 2 erzielt, welche laut Spezifikation in Kapitel 5.2 das Detektionsmodell ResNet_V211 (07.12.2022), trainiert auf dem großen Datensatz, implementiert. Als Erweiterung des Genauigkeitstests erfolgt zusätzlich die Initialisierung des ResNet-Modells mit den Kantengewichten, optimiert auf der Bildersammlung vom 13.11.2022. Die Testdurchführung ist jedoch frühzeitig abzubrechen, da eine hohe Neigung zu Fehldetektionen in der Perception-Schicht die Leistungsfähigkeit der gesamten Funktion stark beeinträchtigt. Auch die stark abweichenden geschätzten Positionen durch visuelle Odometrie von der realen Position führen zu unzulässig vielen Korrekturen und Unterbrechungen des Rangiermanövers. Auffällig sind die Fehldetektionen dieses Modells bei der Suchstrategie. In der festgelegten Büroumgebung für die Integrationstests erkennt der Algorithmus unter anderem ein Paar weißer Schuhe oder Stuhl- und Tischbeine fälschlicherweise als Zielbereich.
Trotz fehlender zahlenmäßiger Bestätigung ist anhand der geschilderten Beobachtungen als Fazit aufzuführen, dass der Umfang und die Vielfältigkeit der Trainingsdaten des eingesetzten Detektionsmodells einen deutlichen Einfluss auf die generelle Funktionalität als auch die Positioniergenauigkeit der integrierten Softwarefunktion besitzt. Eine Implementierung des Modells mit der kleinen zugrunde gelegten Datenbasis vom 13.11.2022 würde dementsprechend zu keinem validen Endergebnis führen.

Tests zur Robustheit:
Zur detaillierten Analyse der Störanfälligkeit der vorläufigen Endversion des Softwareprototyps wird diese weiteren Tests unterzogen. Konstant bleiben die zuvor bereits verwendeten und in Anhang 9.2 aufgezeigten Einstellungen und Randbedingungen. Zu variieren sind der Untergrund und der Hintergrund des durch die beiden Markierungen festgelegten Zielareals. Außerdem werden gezielt Störobjekte mit zu den verwendeten Markierungen ähnlichen Erscheinungsformen in das Sichtfeld der Kamera eingebracht. Da bereits aus dem vorangegangenen Test zur Positioniergenauigkeit bekannt ist, dass die Güte des Detektionsmodells die Robustheit beeinflusst, werden beide ResNet-Modelle der Untersuchung unterzogen. Die Bewertung findet qualitativ anhand der Detektionsergebnisse der Perception-Schicht statt, da diese Daten als Input für alle anderen Softwarekomponenten dienen. Um die Übersichtlichkeit zu wahren, werden die visuellen Testergebnisse im Anhang unter 9.3 aufgeführt.

Nachfolgende Testfälle sind zu unterscheiden:

(01) Sonneneinstrahlung

(02) Weißer, homogener Untergrund bei den Markierungen

(03) Farbiger und zweigeteilter Untergrund bei den Markierungen

(04) Gelber, homogener Untergrund bei den Markierungen

(05) Gelber, homogener Hintergrund bei den Markierungen

(06) Farbiger und zweigeteilter Hintergrund bei den Markierungen

(07) Dunkler, homogener Hintergrund bei den Markierungen

(08) Kleine gelbe und rechteckige Störobjekte im Sichtfeld (z.B. Post-Its)

(09) Weitere gleichartige und zufällig positionierte Markierungen im Sichtfeld

(10) **Härtetest:** Mehr als zwei Markierungen parallel auf dem Boden nebeneinander

Der folgende Abschnitt beinhaltet eine kurze Interpretation und Einschätzung dieser gewonnenen Testdaten zur Robustheit der Detektionsmodelle.
Wie zu erwarten, ist die Funktionalität bei direkter Sonneneinstrahlung (Testfall 01) nicht gegeben. Beide Modelle zeigen, dass die Landmark Detektion relativ unempfindlich gegenüber der Gestaltung des Untergrunds ist. Bei zu geringem Kontrast zwischen dem Untergrund und den Markierungen (vergleiche Testfall 04) werden diese zwar noch von den Modellen erkannt, jedoch sehr ungenau.
Die Ergebnisse der Testfälle 05 bis 07 lassen Rückschlüsse auf eine ausgeprägte Störanfälligkeit bezüglich des Hintergrunds zu. Auch hier beeinflussen vor allem der gelbe Hintergrund als auch die Verwendung anderer heller Farben die Leistungsfähigkeit negativ. Dieses Phänomen ist auf die Trainingsdaten der Detektionsmodelle zurückzuführen. Da beide Modelle sowohl für auf dem Boden platzierte und an einer Wand angebrachte Markierungen trainiert wurden, reagieren die Detektoren auch sehr sensibel auf die Gestaltung der senkrechten Wand hinter den auf dem Boden liegenden Markierungen. Dieses große Störpotenzial ließe sich durch gezieltes Training der Modelle mit nur einer Form der Markierungsanordnung verringern.
Bei dem Testfall 07 erzielen beide Modelle sehr gute Annäherungen der Lage der Landmarks. Es sind sogar teilweise bessere Resultate als bei einer weißen homogenen Wand als Hintergrund, welche auch stets in den Trainingsdaten verwendet wurde, zu verzeichnen. Eine mögliche Erklärung hierfür ist der höhere Kontrast eines dunklen Hintergrunds zu den gelben Markierungen, wodurch diese besser detektierbar sind. Anderseits spielt möglicherweise auch die Reflektivität der Oberfläche eine Rolle, da eventuelle Spiegelungen der Markierungen im Hintergrund eine Störquelle darstellen. Eine nähere Untersuchung dieses speziellen Kriteriums ist nicht Inhalt dieser Arbeit.
Eine deutliche Verschlechterung der Funktionalität ist bei der Verwendung gleichartiger Markierungen als gezielte Störobjekte (Testfall 09) zu beobachten. Das Modell, trainiert auf dem kleineren Datensatz, versagt hier komplett und liefert keine brauchbaren Ergebnisse mehr. Bei dem Vergleichsmodell ResNet_V211 (07.12.2022) äußert sich der Störeinfluss durch eine Verschiebung der vorhergesagten Punkteposition hin zu dem in unmittelbarer Nachbarschaft senkrecht angebrachten Markierungsstreifen. Dagegen zeigen sich bei kleinen gelben Klebezetteln im Sichtfeld (Testfall 08) bei beiden Modellen keine Auswirkungen.
Werden nun statt bisher zwei parallelen Markierungen insgesamt vier Stück im Sicht-

feld der Kamera platziert und somit theoretisch drei Zielbereiche definiert (Testfall 10), so versucht das Modell mit geringerem Erfahrungsschatz stets die Eckpunkte des mittleren Bereichs zu approximieren. Bei Bereitstellung eines größeren Trainingsdatensatzes zeigt das resultierende Modelle diesbezüglich Fortschritte, indem tendenziell der näheste Zielbereich fokussiert wird. Durch eine Ergänzung der Trainingsdaten mit genauen Abstandsinformationen und Situationen mit mehreren möglichen Zielbereichen ließe sich theoretisch auch ein selektives Verhalten des Detektionsmodells realisieren, sodass weitere Markierungen im Sichtfeld keine Störung mehr verursachen.

Zusammenfassend bestätigt dieser Test, dass eine Verdoppelung des Umfangs der Trainingsdaten (hier von 5192 auf 13244 Bilder) einer DL-basierten Objekterkennung die Robustheit deutlich erhöht. Bei beiden Modellen wurden 80% der Bilderansammlung für das eigentliche Training verwendet. Somit besitzt das Modell ResNet_V211 (07.12.2022) einen um 155 % größeren Erfahrungsschatz als das vergleichbare Modell. Ist in den Daten konsistent die gleiche Positionierung der Markierungen zur Festlegung des Zielbereichs aus unterschiedlichen Perspektiven enthalten, sind die trainierten Detektionsmodelle höchstwahrscheinlich sehr robust gegenüber Änderungen des Untergrunds als auch des Hintergrunds. Die größte Herausforderung für die Funktionsfähigkeit der Version 2 stellen Situationen dar, in denen mehr als zwei solcher gelber Markierungen im Kamerabild zu erkennen sind. In diesen Fällen ist kein robustes Verhalten zu erwarten und ein Ausfall der gesamten Funktion denkbar. Außerdem bedarf es für solche Fehlinterpretationen der Umgebung eines Sicherheitskonzepts, mindestens in Form einer Plausiblitätskontrolle, um ungewollte Fahrmanöver des Roboters zu vermeiden. In dieser Arbeit ist die Entwicklung und Implementierung solcher Absicherungskonzepte allerdings nicht vorgesehen.

6 Übertragbarkeit auf das U-Shift-Fahrzeugkonzept

Den Abschluss der Entwicklungsarbeit stellt eine Evaluierung der Übertragbarkeit des in Kapitel 5.2 vorgestellten Softwareprototypen der Fahrfunktion auf den eingangs beschriebenen, zu automatisierenden Kapselwechsel des *U-Shift* dar, vergleiche Kapitel 1.1. Herangezogen werden hierfür die Testergebnisse aus Kapitel 5.3.

Hardware-Setup:
Zu klären ist, welche Anpassungen vorzunehmen sind, wenn eine Implementierung der Fahrfunktion auf einem funktionsfähigen Fahrzeugprototypen im Maßstab 1:1 angestrebt wird. In erster Linie ist dann nicht mehr der innere Bereich der U-förmigen Antriebseinheit als Zielareal zu verstehen, in welches der Fahrroboter - eine bewegliche Kapsel darstellenden - einfährt. Die Ausgestaltung dieses U-förmigen Targets wurde in dieser Arbeit durch den Einsatz von flachen, rechteckigen Markierungsstreifen realisiert. Für das U-Shift-Konzept ist jedoch vorgesehen, dass die Driveboards als bewegliches Modul die abgestellten und somit statischen Transportkapseln ansteuern und aufnehmen, vergleiche Kapitel 1.1.
Die Anzahl als auch Verteilung der Kameras über dem Driveboard ist so abzustimmen, dass bestenfalls das gesamte Fahrzeugumfeld erfasst wird. Vor allem eine weiträumige Überwachung der frontalen als auch rückseitigen Umgebung ist essentiell, um sowohl die Suchstrategie während einer Vorwärtsfahrt als auch das rückwärts auszuführende Rangiermanöver zu realisieren. Seitig angebrachte Kameras können zusätzlich das Sichtfeld durch eine Sensorfusion erweitern. Wie bei aktuellen Ansätzen zu autonomen Parkfunktionen, vergleiche Kapitel 3.2.2, lässt sich daraus ein Bild der Umgebung in Vogelperspektive ableiten, in welchem Objekte detektiert werden können. An der frontalen Stirnfläche des Sockels der Transportkapseln ließen sich durch eindeutige Marker vier koplanare Landmarks definieren. Diese müssen eine Symmetrie zur Mittellinie dieser Stirnfläche besitzen, um als Referenz für die Ausrichtung des Driveboards relativ zu der Kapsel zu dienen. Weiterhin ist auch die Festlegung von vier Referenzpunkten auf dem Boden vor der Kapsel oder einer festen Wand hinter dem Abstellplatz denkbar. Hierbei müsste jedoch stets sichergestellt werden, dass die Kapsel ebenfalls innerhalb dieses markierten Bereichs mittig und gerade abgestellt wurde. Eine Ausrichtung des Driveboards anhand von direkten Referenz zu der anvisierten Kapsel ist aus diesem Grund robuster und zu bevorzugen.

Eine umfangreiche Adaption ist hinsichtlich der Ansteuerung der für die Fahrfunktion relevanten Aktuatorik vorzunehmen. Zwar ist für das *U-Shift* ebenfalls ein elektrisches Antriebssystem vorgesehen, jedoch wird das Driveboard vier Räder mit einer standardmäßig gelenkten Vorderachse besitzen. Die Querdynamik ist dann durch Vorgabe eines Lenkwinkels zu steuern und nicht mehr durch individuelle Ansteuerung einzelner angetriebener Räder. Außerdem verfügt ein solcher Fahrzeugaufbau nicht über die Möglichkeit, eine Drehung auf der Stelle durchzuführen. Die Wendigkeit und die Fahrdynamik sind aufgrund des grundlegend verschiedenen Aufbaus des Fahrgestells definitiv nicht vergleichbar mit den Ausprägungen dieser Größen bei dem miniaturisierten Fahrroboter. Auch die voraussichtlich deutlich höhere Trägheit des Gesamtsystems ist bei einer späteren Implementierung zu berücksichtigen, besonders bei der Wahl der Reglerparameter.
Bei Ausarbeitung einer Rangierfunktion für den realen Kapselwechsel sind außerdem Odometriedaten aus Daten der ausgedehnteren fahrzeugeigenen internen Sensorik ab-

zuleiten und als Ergänzung zu der visuellen Odometrie einzusetzen. Dieses zu erwartende größere Angebot an Sensoren zur Erfassung des Fahrzustands ermöglicht eine Absicherung der Fahrfunktion in Form von Redundanzen, Fehlerkorrekturen oder Plausibilitätskontrollen. Weiterhin ist die Erweiterung der exterperceptiven Sensorik durch andere Sensortypen (z.B. LiDAR, Radar, Ultraschall, etc.) nicht auszuschließen, um unterschiedliche Messprinzipien für eine robuste und sichere Erfassung der Umgebung zu nutzen.
Der Inhalt der vorliegenden Arbeit zeigt auf, dass mit vergleichsweise geringen Rechenkapazitäten (CPU und GPU) mit einer maximalen Leistungsaufnahme von 5 W bereits die Ausführung des Softwareprototypen in Echtzeit möglich ist. Vorausgesetzt die Geschwindigkeit während des Rangiervorgangs ist im Bereich des Schritttempos und eine Aktualisierungsrate der Kamera von 5 FPS ist für das Echtzeitverhalten ausreichend. Sollten weitere größere neuronale Netze zur Bildverarbeitung parallel verwendet werden, so ist möglicherweise eine leistungsstärkere Grafikkarte im Realfahrzeug vorzusehen.

Infrastruktur:
Da als zentraler Bestandteil des U-Shift-Fahrzeugkonzepts auch der Ausbau der Infrastruktur zur Unterstützung autonomer Fortbewegungsmittel (MAD) verfolgt wird, ist dieser Aspekt hier zu berücksichtigen. Der im Rahmen dieser Arbeit verfolgte Computer Vision-Ansatz ließe sich in der Hinsicht abändern, dass die Daten der Perception-Softwareschicht nicht mehr von einer fahrzeugeigenen Kamera bereitgestellt werden, sondern von der Infrastruktur. Eine statisch installierte Kamera sendet über eine geeignete V2I-Kommunikationsschnittstelle bereits aufbereitete Informationen an das Automatisierungssystem des Driveboards. Über Anbringung von vier koplanaren Referenzpunkten am Driveboard lässt sich dessen Lage relativ zu der Kapsel ebenfalls über die homographiebasierte visuelle Odometrie aus den gelieferten Videodaten ableiten und das Rangiermanöver, wie in Kapitel 5.2 beschrieben, ausführen. Zu klären ist, welche Rechenschritte bei einem solchen Vorgehen in das Backend ausgelagert werden und ob eher eine zentrale oder dezentrale Datenverarbeitung angestrebt wird.
Ein möglicher Vorteil einer gezielten Unterstützung durch die Infrastruktur ist die Unabhängigkeit der Bilddaten von der Fahrzeugbewegung (Vibrationen, Schwingungen, Stöße, etc.). Das Driveboard ist nun als bewegtes Objekt innerhalb der Bilder zu detektieren. Die Kapsel lässt sich des Weiteren bei diesem abgeänderten Konzept zur Objektdetektion mit Computer Vision als statisches Kalibrierobjekt nutzen. Vor allem können durch weitere statische Kameras in der Infrastruktur Situationen abgedeckt werden, in denen die fahrzeugeigene Sensorik ausfällt oder deren Sichtfeld nicht ausreichend ist. Die Latenz der zusätzlich notwendigen Datenübertragung darf jedoch nicht zu groß sein, um weiterhin eine Ausführung der Fahrfunktion in Echtzeit zu gewährleisten.

Genauigkeitsanforderungen:
Aus einer Toleranzbetrachtung, basierend auf der Spezifikation des *U-Shift*, gehen Anforderungen an die Positioniergenauigkeit hervor. Diese werden schließlich mit der erreichbaren Genauigkeit der Version 2, untersucht in Kapitel 5.3, gegenübergestellt.
Die Spezifikation gibt einen lateralen Versatz der Antriebseinheit bezogen auf die Mittellinie des Zielobjekts während der Unterfahrung der Kapsel von maximal $^{+}/-$ 1 cm vor. Durch Zuhilfenahme einfacher geometrischer Zusammenhänge lässt sich außerdem die maximal zulässige Drehung der Kapsel um die eigene Achse, bezogen auf eine Parallele zur Längsachse des Driveboards, bestimmen, vergleiche Abbildung 6.1. Der Toleranzbereich für die Verdrehung α ist abhängig von der Länge L und der Breite B des Sockels zusam-

men mit der vorzugebenden Variable x (siehe Abbildung 6.1 (b) und Formel 6.4) und ist wie folgt herzuleiten:

$$\beta = arctan\left(\frac{B}{L}\right) \tag{6.1}$$

$$\gamma = arcsin\left(\frac{x+\frac{B}{2}}{\sqrt{\left(\frac{B}{2}\right)^2+\left(\frac{L}{2}\right)^2}}\right) \tag{6.2}$$

$$\gamma = \alpha + \beta \tag{6.3}$$

$$\alpha = arcsin\left(\frac{x+\frac{B}{2}}{\sqrt{\left(\frac{B}{2}\right)^2+\left(\frac{L}{2}\right)^2}}\right) - arctan\left(\frac{B}{L}\right),\ [\alpha] = \,^\circ \tag{6.4}$$

Es sei festgelegt, dass zwischen dem Driveboard und der Kapsel auf beiden Seiten jeweils konstruktiv 20 mm Luftspalt vorgesehen sind. Um ebenfalls Fertigungstoleranzen zu berücksichtigen, sollten insgesamt mindestens 2 mm vorgehalten werden. Wird eine Worst-Case-Betrachtung durchgeführt, indem der mögliche laterale Versatz von 10 mm miteinbezogen wird, dann muss gelten $x \leq 8\,mm\,(=\ (20\,mm\ -\ 10\,mm)\ -\ 2\,mm)$. Für B = 1340 mm und L = 4350 mm resultiert somit eine max. zulässige Drehung der Kapsel von $\alpha\ \leq\ \pm\,0,21$ °. Unter isolierter Anschauung der Verdrehung gilt $x\ \leq\ 18\,mm$ und der Wertebereich für α ergibt sich zu $\pm\,0,47$ °.

(a)
Max. lateraler Versatz und Verdrehung der Kapsel

(b)
Herleitung der zulässigen relativen Kapseldrehung aus der Geometrie

Abb. 6.1: Toleranzbetrachtung bei der hochpräzisen Positionierung des U-Shift Fahrzeugs während des Modulwechsels. [eigene Darstellung, nicht maßstabsgetreu]

Diese hochpräzise Positionierung kann mit der in dieser Arbeit entwickelten Fahrfunktion nicht sichergestellt werden. Hier sind durchschnittliche Versätze von ca. 8 cm und Verdrehungen von ca. 12 ° realistisch, siehe Testergebnisse in Kapitel 5.3. Diese unterliegen außerdem großen Schwankungen, sodass keine reproduzierbare hochpräzise Positionierung mit dem aktuellen Stand der Rangierfunktion zu erwarten ist. Vor allem die visuelle Odometrie ermöglicht bei dieser Ausarbeitung keine exakte Lokalisierung des Ego-Fahrzeugs im Welt-KS, wodurch auch die gewollte Endlage nur schwer durch abgestimmte Regel- und Fahrstrategien erreichbar ist. Diese Ungenauigkeiten sind vor einer Implementierung im Realfahrzeug in weiteren Optimierungsschritten zu minimieren.

7 Resümee und Ausblick

Die vorliegende Arbeit liefert einen Beitrag zur rein kamerabasierten Automatisierung von Fahrfunktionen unter Anwendung von Deep Learning in Form eines validierten Softwareprototypen. Eine Besonderheit stellt die Begrenzung der verfügbaren exteroperceptiven Sensorik auf eine monokulare Kamera dar. Die vorgestellte Funktion basiert auf einem modularen Automatisierungsansatz durch Einteilung der notwendigen Berechnungsschritte in in sich abgeschlossene Funktionsbausteine, vergleiche Kapitel 2.1. Zur Bewältigung eines möglichst präzisen Rangiermanövers wurden vier essentielle Softwarekomponenten identifiziert und zu verifizierten Ergebnissen ausgearbeitet, siehe Kapitel 4.
Die Umfelderfassung (engl.: Perception) ist rein auf die Erschließung des zu befahrenden Zielbereichs über die Detektion der vier markanten Eckpunkte ausgerichtet. Zur Definition dieses Targets werden zwei rechteckige gelbe Markierungsstreifen genutzt. Diese Landmark Detektion wird durch Anwendung eines CNNs mit ResNet50-Backbone realisiert. Eine manuelle Optimierung der Hyperparameter und das Training des neuronalen Netzes mit einem 13244 Bilder umfassenden, eigens erstellten Datensatzes münden schließlich in einer verifizierten punktbasierten Objektdetektion. Sie dient als essentielle Grundlage für alle weiteren Tasks.
Durch Implementierung eines PD-Regelungsalgorithmus und eine qualitative Lokalisierung des verwendeten miniaturisierten Fahrroboters anhand der vorhergesagten Lage dieser Referenzpunkte im Kamerabild wird der Aufbau der Version 1 der Fahrfunktion ermöglicht, vergleiche Kapitel 5.1. Mit diesem Algorithmus wird der Roboter bereits dazu befähigt, den Mittelpunkt eines festgelegten Bereichs rein mit zugrunde gelegten Bilddaten einer einzelnen frontalen Weitwinkelkamera anzusteuern.
Eine deutlich umfangreichere Funktionalität stellt die in Kapitel 5.2 diskutierte vorläufige Endversion (Version 2) bereit. Sie stellt eine Kombination aus visueller Objektdetektion (Kapitel 4.1.2), Trajektorienplanung (Kapitel 4.4), visueller Odometrie (Kapitel 4.3.2) und Einsatz eines universellen PID-Reglers zur Trajektorienverfolgung dar (Kapitel 4.2). Die visuelle Odometrie nutzt die Koplanarität der vier Landmarks, um über Computer Vision-Algorithmen die Lage der Kamera im Welt-KS abzuschätzen. Eine parabolische Solltrajektorie dient schließlich als Führungsgröße für die Regelung der ebenen Fahrzeugbewegung. Weiterhin enthält Version 2 mehrere Fahrstrategien, welche basierend auf der lokalisierten Position des Roboters relativ zu dem Ziel selektiert werden.
Die durchgeführten Integrationstests in Kapitel 5.3 mithilfe des Rapid Prototyping-Tools zeigen auf, dass Positioniergenauigkeiten innerhalb des Zielareals von ungefähr 8 cm Versatz zur Mittellinie und ca. 12 ° Drehung um die Fahrzeughochachse erreichbar sind. Jedoch sind hierfür teilweise noch mehrere Korrekturmanöver notwendig, bis die Lücke über die Einfahrtslinie befahren wird. Die Robustheit der Objektdetektion zeigt eine Abhängigkeit von der Größe des zur Verfügung gestellten Datensatzes während des Trainings auf und ist bei Version 2 positiv zu bewerten. Allerdings stellen Störquellen in Form von weiteren im Sichtfeld der Kamera befindlichen gelben Markierungsstreifen eine große Herausforderung für eine funktionierende Objektdetektion dar.
Eine Beurteilung der Übertragbarkeit auf die Automatisierung des Kapselwechsel des *U-Shift* des DLR Instituts für Fahrzeugkonzepte rundet diese Arbeit ab, siehe Kapitel 6. Sie zeigt auf, dass vor allem hinsichtlich der hardwareseitigen Vorgaben und der geforderten hochpräzisen Positionierung weitere Optimierungen und Anpassungen an dem entwickelten Softwareprototypen vorzunehmen sind.

Aus den Erkenntnissen, welche im Laufe der Entwicklung der Fahrfunktion mit dem Fahrroboter gewonnen wurden, kristallisierten sich hauptsächlich zwei Themengebiete mit großem Optimierungspotenzial heraus.
Einerseits sind die Genauigkeit und Robustheit des Landmark Detektionsmodells zu steigern. Sowohl die Detektion der Referenzpunkte im Betrieb als auch der Labeling-Prozess der Trainingsdaten ließen sich durch eine ausgereifte Wahl der Markierungen verbessern. Werden zum Beispiel einfache geometrische Formen, welche bereits eindeutig einen Punkt festlegen (z.B. punktförmige Marker oder Kreuze) verwendet, kann der Labeling-Prozess möglicherweise mit einfachen Bildverarbeitungsalgorithmen automatisiert werden. Es ist zu untersuchen, ob solche speziellen Marker überhaupt eines DL-Modells zur Regression der Punktkoordinaten im Kamerabild bedürfen und nicht bereits CV-Algorithmen diese Aufgaben zufriedenstellend erfüllen. Vor allem für die Lokalisierung durch visuelle Odometrie muss von der Perception-Schicht die Lage von vier koplanaren Punkten in jedem Frame zur Verfügung gestellt werden. Die restliche Gestaltung der Landmark Detektion unterliegt sonst keinen weiteren Vorgaben.
Bei weiterer Verfolgung des Deep Learnings sind die Hyperparameter in weiterführenden Arbeiten automatisch zu optimieren, um ein optimales Training zu ermöglichen. Auch das Design der Trainingsdaten bei Anwendung des überwachten Lernens ist ausbaufähig. Nicht nur der Umfang, sondern auch der Informationsgehalt der Bilddaten ist zu steigern. Eine umfangreiche Data Augmentation sollte hierbei in Betracht gezogen werden. Weiterhin ist eine ausgeprägte Robustheit erzielbar, wenn in dem gesamten Trainingsdatensatz stets dieselbe Anordnung der Marker verwendet wird. Dafür sind jedoch Einbußen hinsichtlich der Anpassungsfähigkeit hinzunehmen. Da der euklidische Abstand zu dem Zielbereich im Welt-KS eine entscheidende Größe für die Funktionsfähigkeit darstellt, bietet sich die Erweiterung der Trainingsdaten des DL-Modells mit Abstandsinformationen an. Während der Aufnahme des Datensatzes würde ein zusätzlicher LiDAR-Sensor genaue Ground Truth-Abstandswerte liefern, welche schließlich den Bildern im jeweiligen Label beigefügt werden. Das eingesetzte neuronale Netz kann nach dem Training dann ebenfalls zur Abstandsschätzung genutzt werden.
Die zweite große Stellschraube stellt die Ausarbeitung der Verknüpfung der einzelnen Funktionsbausteine bei der Integration dar. Neben einer Optimierung der Fahrstrategien ist vor allem das abstandsbasierte Konzept zu überarbeiten. Bedingt durch die Schwankungen in den Daten der VO sind so keine reproduzierbaren Fahrmanöver ausführbar. Auch der Trajektorienplanung ist gesondert Aufmerksamkeit zukommen zu lassen, da in dieser Arbeit bisher nur eine einfache statische Berechnung der Funktionsparameter einer Parabel umgesetzt wurde. Diese scheint nicht optimal im Zusammenspiel mit der VO, da sie genauere Abstandsinformationen benötigt.

Zuletzt ist eine Funktionserweiterung durch eine integrierte Hinderniserkennung bei der Weiterentwicklung der hier dargebotenen Fahrfunktion anzustreben. Ist das System zusätzlich in der Lage, bei erkannten Hindernissen entlang der geplanten Referenztrajektorie den risikominimalen Zustand einzunehmen oder sogar ein Ausweichmanöver auszuführen, werden die Sicherheitsanforderungen an den autonomen Betrieb des Realfahrzeugs adressiert.

Eine zielführende Anwendung der Softwarefunktion für den automatischen Kapselwechsel im Rahmen des U-Shift-Konzepts ist nach Abschluss weiterer Entwicklungsarbeiten aus technischer Sicht realisierbar.

8 Quellenverzeichnis

[1] "U-Shift: Information zu dem Fahrzeugkonzept," 2022, Zugriff am 18.10.2022. [Online]. Verfügbar: https://verkehrsforschung.dlr.de/de/projekte/u-shift

[2] Rinspeed AG, "Rinspeed AG - Snap," 2023, Zugriff am 08.01.2023. [Online]. Verfügbar: https://www.rinspeed.com/de/Snap_48_concept-car.html

[3] Continental AG, "Continental startet mit BEE in urbane Zukunftswelten," 15.09.2017, Zugriff am 08.01.2023. [Online]. Verfügbar: https://www.continental.com/de/presse/pressemitteilungen/bee/

[4] marsMediaSite, "Vision URBANETIC - Mercedes-Benz Group Media," 19.09.2018, Zugriff am 08.01.2023. [Online]. Verfügbar: https://group-media.mercedes-benz.com/marsMediaSite/de/instance/ko.xhtml?oid=41169005&relId=60829&resultInfoTypeId=40626#toRelation

[5] D. Gruyer, V. Magnier, K. Hamdi, L. Claussmann, O. Orfila, und A. Rakotonirainy, "Perception, information processing and modeling: Critical stages for autonomous driving applications," *Annual Reviews in Control*, Bd. 44, SS. 323–341, 2017, ISSN: 1367-5788.

[6] H. Fujiyoshi, T. Hirakawa, und T. Yamashita, "Deep learning-based image recognition for autonomous driving," *IATSS Research*, Bd. 43, Nr. 4, SS. 244–252, 2019, ISSN: 0386-1112.

[7] "Home - JetBot," 2022, Zugriff am 17.12.2022. [Online]. Verfügbar: https://jetbot.org/v0.4.3/index.html

[8] x. R. Team, "Deep-Dive: Autonomes Fahren," 03.08.2021, Zugriff am 02.01.2023. [Online]. Verfügbar: https://10xdna.com/de/expertise/autonomes-fahren-deep-dive/

[9] H. Unbehauen, *Regelungstechnik I*. Wiesbaden: Vieweg+Teubner, 2008, ISBN: 978-3-8348-0497-6.

[10] G. G. Slabaugh, "Computing Euler angles from a rotation matrix," 2020, Zugriff am 02.02.2023. [Online]. Verfügbar: http://eecs.qmul.ac.uk/~gslabaugh/publications/euler.pdf

[11] NVIDIA, "NVIDIA Jetson Nano-Entwicklerkit kaufen," 2022, Zugriff am 17.12.2022. [Online]. Verfügbar: https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-nano-developer-kit/

[12] M. Nolting, *Künstliche Intelligenz in der Automobilindustrie: Mit KI und Daten vom Blechbieger zum Techgiganten*, Serie Sachbuch. Wiesbaden and Heidelberg: Springer Vieweg, 2021, ISBN: 978-3-658-31567-2.

[13] M. Schindewolf, H. Guissouma, und E. Sax, "Analysis and Modeling of Future Electric/Electronic Architectures for Modular Vehicles Concepts," in *21. Internationales Stuttgarter Symposium*, Serie Proceedings, M. Bargende, H.-C. Reuss, und A. Wagner, Hrsgg. Wiesbaden: Springer Fachmedien Wiesbaden, 2021, SS. 32–46, ISBN: 978-3-658-33520-5.

[14] M. Münster, M. Brost, T. Siefkes, G. Kopp, E. Beeh, F. Rinderknecht, S. Schmid, M. Osebek, S. Scheibe, R. Hahn, D. Heyner, P. Klein, G. Piazza, C. Ulrich, W. Kraft,

F. Philipps, L. Köhler, M. Buchholz, T. Wodtko, K. Dietmayer, M. Frey, F. Weitz, F. Gauterin, H. Stoll, M. Schindewolf, H. Guissouma, F. Krauter, E. Sax, J. Neubeck, S. Müller, S. Eberts, M. Göldner, S. Teichmann, J. Kiebler, M. Saljanin, M. Bargende, und A. Wagner, “U-Shift II Vision und Project Goals,” in *22. Internationales Stuttgarter Symposium*, Serie Proceedings, M. Bargende, H.-C. Reuss, und A. Wagner, Hrsgg. Wiesbaden: Springer Fachmedien Wiesbaden, 2022, SS. 18–31, ISBN: 978-3-658-37010-7. [Online]. Verfügbar: http://www.springer.com/

[15] J. Schäuffele und T. Zurawka, *Automotive Software Engineering*. Springer Fachmedien Wiesbaden, 2016, ISBN: 978-3-658-11814-3.

[16] S. Rathour, V. John, M. K. Nithilan, und S. Mita, “Vision and Dead Reckoning-based End-to-End Parking for Autonomous Vehicles,” in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, SS. 2182–2187, ISBN: 978-1-5386-4452-2.

[17] A. R. Munappy, J. Bosch, und H. H. Olsson, “Data Pipeline Management in Practice: Challenges and Opportunities,” in *Product-Focused Software Process Improvement*, Serie Lecture Notes in Computer Science, M. Morisio, M. Torchiano, und A. Jedlitschka, Hrsgg. Cham: Springer International Publishing, 2020, Bd. 12562, SS. 168–184, ISBN: 978-3-030-64147-4.

[18] W. Shi, M. B. Alawieh, X. Li, und H. Yu, “Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey,” *Integration*, Bd. 59, SS. 148–156, 2017, ISSN: 0167-9260.

[19] M. Werner, *Digitale Bildverarbeitung: Grundkurs mit neuronalen Netzen und MATLAB®-Praktikum*. Wiesbaden: Springer Vieweg, 2021, ISBN: 978-3-658-22184-3. [Online]. Verfügbar: http://www.springer.com/

[20] H. Süße und E. Rodner, *Bildverarbeitung und Objekterkennung: Computer Vision in Industrie und Medizin*, Serie Lehrbuch. Wiesbaden: Springer Vieweg, 2014, ISBN: 978-3-8348-2605-3.

[21] O. Schreer, *Stereoanalyse und Bildsynthese: Mit 6 Tabellen*. Berlin and Heidelberg and New York: Springer, 2005, ISBN: 978-3-5402-3439-5.

[22] R. Hartley und A. Zisserman, *Multiple view geometry in computer vision*, second edition Ed. Cambridge: Cambridge University Press, 2003, ISBN: 978-0-521-54051-3. [Online]. Verfügbar: https://books.google.de/books?id=si3R3Pfa98QC

[23] Z. Kukelova, J. Heller, M. Bujnak, und T. Pajdla, “Radial distortion homography,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2015)*. Piscataway, NJ: IEEE, 2015, SS. 639–647, ISBN: 978-1-4673-6964-0.

[24] “OpenCV: Basic concepts of the homography explained with code,” 2022, Zugriff am 29.12.2022. [Online]. Verfügbar: https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html

[25] C. Hughes, M. Glavin, E. Jones, und P. Denny, “Wide-angle camera technology for automotive applications: a review,” *IET Intelligent Transport Systems*, Bd. 3, Nr. 1, S. 19, 2009, ISSN: 1751-956X.

[26] “OpenCV: Camera Calibration,” 2022, Zugriff am 28.12.2022. [Online]. Verfügbar: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

[27] M. Heimberger, J. Horgan, C. Hughes, J. McDonald, und S. Yogamani, "Computer vision in automated parking systems: Design, implementation and challenges," *Image and Vision Computing*, Bd. 68, SS. 88–101, 2017, ISSN: 0262-8856.

[28] R. Szeliski, *Computer Vision*. Cham: Springer International Publishing, 2022, ISBN: 978-3-030-34371-2.

[29] S. Tuohy, D. O'Cualain, E. Jones, und M. Glavin, "Distance determination for an automobile environment using inverse perspective mapping in OpenCV," in *IET Irish Signals and Systems Conference (ISSC 2010)*. IET, 23-24 June 2010, SS. 100–105, ISBN: 978-1-8491-9252-1.

[30] P. Schiekofer, Y. Erdogan, S. Schindler, und M. Wendl, "Maschinelles Lernen für das automatisierte Fahren," *ATZ - Automobiltechnische Zeitschrift*, Bd. 121, Nr. 12, SS. 48–51, 2019, ISSN: 0001-2785.

[31] J. Frochte, *Maschinelles Lernen: Grundlagen und Algorithmen in Python*, 3rd Ed., Serie Plus.Hanser-Fachbuch. München: Hanser, 2021, ISBN: 978-3-446-46355-4.

[32] "torch.nn — PyTorch 1.13 documentation," 2022, Zugriff am 26.12.2022. [Online]. Verfügbar: https://pytorch.org/docs/stable/nn.html#

[33] K. Team, "Keras documentation: Keras layers API," 2022, Zugriff am 26.12.2022. [Online]. Verfügbar: https://keras.io/api/layers/

[34] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, und L. Farhan, "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," *Journal of big data*, Bd. 8, Nr. 1, S. 53, 2021, ISSN: 2196-1115.

[35] NVIDIA Developer, "Jetson Modules, Support, Ecosystem, and Lineup," 2022, Zugriff am 17.12.2022. [Online]. Verfügbar: https://developer.nvidia.com/embedded/jetson-modules

[36] ——, "JetPack SDK 5.0.2," 2022, Zugriff am 17.12.2022. [Online]. Verfügbar: https://developer.nvidia.com/embedded/jetpack-sdk-502

[37] ——, "Jetpack 4.5 Archive," 2022, Zugriff am 17.12.2022. [Online]. Verfügbar: https://developer.nvidia.com/embedded/jetpack-sdk-45-archive

[38] NVIDIA Corporation, "NVIDIA T1000: Full-size Features. Compact Design. Datasheet," 2022, Zugriff am 19.12.2022. [Online]. Verfügbar: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/proviz-print-nvidia-T1000-datasheet-us-nvidia-1670054-r4-web.pdf

[39] K. Divya und G. Girisha, "Autonomous car data collection and analysis," in *Int. Journal of Scientific Research & Engineering Trends*, 2021, Bd. 7, SS. 2056–2059. [Online]. Verfügbar: https://ijsret.com/wp-content/uploads/2021/05/IJSRET_V7_issue3_483.pdf

[40] P. Komarnicki, J. Haubrock, und Z. A. Styczynski, "Autonomes Fahren," in *Elektromobilität und Sektorenkopplung*, P. Komarnicki, J. Haubrock, und Z. A. Styczynski, Hrsgg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, SS. 223–236, ISBN: 978-3-662-62035-9.

[41] J. Janai, F. Güney, A. Behl, und A. Geiger, “Computer Vision for Autonomous Vehicles: Problems, Datasets and State of the Art,” 18.04.2017. [Online]. Verfügbar: http://arxiv.org/pdf/1704.05519v3

[42] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, und W. Shi, “Edge Computing for Autonomous Driving: Opportunities and Challenges,” *Proceedings of the IEEE*, Bd. 107, Nr. 8, SS. 1697–1716, 2019, ISSN: 0018-9219.

[43] T. Bertram, Hrsg., *Automatisiertes Fahren 2019: Von der Fahrerassistenz zum autonomen Fahren : 5. Internationale ATZ-Fachtagung*, Serie Springer eBooks computer science and engineering. Wiesbaden and Heidelberg: Springer Vieweg, 2020, ISBN: 978-3-658-27990-5.

[44] N. Saxena und S. Vibhandik, “Tesla's Competitive Strategies and Emerging Markets Challenges,” in *IUP Journal of Brand Management*, 2021, SS. 57–72.

[45] Tesla, “Tesla Vision Update: Replacing Ultrasonic Sensors with Tesla Vision | Tesla Support Germany,” 2023, Zugriff am 02.01.2023. [Online]. Verfügbar: https://www.tesla.com/de_de/support/transitioning-tesla-vision

[46] H. Brecher, “Teslas Elektroautos erhalten wieder Radar, nach Problemen mit Tesla Vision,” *Notebookcheck*, 08.12.2022, Zugriff am 02.01.2023. [Online]. Verfügbar: https://www.notebookcheck.com/Teslas-Elektroautos-erhalten-wieder-Radar-nach-Problemen-mit-Tesla-Vision.673348.0.html

[47] M. Hilgers, *Elektrik und Mechatronik*, Serie SpringerLink Bücher. Wiesbaden: Springer Vieweg, 2016, ISBN: 978-3-658-12749-7.

[48] N. B. Chetan, J. Gong, H. Zhou, D. Bi, J. Lan, und L. Qie, “An Overview of Recent Progress of Lane Detection for Autonomous Driving,” in *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2020, SS. 341–346, ISBN: 978-1-7281-6057-3.

[49] B. Dorj, S. Hossain, und D.-J. Lee, “Highly Curved Lane Detection Algorithms Based on Kalman Filter,” *Applied Sciences*, Bd. 10, Nr. 7, S. 2372, 2020, DOI: 10.3390/app1007237.

[50] Y. Son, E. S. Lee, und D. Kum, “Robust multi-lane detection and tracking using adaptive threshold and lane classification,” *Machine Vision and Applications*, Bd. 30, Nr. 1, SS. 111–124, 2019, ISSN: 0932-8092.

[51] S. Lee, J. Kim, J. S. Yoon, S. Shin, O. Bailo, N. Kim, T.-H. Lee, H. S. Hong, S.-H. Han, und S. in Kweon, “VPGNet: Vanishing Point Guided Network for Lane and Road Marking Detection and Recognition,” 17.10.2017. [Online]. Verfügbar: http://arxiv.org/pdf/1710.06288v1

[52] J. Zang, W. Zhou, G. Zhang, und Z. Duan, “Traffic Lane Detection using Fully Convolutional Neural Network,” in *2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, 2018, SS. 305–311, ISBN: 978-9-8814-7685-2.

[53] R. Gopalan, T. Hong, M. Shneier, und R. Chellappa, “A Learning Approach Towards Detection and Tracking of Lane Markings,” *IEEE Transactions on Intelligent Transportation Systems*, Bd. 13, Nr. 3, SS. 1088–1098, 2012, ISSN: 1524-9050.

[54] F. Pizzati, M. Allodi, A. Barrera, und F. García, “Lane Detection and Classification using Cascaded CNNs,” 02.07.2019. [Online]. Verfügbar: http://arxiv.org/pdf/1907.01294v2

[55] K. He, G. Gkioxari, P. Dollár, und R. Girshick, “Mask R-CNN,” 20.03.2017. [Online]. Verfügbar: http://arxiv.org/pdf/1703.06870v3

[56] S. Ren, K. He, R. Girshick, und J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” 04.06.2015. [Online]. Verfügbar: http://arxiv.org/pdf/1506.01497v3

[57] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, und A. C. Berg, “SSD: Single Shot MultiBox Detector,” Bd. 9905, SS. 21–37, 2016, DOI: 10.1007/978-3-319-46448-0_2. [Online]. Verfügbar: http://arxiv.org/pdf/1512.02325v5

[58] J. Redmon und A. Farhadi, “YOLOv3: An Incremental Improvement,” 2018, DOI: 10.48550/arXiv.1804.02767.

[59] M. Schmidt, M. Krüger, C. Lienke, M. Oeljeklaus, T. Nattermann, M. Mohamed, F. Hoffmann, und T. Bertram, “Fahrstreifenerkennung mit Deep Learning für automatisierte Fahrfunktionen,” *at - Automatisierungstechnik*, Bd. 67, Nr. 10, SS. 866–878, 2019, ISSN: 0178-2312. [Online]. Verfügbar: https://www.degruyter.com/document/doi/10.1515/auto-2019-0029/html

[60] J. Huang, L. Zhang, Y. Shen, H. Zhang, S. Zhao, und Y. Yang, “DMPR-PS: A Novel Approach for Parking-Slot Detection Using Directional Marking-Point Regression,” in *2019 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, 2019, SS. 212–217, ISBN: 978-1-5386-9552-4.

[61] U. Suddamalla, A. Wong, R. Balaji, B. Lee, und D. K. Limbu, “Camera Based Parking Slot Detection for Autonomous Parking,” in *Computer Vision and Image Processing*, Serie Communications in Computer and Information Science, S. K. Singh, P. Roy, B. Raman, und P. Nagabhushan, Hrsgg. Singapore: Springer Singapore, 2021, Bd. 1378, SS. 58–69, ISBN: 978-981-16-1102-5.

[62] M. Khalid, K. Wang, N. Aslam, Y. Cao, N. Ahmad, und M. K. Khan, “From smart parking towards autonomous valet parking: A survey, challenges and future Works,” *Journal of Network and Computer Applications*, Bd. 175, S. 102935, 2021, ISSN: 1084-8045.

[63] W. Li, H. Cao, J. Liao, J. Xia, L. Cao, und A. Knoll, “Parking Slot Detection on Around-View Images Using DCNN,” *Frontiers in neurorobotics*, Bd. 14, S. 46, 2020, ISSN: 1662-5218.

[64] C. Álvarez Casado und M. Bordallo López, “Real-time face alignment: evaluation methods, training strategies and implementation optimization,” *Journal of Real-Time Image Processing*, Bd. 18, Nr. 6, SS. 2239–2267, 2021, ISSN: 1861-8200.

[65] V.-L. Le, M. Beurton-Aimar, A. Zemmari, und N. Parisey, “Landmarks Detection by Applying Deep Networks,” in *2018 1st International Conference on Multimedia Analysis and Pattern Recognition (MAPR)*. IEEE, 2018, SS. 1–6, ISBN: 978-1-5386-4180-4.

[66] “OpenCV: Canny Edge Detector,” 2023, Zugriff am 14.01.2023. [Online]. Verfügbar: https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html

[67] "OpenCV: Hough Line Transform," 2023, Zugriff am 14.01.2023. [Online]. Verfügbar: https://docs.opencv.org/3.4/d3/de6/tutorial_js_houghlines.html

[68] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, und R. Qu, "A Survey of Deep Learning-based Object Detection," *IEEE Access*, Bd. 7, SS. 128 837–128 868, 2019, ISSN: 2169-3536. [Online]. Verfügbar: http://arxiv.org/pdf/1907.09408v2

[69] S. K. Pal, A. Pramanik, J. Maiti, und P. Mitra, "Deep learning in multi-object detection and tracking: state of the art," *Applied Intelligence*, Bd. 51, Nr. 9, SS. 6400–6429, 2021, ISSN: 0924-669.

[70] K. He, X. Zhang, S. Ren, und J. Sun, "Deep Residual Learning for Image Recognition," 10.12.2015. [Online]. Verfügbar: http://arxiv.org/pdf/1512.03385v1

[71] M. Tan und Q. Le V, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," *International Conference on Machine Learning*. [Online]. Verfügbar: http://arxiv.org/pdf/1905.11946v5

[72] T. Bertram, Hrsg., *Automatisiertes Fahren 2020: Von der Fahrerassistenz zum autonomen Fahren, 6. Internationale ATZ-Fachtagung*, Serie ATZ live. Wiesbaden and Heidelberg: Springer Vieweg, 2021, ISBN: 978-3-658-34752-9. [Online]. Verfügbar: http://www.springer.com/

[73] Y. Dai und S.-G. Lee, "Perception, Planning and Control for Self-Driving System Based on On-board Sensors," *Advances in Mechanical Engineering*, Bd. 12, Nr. 9, S. 168781402095649, 2020, ISSN: 1687-8140.

[74] D. Dang, F. Gao, und Q. Hu, "Motion Planning for Autonomous Vehicles Considering Longitudinal and Lateral Dynamics Coupling," *Applied Sciences*, Bd. 10, Nr. 9, S. 3180, 2020, DOI: 10.3390/app10093180.

[75] W. Schwarting, J. Alonso-Mora, und D. Rus, "Planning and Decision-Making for Autonomous Vehicles," *Annual Review of Control, Robotics, and Autonomous Systems*, Bd. 1, Nr. 1, SS. 187–210, 2018, ISSN: 2573-5144.

[76] J. Kiebler, M. Saljanin, S. Müller, S. Todorovic, J. Neubeck, und A. Wagner, "Novel Approach for Vehicle-Self-Localization," in *22. Internationales Stuttgarter Symposium*, Serie Proceedings, M. Bargende, H.-C. Reuss, und A. Wagner, Hrsgg. Wiesbaden: Springer Fachmedien Wiesbaden, 2022, SS. 75–88, ISBN: 978-3-658-37010-7.

[77] S. Grigorescu, B. Trasnea, T. Cocias, und G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, Bd. 37, Nr. 3, SS. 362–386, 2020, ISSN: 1556-4959.

[78] "OpenCV: Perspective-n-Point (PnP) pose computation," 2023, Zugriff am 02.02.2023. [Online]. Verfügbar: https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html

[79] NVIDIA Developer, "NVIDIA TensorRT," 2023, Zugriff am 04.02.2023. [Online]. Verfügbar: https://developer.nvidia.com/tensorrt

9 Anhang

9.1 Pythoncode zu Version 2 der Fahrfunktion

```
1. import torchvision
2. import torch
3. import torchvision.models as models
4. import torch.nn as nn
5. import torch.cuda
6. from IPython.display import display
7. import traitlets
8. import ipywidgets.widgets as widgets
9. import ipywidgets
10. import traitlets
11. from jetbot import Camera,Robot, bgr8_to_jpeg
12. import cv2
13. import PIL.Image
14. import torchvision.transforms as transforms
15. import numpy as np
16. import time
17. import os
18. import timeit
19. from jetbot import Robot
20. import math
21. import cv2 as cv
22. from torch2trt import TRTModule
23.
24. device = torch.device('cuda')
25. torch.cuda.is_available()
26.
27. float_formatter = "{:.4f}".format
28. np.set_printoptions(formatter={'float_kind':float_formatter})
```

```
1. # from calibration in October 20222
2. ################################################# should not be used #######################
3. camera_matrix_1 = np.array([[164.29619252, 0.,0.],
4.                             [0., 211.9205235, 0.],
5.                             [0., 0., 1.]], dtype = np.float32)
6.
7. dist_coeff_1 = np.array([[-0.28631401],[-1.29287375], [0.02954699], [0.00237475],
[1.72071795]])
8.
9. # from calibration in January 2023
10. ############################################ RECOMMENDED ###############################
11. camera_matrix_2 = np.array([[105.04394428, 0.,0.],        #116.300; 119.1736
12.                             [0., 134.62356942, 0.],
13.                             [0., 0., 1.]], dtype = np.float32)
14.
15. dist_coeff_2 = np.array([[-0.15379067],[-0.14414304], [0.00062517], [-0.00218034],
[0.08020488]])
16.
17.
18. # Set coordinates of markings on the ground (markings on the wall do not work yet -
perception model not properly trained for that use case)
19. # Origin of the world coordinate system is supposed to be placed in the middle of the front
(entrance) line of the 'parking spot'
20. # coordinates of the four inner corners of the markings accordingly; anti-clockwise; starting
on the left bottom of the area
21.
22. ############### Here: distance between markings = 18 cm
############################################################################
23. src_3d = np.array([(-9,0,0),(9,0,0),(9,-37,0),(-9,-37,0)],dtype=np.float32)
```

```
1. #### Image preprocessing #########################################
2.
3. mean = torch.Tensor([0.485, 0.456, 0.406]).cuda().half()
4. std = torch.Tensor([0.229, 0.224, 0.225]).cuda().half()
5.
6. def preprocess(image):
7.     #image = cv2.resize(image, (224,224))     # here not necessary; images from camera
already have right size after correct initialization of the camera
8.     image = PIL.Image.fromarray(image)
9.     image = transforms.functional.to_tensor(image).to(device).half()
10.    image.sub_(mean[:, None, None]).div_(std[:, None, None])
11.    return image[None, ...]
```

```
 1. ###### Class for the ResNet model V211 ################################
 2. class ResNet_simple_with_tanh (nn.Module):
 3.     def __init__(self):
 4.         super(ResNet_simple_with_tanh,self).__init__()
 5.
 6.         self.model = models.resnet50(pretrained = False)
 7.         num_ftrs = self.model.fc.in_features
 8.         self.model.fc = nn.Linear(num_ftrs, output_size, bias = True)
 9.         self.tanh = nn.Tanh()
10.
11.     def forward(self,x):
12.         batch_size, _, _, _ = x.shape
13.
14.         x = self.model(x)
15.         x = self.tanh(x)
16.
17.         return x
```

```
1. ### Set paths of the .pth-file with the weights of the model used for landmark detection
##################################
2. output_size = 9
3. model_weights_path = 'LD_ResNet50-Backbone_v211_Data-07-12-22_bestmodel.pth'
4. model_weights_path_TRT_1 = 'TRT-LD_ResNet50-Backbone_v211_Data-07-12-22_bestmodel.pth'   #
Trained on dataset from 07.12.2022
5. model_weights_path_TRT_2 = 'TRT-LD_ResNet50-Backbone_v211_Data-13-11-22_bestmodel.pth'   #
Trained on dataset from 13.11.2022
```

```
1. model_weights_path_TRT = model_weights_path_TRT_1
```

```
1. #### Initialize TRT model for faster interference on gpu ########################
2. model_trt = TRTModule()
3. model_trt.load_state_dict(torch.load(model_weights_path_TRT))
4. model_trt = model_trt.to(device)
5. model_trt = model_trt.eval().half()
```

```
 1. #### Start camera and initialize widgets for visualisation of the important variables and
parameters ################################################
 2. camera = Camera(height = 224, width = 224, fps = 5)      # height and width important; no
resizing in preprocessing function!!!! 5 FPS, else runtime problems
 3. #camera.start()
 4.
 5. camera_widget = ipywidgets.Image()
 6. image_bev_widget = ipywidgets.Image()
 7.
 8. title_widget_1 = widgets.HTML('<em>Calculated values</em>')
 9. translation_x = ipywidgets.FloatSlider(min=-200, max=200, step=0.01, value =
0.0,description='transl. x (cm)')
10. translation_y = ipywidgets.FloatSlider(min=-200, max=200, step=0.01, value=0.0,
description='transl. y (cm)')
11. translation_z = ipywidgets.FloatSlider(min=-200, max=200, step=0.01, value=0.0,
description='transl. z (cm)')
12. alpha_1 = ipywidgets.FloatSlider(min=-360, max=360, step=0.01, value=0.0, description='alpha
1 (x-Axis)')
13. alpha_2 = ipywidgets.FloatSlider(min=-360, max=360, step=0.01, value=0.0, description='alpha
2 (x-Axis)')
14. beta_1= ipywidgets.FloatSlider(min=-360, max=360, step=0.001, value=0.0, description='beta 1
(y-Axis)')
15. beta_2 = ipywidgets.FloatSlider(min=-360, max=360, step=0.001, value=0.0, description='beta 2
(y-Axis)')
16. gamma_1 = ipywidgets.FloatSlider(min=-360, max=360, step=0.01, value=0.0, description='gamma
1 (z-Axis)')
17. gamma_2 = ipywidgets.FloatSlider(min=-360, max=360, step=0.01, value=0.0, description='gamma
2 (z-Axis)')
18. ref_a = ipywidgets.FloatSlider(min=-360, max=360, step=0.01, value=0.0, description='Ref.
angle (z-Axis)')
19. distance = ipywidgets.FloatSlider(min=-200, max=200, description='distance (cm)')
20.
21. title_widget_2 = widgets.HTML('<em> Control values</em>')
22. speed_gain_slider = ipywidgets.FloatSlider(min=0.0, max=2.0, step=0.01, description='speed
gain')
23. speed_bias_slider = ipywidgets.FloatSlider(min=-0.4, max=0.5, step=0.01, value=0.0,
description='speed bias')
```

```
24. steering_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.0,
description='steering kp far')
25. steering_gain_orient_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.0,
description='steering kp near')
26. steering_dgain_slider = ipywidgets.FloatSlider(min=0.0, max=0.5, step=0.001, value=0.0,
description='steering kd')
27. steering_igain_slider = ipywidgets.FloatSlider(min=0.0, max=0.5, step=0.001, value=0.0,
description='steering ki')
28. steering_bias_slider = ipywidgets.FloatSlider(min=-0.3, max=0.3, step=0.01, value=0.0,
description='steering bias')
29. weight_1_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.5,
description='weight grad')
30. weight_2_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.5,
description='weight transl')
31. st_gain_slider = ipywidgets.FloatSlider(min=0.0, max=100, step=0.1, value=1, description='st
gain v=0')
32.
33. title_widget_3 = widgets.HTML('<em>Output: Motor control</em>')
34. steering_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='steering')
35. speed_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='speed')
36.
37.
display(ipywidgets.HBox([image_bev_widget,ipywidgets.HBox([ipywidgets.VBox([title_widget_1,transl
ation_x, translation_y, translation_z,
38.                                                           alpha_1, alpha_2, beta_1, beta_2,
gamma_1, gamma_2, ref_a, distance]),
39.                                                               ipywidgets.VBox([title_widget_2,
speed_gain_slider, speed_bias_slider,
40.
steering_gain_slider,steering_gain_orient_slider,
41.
steering_dgain_slider, steering_igain_slider,steering_bias_slider,
42.                                                                                weight_1_slider,
weight_2_slider, st_gain_slider]),
43.                                                              ipywidgets.VBox([title_widget_3,
steering_slider, speed_slider])])]))
```

```
 1. ##### Create Bird's eye of view (BEV) - Matrix/Grid in the x-y-Plane of the world coordinate
system #####################################
 2. ### Area of 500 x 500 Pixels (Edge lenght of the quadratic pixels = 1 cm)
 3. ## Origin of the coordinate system (37,250)              # 37 = length of the markings; 250
= center of the grid in x-direction
 4.
 5. x_origin = 250
 6. y_origin = 37
 7.
 8. bound_1 = 60 + y_origin                 # boundary for differentiation between far and middle
field
 9. bound_2 = 20 + y_origin                  # boundary for differentiation between middle and
near field
10.
11. border_line_1 = [(0,bound_1),(500,bound_1)]
12. border_line_2 = [(0,bound_2),(500,bound_2)]
13.
14. ######################################################################
15. image_background = np.zeros((500, 500, 3), np.float32)
16. # origin of the world coordinate system (also center of the front (entrance) line)
17. image_background = cv.circle(image_background,(x_origin,y_origin),2,(255,0,0),-1)
18. #### Corresponding lines of the target area
19. image_background =
cv.line(image_background,(int(src_3d[0,0]+x_origin),int(src_3d[0,1]+y_origin)),
20.           (int(src_3d[3,0]+x_origin),int(src_3d[3,1]+y_origin)),(0,0,255),2)
21. image_background =
cv.line(image_background,(int(src_3d[1,0]+x_origin),int(src_3d[1,1]+y_origin)),
22.            (int(src_3d[2,0]+x_origin),int(src_3d[2,1]+y_origin)),(0,0,255),2)
23. image_background =
cv.line(image_background,(int(src_3d[1,0]+x_origin),int(src_3d[1,1]+y_origin)),
24.            (int(src_3d[2,0]+x_origin),int(src_3d[2,1]+y_origin)),(0,0,255),2)
25. image_background = cv.line(image_background,border_line_1[0],
26.         border_line_1[1],(255,255,255),1)
27. image_background = cv.line(image_background,border_line_2[0],
28.         border_line_2[1],(255,255,255),1)
```

```
29. image_background = cv.putText(image_background, 'Y1 = %s' %str(bound_1), (50+10,bound_1+15), 
fontFace = 0,
30.                                    fontScale = 0.4, color = (255,255,255), thickness = 1)
31. image_background = cv.putText(image_background, 'Y2 = %s' %str(bound_2), (50+10,bound_2+15), 
fontFace = 0,
32.                                    fontScale = 0.4, color = (255,255,255), thickness = 1)
33.
34. image_bev_widget.value = bgr8_to_jpeg(image_background[0:400,50:450])
```

```
 1. ############ Function for trajectory planning in the middle field 
#################################################################
 2. # Two points must be provided: positon of the camera and end-point (along center line with 
certain distance to the markings; f'(y)= 0)
 3.
 4. def grad_polynom(x1,y1,x2,y2):
 5.     # f(y) = ay^2 + by +c
 6.     # f'(y) = 2a y + b
 7.     M = np.array([[2*y1, 1,0],
 8.                  [(y1)**2, y1,1],
 9.                  [(y2)**2,y2,1]], dtype=np.float32)
10.     b = np.array([0,x1,x2],dtype=np.float32)
11.
12.     coeffs_poly = np.dot(np.linalg.inv(M), b)
13.     #coeffs_poly = np.linalg.solve(M, b)
14.
15.     return coeffs_poly
```

```
 1. ### Control algorithm for the far field (inputs: reference angle according the position of 
the camera, angle of orientation of the camera, factor if forward or backward driving)
 2. def control_farfield (angle, phi, factor_sp, factor_st):
 3.     global translation_vector, sensitivity, om_last, om_sum
 4.
 5.     speed = 1 * factor_sp           # speed is constant
 6.     speed_slider.value = speed * speed_gain_slider.value + speed_bias_slider.value
 7.
 8.     # Steering according to angle difference - camera should point at the origin of the world 
coordinate system
 9.     delta_omega = angle - phi
10.
11.     # PID-Controller
12.     if abs(translation_vector[0]) > sensitivity[1]:
13.         om_sum += delta_omega
14.         pid_x = delta_omega * steering_gain_slider.value + (delta_omega - om_last) * 
steering_dgain_slider.value + om_sum * steering_igain_slider.value
15.         om_last = delta_omega
16.     else:
17.         om_sum += delta_omega
18.         pid_x = delta_omega * steering_gain_orient_slider.value + (delta_omega - om_last) * 
steering_dgain_slider.value + om_sum * steering_igain_slider.value
19.         om_last = delta_omega
20.
21.     steering_slider.value = factor_st *(pid_x + steering_bias_slider.value)
22.
23.     # Control commands to motors
24.     robot.left_motor.value = max(min(speed_slider.value + steering_slider.value, 1.0), -1.0)
25.     robot.right_motor.value = max(min(speed_slider.value - steering_slider.value, 1.0), -1.0)
```

```
 1. #### Control algorithm for the middle field
 2. # Inputs: reference angle, x according the trajectory for given y-value; position of camera 
in x-direction; derivative of the trajectory for given y-value; angle of orientation
 3. # factor if forwars or backward
 4. def control_middlefield (ref_angle, x, camera_x, grad_x, phi, factor_sp, factor_st):
 5.     global weight_g, weight_t, translation_vector, t_last, g_last, t_sum, g_sum, sensitivity
 6.
 7.     speed = 1 * factor_sp        # constant speed
 8.     speed_slider.value = speed * speed_gain_slider.value + speed_bias_slider.value
 9.
10.     weight_g = weight_1_slider.value
11.     weight_t = 1 - weight_g
12.     weight_2_slider.value = weight_t
13.
14.     delta_g = abs(grad_x) - abs(math.tan(phi))
15.     #delta_t = abs(translation_vector[0]/sensitivity[0])
```

```
16.     delta_t = (x - camera_x)/max(abs(translation_vector[0]), 1)      # evtl ohne Normierung
!?
17.
18.     if abs(translation_vector[0]) > sensitivity[1]:
19.         t_sum += delta_t
20.         pid_t = delta_t * steering_gain_slider.value + (delta_t - t_last) *
steering_dgain_slider.value+ t_sum * steering_igain_slider.value
21.         t_last = delta_t
22.
23.         g_sum += delta_g
24.         pid_g = delta_g * steering_gain_slider.value + (delta_g - g_last) *
steering_dgain_slider.value + g_sum*steering_igain_slider.value
25.         g_last = delta_g
26.
27.     else:
28.         t_sum += delta_t
29.         pid_t = delta_t * steering_gain_orient_slider.value + (delta_t - t_last) *
steering_dgain_slider.value + t_sum*steering_igain_slider.value
30.         t_last = delta_t
31.
32.         g_sum += delta_g
33.         pid_g = delta_g * steering_gain_orient_slider.value + (delta_g - g_last) *
steering_dgain_slider.value + g_sum*steering_igain_slider.value
34.         g_last = delta_g
35.
36.     # Control commands to motors
37.     if ref_angle > 0:
38.         steering_slider.value = factor_st*((weight_t * pid_t) + (weight_g * pid_g) +
steering_bias_slider.value)
39.         #robot.left_motor.value = max(min(speed_slider.value + steering_slider.value, 1.0), -
1.0)
40.         #robot.right_motor.value = max(min(speed_slider.value - steering_slider.value, 1.0),
-1.0)
41.     else:
42.         steering_slider.value = factor_st*((weight_t * pid_t) - (weight_g * pid_g) +
steering_bias_slider.value)
43.         #robot.left_motor.value = max(min(speed_slider.value - steering_slider.value, 1.0), -
1.0)
44.         #robot.right_motor.value = max(min(speed_slider.value + steering_slider.value, 1.0),
-1.0)
45.
46.     robot.left_motor.value = max(min(speed_slider.value + steering_slider.value, 1.0), -1.0)
47.     robot.right_motor.value = max(min(speed_slider.value - steering_slider.value, 1.0), -1.0)
```

```
 1. ############# Function for initialization of all variables
##################################################
 2. def initialization (th_obj, frames, th_om, th_tr):
 3.     global height, width, threshold, robot,translation_vector, rotation_vector, length,
theta, psi, phi
 4.     global delta_g, g_last, delta_t, t_last, delta_omega, om_last, om_sum, t_sum, g_sum,
sensitivity
 5.     global count_frames, count_frames2, count_frames3, min_frames, counts
 6.     global weight_g, weight_t, x0,y0,x0_sum, y0_sum, x1, y1, x1_sum, y1_sum
 7.     global check_firststop, check_secondstop, check_back1, check_back2, check_search,
image_traj
 8.     global factor_speed, factor_steering, border_1, bound_1, border_2, bound_2
 9.     global thres_omega, thres_trans, release_forward, check_reset
10.
11.     # Initialization step for all the variables
12.
13.     height = 224     # image height
14.     width = 224       # image width
15.     threshold = th_obj  # threshold for object probability in model output
16.     robot = Robot()
17.
18.     translation_vector = np.zeros((3,1))      # initialization of the translation vector
19.     rotation_vector = np.zeros((3,1))      # initialization of the rotation vector
20.     length = 10                        # length of the arrow of camera orientation in birds eye
view
21.     theta = np.zeros((1,2))   # initialization of the euler rotation angle around x-Axis
22.     psi = np.zeros((1,2))         # initialization of the euler rotation angle around y-Axis
23.     phi = np.zeros((1,2))   # initialization of the euler rotation angle around z-Axis
```

```
24.
25.     delta_g = 0.0     # initialization of the control value for the gradient
26.     g_last = 0.0   # initialization of the derivative value for the gradient
27.     delta_t = 0.0 #initialization of the lateral control value
28.     t_last = 0.0    # initizialization of the derivative value for the lateral control value
29.     delta_omega = 0.0
30.     om_last = 0
31.     om_sum = 0
32.     t_sum = 0
33.     g_sum = 0
34.     sensitivity = [250,10]  # Declare the sensitivity values; first one is used for
normalization when translation in x-dir is greater than second value
35.                             # Else the second value is used for normalization of delta_t
36.     count_frames = 0
37.     count_frames2 = 0
38.     count_frames3 = 0
39.     min_frames = frames
40.     counts = 0
41.
42.     weight_g = 0.5       # Weighing of the controller outputs; corresponding to the
controller output regarding the gradient
43.     weight_t = 1 - weight_g     # Corresponding to the controller output regarding the
translation in x-Direction
44.
45.     x0 = 0
46.     y0 = 0
47.     x0_sum = 0
48.     y0_sum = 0
49.     x1 = 0
50.     y1 = 0
51.     x1_sum = 0
52.     y1_sum = 0
53.
54.     check_firststop = 0
55.     check_secondstop = 0
56.     check_back1 = 0
57.     check_back2 = 0
58.     check_search = 0
59.     image_traj = 0
60.     factor_speed = 0
61.     factor_steering = 0
62.
63.     border_1 = bound_1
64.     border_2 = bound_2
65.
66.     thres_omega = th_om
67.     thres_trans = th_tr
68.     release_forward = False
69.     check_reset = 0
```

```
1. initialization(0.5,3,20,20)      # threshold object probability, minimial frame count,
threshold final orientation in deg, threshold final lateral offset in cm
```

```
  1. # manouveuring function; gets called on every camera frame
  2. def execute(change):
  3.     global count_frames, count_frames2, count_frames3, min_frames,x0,x0_sum,y0,y0_sum,
x1,x1_sum,y1,y1_sum, om_last, check_search,image_traj, thres_trans, thres_omega, release_forward
  4.     global weight_g, weight_t, sensitivity,check_firststop, check_secondstop,
check_back1,check_back2,coeff,bound_1,bound_2, src_3d,dst_2d,camera_matrix_2,dist_coeff_2, theta
  5.     global psi,phi, image_bev,image_background, x_origin, y_origin, t_last,g_last,
translation_vector, border_1, border_2, counts
  6.     global om_sum, t_sum, g_sum, factor_speed, check_reset, factor_steering
  7.
  8.     # if all the conditions are fulfilled, the robot can drive straight into the parking
slot for a specific number of counted frames
  9.     # here could alse be considered the outputs of a collision avoidance model, etc.
 10.     if release_forward == True and counts <= 30 and speed_gain_slider.value > 0:
 11.         if counts == 30:
 12.             speed_gain_slider.value = 0.0
 13.             robot.set_motors(0.0,0.0)
 14.             #counts = 0
 15.         else:
```

```
16.             robot.set_motors(0.215,0.185)    #0.202, 0.198
17.             #time.sleep()
18.             #robot.stop()
19.             counts += 1
20.     elif release_forward == False and speed_gain_slider.value > 0:
21.         counts = 0
22.
23.         # starting the processing of the images obtained in each frame
24.
25.         image = change['new']
26.
27.         # model for landmark detection --> ([object probability,x1,y1,x2,y2,x3,y3,x4,y4])
28.         output = model_trt(preprocess(image)).detach().float().cpu().numpy().flatten()
29.
30.         object_prop = output[0]       # Object probability; should be near 0 or 1
31.
32.         if output[0] > threshold:         # if object probability is greater than certain
threshold to suppress false detections
33.             check_search = 0
34.             count_frames3 = 0
35.             # the coordinates of the four landmarks in the image with the origin of the
coordinate system in the middle of the image
36.             dst_2d = np.array([(output[1]*(width/2),output[2]*(height/2)),
37.                                (output[3]*(width/2),output[4]*(height/2)),
38.                                (output[5]*(width/2),output[6]*(height/2)),
39.                                (output[7]*(width/2),output[8]*(height/2))],
40.                               dtype=np.float32)
41.
42.             # Derive the rotation and translation vector from solving eight equations with
the four point correspondencies
43.             success, rotation_vector, translation_vector =
cv.solvePnP(src_3d,dst_2d,camera_matrix_2,dist_coeff_2, flags=6)
44.
45.             if success == True:
46.
47.                 R,jacobian = cv.Rodrigues(rotation_vector)
48.
49.                 R = np.linalg.inv(R)
50.                 translation_vector = np.dot(-R,translation_vector)
51.
52.                 # Derive euler angles from the Rodrigues rotation vector
53.                 if abs(R[2,0]) != 1:
54.                     #print('First requirement fulfilled')
55.                     theta[0,0] = -math.asin(R[2,0])
56.                     theta[0,1] = np.pi - theta[0,0]
57.                     psi[0,0] =
math.atan2((R[2,1]/math.cos(theta[0,0])),(R[2,2]/math.cos(theta[0,0])))
58.                     psi[0,1] =
math.atan2((R[2,1]/math.cos(theta[0,1])),(R[2,2]/math.cos(theta[0,1])))
59.                     phi[0,0] =
math.atan2((R[1,0]/math.cos(theta[0,0])),(R[0,0]/math.cos(theta[0,0])))
60.                     phi[0,1] =
math.atan2((R[1,0]/math.cos(theta[0,1])),(R[0,0]/math.cos(theta[0,1])))
61.                 else:
62.                     #print('Second requirement fulfilled')
63.                     phi[0,0] = 0
64.                     if R[2,0] == -1:
65.                         theta [0,0] = np.pi / 2
66.                         psi[0,0] = phi[0,0] + math.atan2(R[0,1],R[0,2])
67.                     else:
68.                         theta[0,0] = - np.pi/2
69.                         psi [0,0] = - phi[0,0] + math.atan2(-R[0,1],-R[0,2])
70.
71.                 # Position of the camera in the world coordinate system - x-y-plane
72.                 camera_x = x_origin + translation_vector[0]
73.                 camera_y = y_origin + translation_vector[1]
74.
75.                 # Distance in the world x-y-plane (Pythagoras)
76.                 #dist = math.cos((np.pi/2)-abs(psi[0,0]))*translation_vector[2]
77.                 dist = math.sqrt(translation_vector[0]**2 + translation_vector[1]**2)
78.
```

```
79.                 # Reference angle for control algorithm
80.                 ref_angle = - math.atan2(translation_vector[0],translation_vector[1])
81.
82.                 image_bev = image_background.copy()
83.
84.                 if check_firststop == 0:            # Define position of camera/robot in
world coordinate system after first detection of target object
85.                     robot.stop()
86.
87.                     om_sum = 0
88.                     t_sum = 0
89.                     g_sum = 0
90.                     # Average position of the camera over certain number of frames to
suppress outliers
91.                     if count_frames == min_frames:
92.                         x0 = x0_sum / min_frames
93.                         y0 = y0_sum / min_frames
94.                         check_firststop = 1
95.                         count_frames = 0
96.                         x0_sum = 0
97.                         y0_sum = 0
98.                     else:
99.                         x0_sum = x0_sum + camera_x
100.                        y0_sum = y0_sum + camera_y
101.                        count_frames += 1
102.
103.                elif check_firststop == 1:      # After first position is defined - further
distinction for choice of the right strategy
104.
105.                    if y0 > bound_1:        # Case 01: Vehicle in far field
106.
107.                        border_1 = bound_1
108.                        border_2 = bound_2
109.                        check_back1 = 1         ########## NEU
110.
111.                        if (translation_vector[1]+y_origin) > border_1:
112.                            # roboter drives to first border with camera targeted to origin
of world coordinate system
113.                            factor_speed = 1      # forward
114.                            factor_steering = 1
115.                            control_farfield(ref_angle, phi[0,0], factor_speed,
factor_steering)  # Control Algorithm for the far field
116.
117.                            t_sum = 0
118.                            g_sum = 0
119.
120.                        elif (translation_vector[1]+y_origin) < border_1 and
(translation_vector[1]+y_origin) > border_2 :
121.                            # robot stops when entering the middle field and calculates
trajectory if it is the first time entering the area - check_secondstop = 0; else = 1
122.                            count_frames2 = 0
123.
124.                            if check_secondstop == 0:
125.                                robot.stop()
126.
127.                                om_sum = 0
128.                                if count_frames == min_frames:
129.
130.                                    x1 = x1_sum / min_frames
131.                                    y1 = y1_sum / min_frames
132.
133.                                    coeff = grad_polynom(x_origin,bound_2,x1,y1) #
Koeffizienten der Poylnomfunktion
134.                                    y_values = np.arange(bound_2,y1,5)
135.                                    image_traj = image_background.copy()
136.
137.                                    for y_traj in y_values:
138.                                        x_traj = coeff[0]*(y_traj)**2 + coeff[1]*y_traj +
coeff[2]
139.
cv.circle(image_traj,(int(x_traj),int(y_traj)),1,(0,255,0),-1)
```

```
140.
141.                                         image_bev = image_traj.copy()
142.                                         check_secondstop = 1
143.                                         count_frames = 0
144.                                         x1_sum = 0
145.                                         y1_sum = 0
146.                                     else:
147.                                         x1_sum = x1_sum + camera_x
148.                                         y1_sum = y1_sum + camera_y
149.                                         count_frames += 1
150.
151.                                 elif check_secondstop == 1:    # robot already calculated
trajectory and now has to follow it
152.                                     image_bev = image_traj.copy()
153.                                     grad_x = 2*coeff[0]*camera_y + coeff[1]
154.                                     x = coeff[0]*((camera_y)**2) + coeff[1]*camera_y + coeff[2]
155.
156.                                     factor_speed = 1 # forward
157.                                     factor_steering = 1
158.                                     control_middlefield(ref_angle,x, camera_x, grad_x,
phi[0,0],factor_speed, factor_steering) # Control Algorithm for middle field
159.                                 else: pass
160.
161.                             elif (translation_vector[1]+y_origin) < border_2:
162.                                 # robot stops when entering the near field and decides if it is
already positioned correctly within given tolerances
163.                                 grad_x = 0
164.                                 x = x_origin
165.                                 robot.stop()
166.
167.                                 om_sum = 0
168.                                 t_sum = 0
169.                                 g_sum = 0
170.
171.                                 if count_frames2 == min_frames:
172.                                     if abs(phi[0,0]*(180/np.pi)) < thres_omega and
abs(translation_vector[0]) < thres_trans:
173.                                         # corrective manoeuvers necessary
174.
175.                                         check_back2 = 1
176.                                         release_forward = True
177.                                         robot.set_motors(0,0)
178.
179.                                     else:
180.                                         check_back2 = 0
181.                                         check_firststop = 0  # overall position determination in
world coordinate system for case distinction will be pushed by setting to zeros
182.                                         check_secondstop = 0
183.                                         x0_sum = 0
184.                                         y0_sum = 0
185.                                         x1_sum = 0
186.                                         y1_sum = 0
187.                                         release_forward = False
188.                                         count_frames = 0
189.                                 else:
190.                                     x = x_origin
191.                                     grad_x = 0
192.                                     factor_speed = 0
193.                                     factor_steering = st_gain_slider.value
194.                                     control_middlefield (ref_angle, x, camera_x, grad_x,
phi[0,0], factor_speed, factor_steering)
195.                                     time.sleep(0.01)
196.                                     robot.stop()
197.                                     count_frames2 +=1
198.                             else: pass
199.
200.                         elif y0 > bound_2 and y0 < bound_1:      # Case 02: Vehicle in middle
field
201.                             if (translation_vector[1]+y_origin) > border_1 :
202.                                 factor_speed = 1                # forward
203.                                 factor_steering = 1
```

```
204.                             control_farfield(ref_angle, phi[0,0], factor_speed,
factor_steering) # Control Algorithm for far field
205.
206.                             check_back1 = 1
207.                             if check_reset == 0:
208.                                 om_sum = 0
209.                                 check_reset = 1
210.                             else: pass
211.                             check_secondstop =0
212.                             border_1 = bound_1
213.                             border_2 = bound_2
214.
215.                             t_sum = 0
216.                             g_sum = 0
217.
218.                         elif (translation_vector[1]+y_origin) < border_1 and
(translation_vector[1]+y_origin) > border_2 :
219.                             count_frames2 = 0
220.
221.
222.                             if check_back1 == 0:
223.                                 border_1 = bound_1 + 5      # hysteresis
224.                                 factor_speed = -1             # backward
225.                                 factor_steering = 1
226.                                 control_farfield(ref_angle, phi[0,0], factor_speed,
factor_steering) # Control Algorithm   for middle field
227.                                 check_reset = 0
228.                             else:
229.                                 border_1 = bound_1
230.                                 border_2 = bound_2
231.
232.                                 if check_secondstop == 0:
233.                                     robot.stop()
234.
235.                                     om_sum = 0
236.                                     if count_frames == min_frames:
237.                                         x1 = x1_sum / min_frames
238.                                         y1 = y1_sum / min_frames
239.                                         coeff = grad_polynom(x_origin,bound_2,x1,y1)
240.                                         y_values = np.arange(bound_2,y1,5)
241.                                         image_traj = image_background.copy()
242.
243.                                         for y_traj in y_values:
244.                                             x_traj = coeff[0]*(y_traj)**2 + coeff[1]*y_traj
+ coeff[2]
245.
cv.circle(image_traj,(int(x_traj),int(y_traj)),1,(0,255,0),-1)
246.
247.                                         image_bev = image_traj.copy()
248.                                         check_secondstop = 1
249.                                         count_frames = 0
250.                                         x1_sum = 0
251.                                         y1_sum = 0
252.                                     else:
253.                                         x1_sum = x1_sum + camera_x
254.                                         y1_sum = y1_sum + camera_y
255.                                         count_frames += 1
256.
257.                                 elif check_secondstop == 1:
258.                                     image_bev = image_traj.copy()
259.                                     grad_x = 2*coeff[0]*camera_y + coeff[1]
260.                                     x = coeff[0]*((camera_y)**2) + coeff[1]*camera_y +
coeff[2]
261.
262.                                     factor_speed = 1   # forward
263.                                     factor_steering = 1
264.                                     control_middlefield(ref_angle,x, camera_x, grad_x,
phi[0,0],factor_speed, factor_steering) # Control algorithm for middle field
265.                                 else: pass
266.
267.                         elif (translation_vector[1]+y_origin) < border_2:
```

```
268.                            grad_x = 0
269.                            x = x_origin
270.
271.                            om_sum = 0
272.                            t_sum = 0
273.                            g_sum = 0
274.
275.                            if count_frames2 == min_frames:
276.                                robot.stop()
277.                                if abs(phi[0,0]*(180/np.pi)) < thres_omega and
abs(translation_vector[0]) < thres_trans:
278.                                    check_back2 = 1
279.                                    release_forward = True
280.                                    robot.set_motors(0,0)
281.
282.                                else:
283.                                    # Corrective maneouvering
284.                                    check_back2 = 0
285.                                    release_forward = False
286.                                    check_firststop = 0
287.                                    check_secondstop = 0
288.                                    x0_sum = 0
289.                                    y0_sum = 0
290.                                    x1_sum = 0
291.                                    y1_sum = 0
292.                            else:
293.                                count_frames2 += 1
294.
295.                                x = x_origin
296.                                grad_x = 0
297.
298.                                factor_speed = 0
299.                                factor_steering = st_gain_slider.value
300.                                control_middlefield (ref_angle, x, camera_x, grad_x,
phi[0,0], factor_speed, factor_steering)
301.                                time.sleep(0.01)
302.                                robot.stop()
303.
304.                        else: pass
305.
306.                    elif y0 < bound_2:          # Case 03: Vehicle in near field
307.
308.                        if (translation_vector[1]+y_origin) > border_1:
309.
310.                            factor_speed = 1   # forward
311.                            factor_steering = 1
312.                            control_farfield(ref_angle, phi[0,0], factor_speed,
factor_steering)
313.
314.                            if check_reset == 0:
315.                                om_sum = 0
316.                                check_reset = 1
317.                            else:pass
318.
319.                            check_back1 = 1
320.                            check_secondstop = 0
321.                            border_1 = bound_1
322.                            border_2 = bound_2
323.
324.                            t_sum = 0
325.                            g_sum = 0
326.
327.                        if (translation_vector[1]+y_origin) < border_1 and
(translation_vector[1]+y_origin) > border_2:
328.                            count_frames2 = 0
329.
330.
331.                            if check_back1 == 0:
332.                                factor_speed= -1              # backward
333.                                factor_steering = 1
334.                                border_1 = bound_1 + 5     # hysteresis
```

```
335.                                     control_farfield(ref_angle, phi[0,0], factor_speed,
factor_steering)
336.                                     check_reset = 0
337.
338.                                 else:
339.                                     border_1 = bound_1
340.                                     border_2 = bound_2
341.
342.                                     if check_secondstop == 0:
343.                                         robot.stop()
344.                                         om_sum = 0
345.                                         if count_frames == min_frames:
346.
347.                                             x1 = x1_sum / min_frames
348.                                             y1 = y1_sum / min_frames
349.
350.                                             coeff = grad_polynom(x_origin,bound_2,x1,y1)
351.                                             y_values = np.arange(bound_2,y1,5)
352.                                             image_traj = image_background.copy()
353.
354.                                             for y_traj in y_values:
355.                                                 x_traj = coeff[0]*(y_traj)**2 + coeff[1]*y_traj
+ coeff[2]
356.
cv.circle(image_traj,(int(x_traj),int(y_traj)),1,(0,255,0),-1)
357.
358.                                             image_bev = image_traj.copy()
359.                                             check_secondstop = 1
360.                                             count_frames = 0
361.                                             x1_sum = 0
362.                                             y1_sum = 0
363.                                         else:
364.                                             x1_sum = x1_sum + camera_x
365.                                             y1_sum = y1_sum + camera_y
366.                                             count_frames += 1
367.
368.
369.                                     elif check_secondstop == 1:
370.                                         image_bev = image_traj.copy()
371.                                         grad_x = 2*coeff[0]*camera_y + coeff[1]
372.                                         x = coeff[0]*(camera_y)**2 + coeff[1]*camera_y +
coeff[2]
373.
374.                                         factor_speed = 1 # forward
375.                                         factor_steering = 1
376.                                         control_middlefield(ref_angle,x, camera_x, grad_x,
phi[0,0],factor_speed, factor_steering)
377.                                     else: pass
378.
379.                             elif (translation_vector[1]+y_origin) < border_2:
380.                                 grad_x = 0
381.                                 x = x_origin
382.
383.                                 om_sum = 0
384.                                 t_sum = 0
385.                                 g_sum = 0
386.
387.                                 if count_frames2 == min_frames:
388.
389.                                     if abs(phi[0,0]*(180/np.pi)) < thres_omega and
abs(translation_vector[0]) < thres_trans:
390.                                         check_back2 = 1
391.                                         robot.set_motors(0,0)
392.                                         border_2 = bound_2
393.                                         release_forward = True
394.                                     else:
395.                                         check_back2 = 0
396.                                         robot.backward(0.2)
397.                                         check_back1 = 0
398.                                         check_secondstop = 0
399.                                         border_2 = bound_2 + 5
```

```
400.                                     release_forward= False
401.
402.                             else:
403.                                 factor_speed = 0
404.                                 factor_steering = st_gain_slider.value
405.                                 control_middlefield (ref_angle, x, camera_x, grad_x,
phi[0,0], factor_speed, factor_steering)
406.                                 time.sleep(0.01)
407.                                 robot.stop()
408.                                 count_frames2 += 1
409.
410.                         else: pass
411.                     else: pass
412.                 else:pass
413.
414.                 # Update the grid with the located camera position
415.                 image_end = image_bev.copy()
416.                 image_end =
cv.circle(image_end,(int(camera_x),int(camera_y)),3,(255,0,255),-1)
417.                 image_end = cv.arrowedLine(image_end,(int(camera_x),int(camera_y)),
418.                                      (int(camera_x+(length*math.sin(phi[0,0]))),
419.                                       int(camera_y -
(length*math.cos(phi[0,0])))),(255,0,255),1,      tipLength = 0.2)
420.                 image_end = cv.putText(image_end, 'd = %s cm' %(str(round(dist,2))),
(camera_x+10,camera_y+10), fontFace = 0,
421.                                   fontScale = 0.4, color = (255,0,255), thickness = 1)
422.                 image_end = cv.putText(image_end, 'gamma = %s deg'
%(str(round(phi[0,0]*(180/np.pi),2))), (camera_x+10,camera_y+20), fontFace = 0,
423.                                   fontScale = 0.4, color = (255,0,255), thickness = 1)
424.                 image_bev_widget.value = bgr8_to_jpeg(image_end[0:400,50:450])
425.
426.                 #Update the sliders
427.                 translation_x.value = translation_vector[0]
428.                 translation_y.value = translation_vector[1]
429.                 translation_z.value = translation_vector[2]
430.
431.                 alpha_1.value = psi[0,0]*(180/np.pi)
432.                 alpha_2.value = psi[0,1]*(180/np.pi)
433.                 beta_1.value = theta[0,0]*(180/np.pi)
434.                 beta_2.value = theta[0,1]*(180/np.pi)
435.                 gamma_1.value = phi[0,0]*(180/np.pi)
436.                 gamma_2.value = phi[0,1]*(180/np.pi)
437.
438.                 distance.value = dist
439.                 ref_a.value = ref_angle *(180/np.pi)
440.
441.             else: print('Calculation of angles and translation not successful ')
442.
443.         else:                   # Search strategy
444.             if check_search == 0:  # first time starting search strategy
445.                 if count_frames3 == min_frames:
446.
447.                     border_1 = bound_1
448.                     border_2 = bound_2
449.
450.                     check_firststop = 0
451.                     check_secondstop = 0
452.                     check_back1 = 0
453.                     check_back2 = 0
454.                     count_frames2 = 0
455.
456.                     check_search = 1
457.                     count_frames3 = 0
458.
459.                     t_sum = 0
460.                     g_sum = 0
461.                     om_sum = 0
462.
463.                 else:
464.                     count_frames3 += 1
```

```
465.                     # If there are old positions of camera known while loosing track of the
markings within a few frames; robot should turn in the direction of the markings
466.                     if phi[0,0] > 0:
467.                         robot.set_motors(0.0,0.2)
468.                         time.sleep(0.01)
469.                         robot.stop()
470.                     elif phi[0,0] < 0:
471.                         robot.set_motors(0.2,0.0)
472.                         time.sleep(0.01)
473.                         robot.stop()
474.                     else: robot.stop()
475.             else:
476.                 # Search strategy: driving in a circle with expanding radius
477.                 if speed_gain_slider.value > 0:
478.                     speed = speed_gain_slider.value
479.
480.                     robot.set_motors(speed,speed)
481.                     time.sleep(0.01)
482.                     robot.set_motors(speed+0.15,speed-0.05)
483.                 else: robot.stop()
```

```
1. start = time.time()
2.
3. execute({'new': camera.value})
4.
5. end = time.time()
6. print("The time of execution of above program is :",
7.      (end-start) * 10**3, "ms")
```

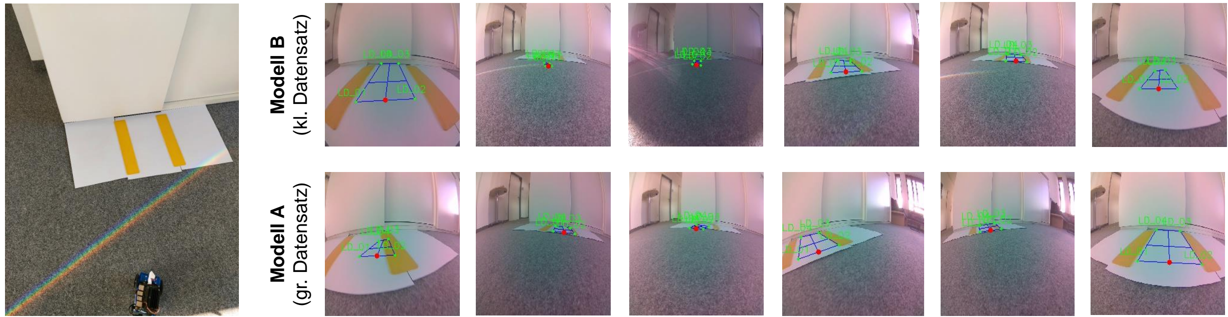
```
1. camera.observe(execute, names='value')
```

9.2 Randbedingungen der Integrationstests

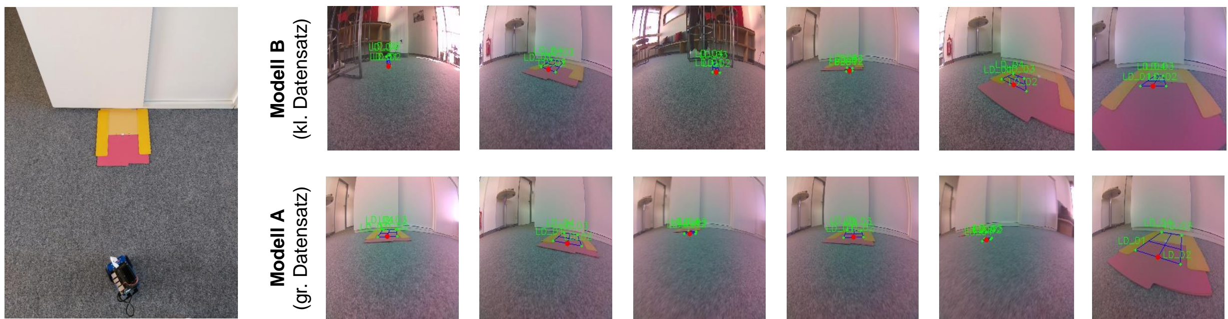
Modell A	ResNet_V211 (07.12.2022)	-
Modell B	ResNet_V211 (13.11.2022)	-
Abstand Markierungen	18	cm
Beleuchtung	Tageslicht	-
Büroumgebung	-	-
Schwellwert Objektwahrscheinlichkeit	0,5	-
Lateraler Versatz (obere Grenze)	20	cm
Verdrehung (obere Grenze)	20	°
Min. Frame count	3	-
K_P (far)	0,05	-
K_P (near)	0,06	-
K_D	0,02	-
K_I	0,01	-
weight_gradient	0,6	-
weight_translation	0,4	-
steering gain (v = 0)	28	-
Radgeschwindigkeit	(0,20 bis 0,25)*u_max	U/min
Bereichsgrenze 1 (fern-mittel; $\triangle$y)	60	cm
Bereichsgrenze 2 (mittel-nah; $\triangle$y)	25	cm

9.3 Ergebnisse der Robustheitsuntersuchung

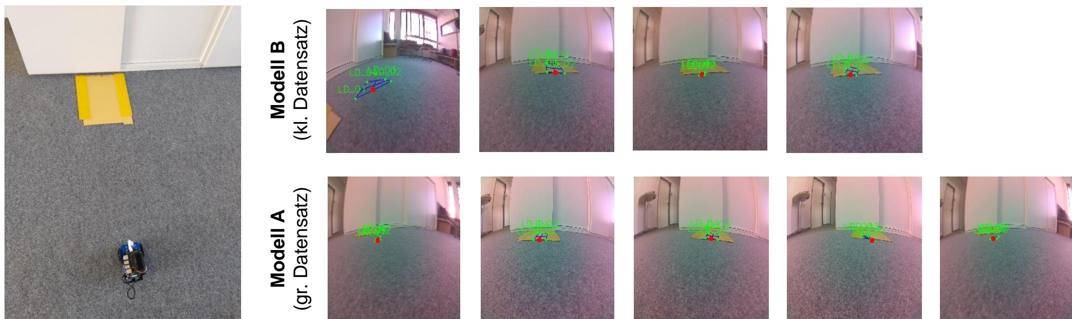
(02) Weißer, homogener Untergrund bei den Markierungen



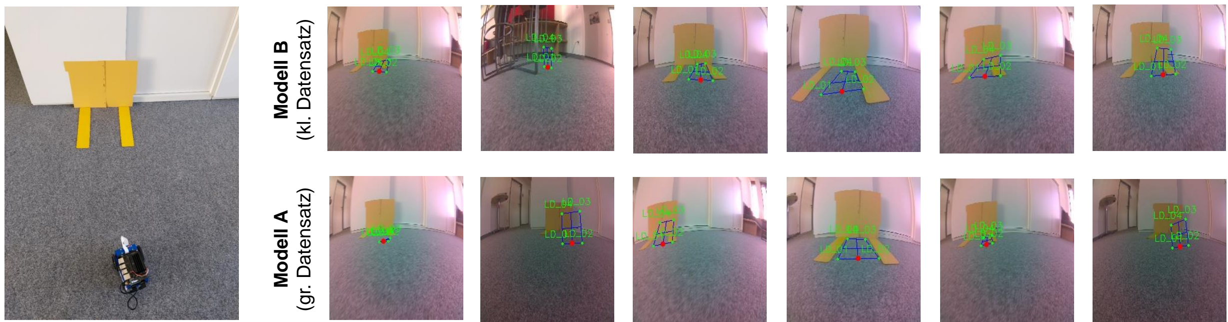
(03) Farbiger und zweigeteilter Untergrund bei den Markierungen



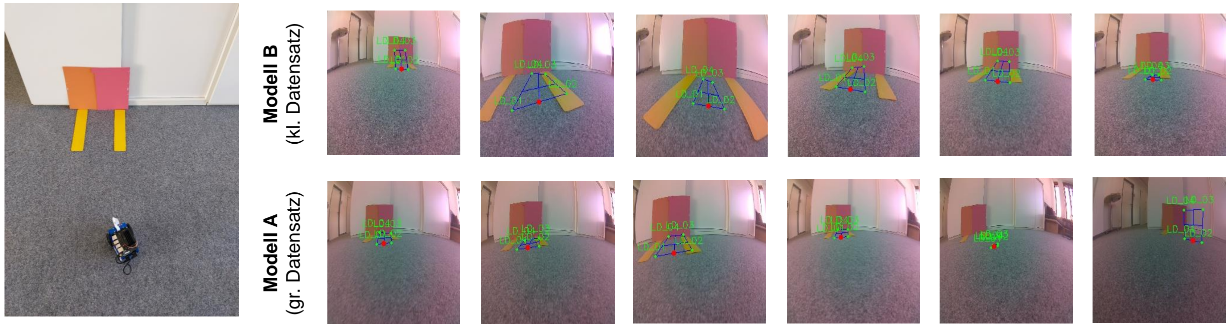
(04) Gelber, homogener Untergrund bei den Markierungen



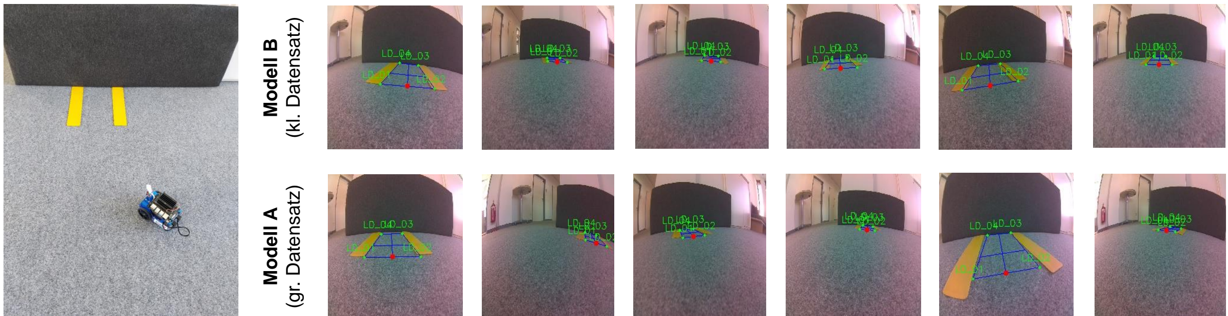
(05) Gelber, homogener Hintergrund bei den Markierungen



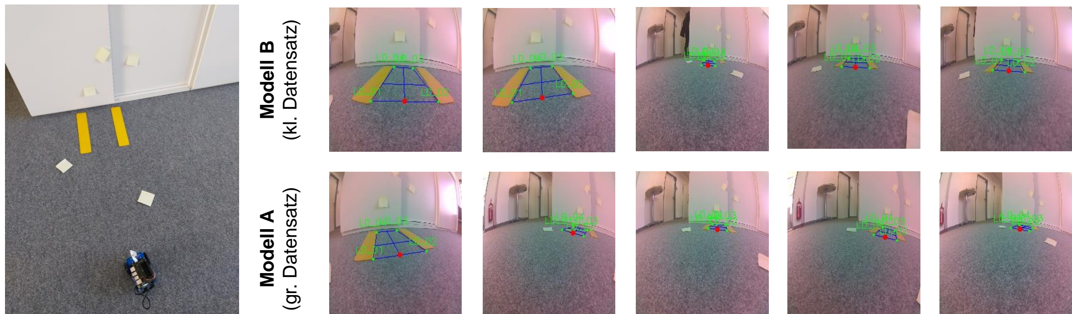
(06) Farbiger und zweigeteilter Hintergrund bei den Markierungen



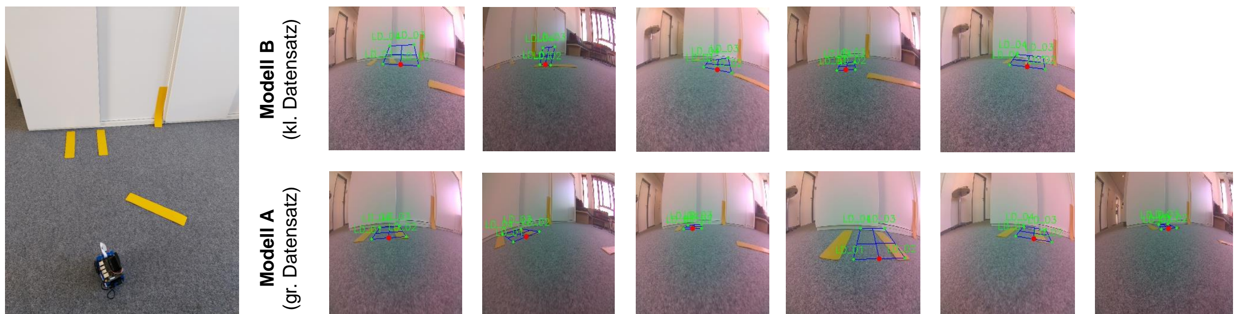
(07) Dunkler, homogener Hintergrund bei den Markierungen



(08) Kleine gelbe und rechteckige Störobjekte im Sichtfeld (z.B. Post-Its)

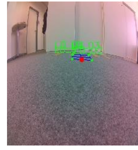
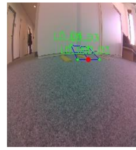
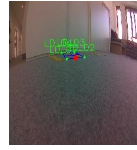


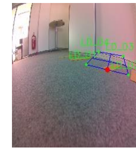
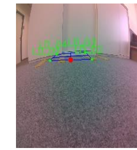
(09) Weitere gleichartige und zufällig positionierte Markierungen im Sichtfeld



(10) Härtetest: Mehr als zwei Markierungen parallel auf Boden nebeneinander

Modell B (kl. Datensatz)

Modell A (gr. Datensatz)