Rettungsgassenbildung mit autonomen Fahrzeugen auf Hardware mit beschränkten Ressourcen

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Jurij Kuzmic

Dortmund

2024

Jurij Kuzmic
Lehrstuhl XI - Algorithm Engineering
Fakultät für Informatik
Technische Universität Dortmund
Otto-Hahn-Str. 14
44227 Dortmund

Tag der mündlichen Prüfung: **27.02.2024**

Dekan **Prof. Dr. Gernot Fink**

Gutachter **Prof. Dr. Günter Rudolph**
(TU Dortmund, Fakultät für Informatik)

Prof. Dr. Sanaz Mostaghim
(Otto-von-Guericke-Universität Magdeburg,
Fakultät für Informatik)

Vorwort

Alle Publikationen dieser Arbeit wurden einem Peer-Review-Verfahren unterzogen, um die Qualität meiner wissenschaftlichen Arbeit durch unabhängige Gutachter aus dem gleichen Fachgebiet zu sichern. Diese Publikationen werden ebenfalls in dieser Forschungsarbeit mehr erläutert und referenziert. Ebenfalls werden die einzelnen Innovationen in den kommenden Kapiteln beschrieben und am Ende dieser Forschungsarbeit nochmal zusammengefasst. Der Eigenanteil der Publikationen 1, 2, 3 und 5 beträgt 95 %. Der Eigenanteil der 4. Veröffentlichung beträgt 90 %.

Die Liste der Publikationen meiner Forschungsarbeit ist wie folgt:

1. Jurij Kuzmic und Günter Rudolph. „Unity 3D Simulator of Autonomous Motorway Traffic Applied to Emergency Corridor Building“. In: *Proceedings of the 5th International Conference on Internet of Things, Big Data and Security*. INSTICC. SciTePress, 2020, S. 197-204. ISBN: 978-989-758-426-8, pp. 197-204.

2. Jurij Kuzmic und Günter Rudolph. „Comparison between Filtered Canny Edge Detector and Convolutional Neural Network for Real Time Lane Detection in a Unity 3D Simulator“. In: *Proceedings of the 6th International Conference on Internet of Things, Big Data and Security (IoTBDS)*. INSTICC. SciTePress, 2021, S. 148-155. ISBN: 978-989-758-504-3, pp. 148-155.

3. Jurij Kuzmic und Günter Rudolph. „Object Detection with TensorFlow on Hardware with Limited Resources for Low-Power IoT Devices“. In: *Proceedings of the 13th International Conference on Neural Computation Theory and Applications (NCTA)*. INSTICC. SciTePress, 2021, S. 302-309. ISBN: 978-989-758-534-0, pp. 302-309.

4. Jurij Kuzmic, Patrick Brinkmann und Günter Rudolph. „Real-time Object Detection with Intel NCS2 on Hardware with Limited Resources for Low-power IoT Devices“. In: *Proceedings of the 7th International Conference on Internet of Things, Big Data and Security (IoTBDS)*. INSTICC. SciTePress, 2022, S. 110-118. ISBN: 978-989-758-564-7, pp 110-118.

5. Jurij Kuzmic und Günter Rudolph. „Real-time Distance Measurement in a 2D Image on Hardware with Limited Resources for Low-power IoT Devices (Radar Control System)“. In: *Proceedings of the 3rd International Conference on Deep Learning Theory and Applications (DeLTA)*. INSTICC. SciTePress, 2022, S. 94-101. ISBN: 978-989-758-584-5, pp. 94-101.

Danksagungen

An dieser Stelle möchte ich mich als Erstes bei meinem Betreuer Prof. Dr. Günter Rudolph für die Möglichkeit der Promotion an der Technischen Universität in Dortmund bedanken. Dieser ließ mich in seiner Arbeitsgruppe Computational Intelligence am Lehrstuhl für Algorithm Engineering arbeiten und gab mir die Freiheit meine eigenen Forschungsinteressen bei meiner Promotion zu verfolgen. Herr Rudolph war immer unterstützend an meiner Seite und hatte immer ein offenes Ohr für Fragen und Ideen, die ich während meiner Promotion hatte. Ebenfalls bin ich Ihm für die Möglichkeiten der Veröffentlichungen dankbar. Durch die Konferenzreisen konnte ich verschiedene Orte auf der Erde besuchen und mich über die Forschungsinteressen mit anderen Promotionsstudenten und Professoren austauschen. Des Weiteren geht ein besonderer Dank an den Arbeitgeber, Technische Universität Dortmund, der mir die Promotion inklusive der Arbeitsplatznutzung und benötigter Hardware ermöglichte. Für das Mentoring bedanke ich mich an dieser Stelle bei Prof. Dr. Falk Howar.

Zusätzlich möchte ich mich bei meinen Arbeitskollegen für die hilfreichen Diskussionen und Forschungsgespräche bedanken. Ein besonderer Dank geht dabei an Fabian Ostermann und Marco Pleines. Dieser Wissensaustausch war immer sehr hilfreich und unterstützend. Für die technische Unterstützung möchte ich mich bei unseren Administratoren Helmut Henning und Peter Svoboda bedanken. Für die Hilfe bei der organisatorischen Arbeit möchte ich ebenfalls Gundel Jankord und Bettina Prenneis danken. Darüber hinaus geht ein besonderer Dank an die Korrekturleser meiner Dissertation. Ebenfalls war die Arbeitsatmosphäre am Lehrstuhl XI und in der Arbeitsgruppe Computational Intelligence sehr entspannt und produktiv. Ein besonderer Dank geht abschließend an meine Familie und Freunde, die mich während meiner Promotion begleitet haben.

Jurij Kuzmic
Dortmund, 13.11.2023

Inhaltsverzeichnis

Vorwort . . . iii
Danksagungen . . . iv

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Aufbau der Arbeit . . . 2

2 Grundlagen **4**
2.1 Einführung . . . 4
2.2 Maschinelles Lernen . . . 4
2.2.1 Künstliche neuronale Netzwerke . . . 5
2.2.2 Faltende neuronale Netzwerke . . . 7
2.2.3 Fehlerrückführung . . . 8
2.3 Computer Vision . . . 10
2.3.1 Spurerkennung . . . 11
2.3.2 Canny-Algorithmus . . . 11
2.3.3 Hough-Transformation . . . 13
2.3.4 Objekterkennung . . . 15
2.3.5 Transformation in die Vogelperspektive . . . 17
2.4 Autonomes Fahren . . . 20
2.4.1 Was ist autonomes Fahren? . . . 20
2.4.2 Lenkverhalten, Spur-, Entfernungs- und Rotationserkennung . . . 21
2.4.3 Kommunikation zwischen autonomen Fahrzeugen . . . 22
2.4.4 Stufen der Automatisierung . . . 23
2.5 Programmiersprachen, Plattformen und Bibliotheken . . . 24
2.5.1 Python . . . 24
2.5.2 C# . . . 24
2.5.3 Unity . . . 25
2.5.4 Unity ML-Agents Toolkit . . . 26
2.5.5 TensorFlow . . . 26
2.5.6 TensorFlow Object Detection API . . . 27
2.5.7 OpenCV . . . 27
2.6 Verwendete Hardware . . . 28
2.6.1 Dell G3 15 3590 Notebook . . . 28
2.6.2 Raspberry Pi . . . 28
2.6.3 Intel Neural Compute Stick 2 . . . 29
2.6.4 Google Colab . . . 29

3 Entwicklung eines Simulators in Unity **30**
3.1 Einführung . . . 30
3.2 Verwandte Arbeiten . . . 31
3.3 Simulation in Unity . . . 32
3.3.1 Umwelt und Objekte . . . 33
3.3.2 Lenkverhalten, Spur-, Entfernungs- und Rotationserkennung . . . 35
3.3.3 Kommunikation zwischen simulierten Fahrzeugen . . . 37
3.4 Algorithmen . . . 41
3.4.1 Bildung der Rettungsgasse: Stockender Verkehr . . . 41
3.4.2 Bildung der Rettungsgasse: Stehender Verkehr . . . 42
3.5 Experimente . . . 44
3.5.1 Bildung der Rettungsgasse: Stockender Verkehr . . . 44
3.5.2 Bildung der Rettungsgasse: Hindernisse auf der Fahrbahn . . . 45
3.5.3 Bildung der Rettungsgasse: Platzen des Vorderreifens . . . 46
3.5.4 Bildung der Rettungsgasse: Stehender Verkehr . . . 47
3.5.5 Auswertung der Ergebnisse . . . 48
3.6 Schlussfolgerungen . . . 49

4 Spurerkennung **51**
4.1 Einführung . . . 51
4.2 Verwandte Arbeiten . . . 52
4.3 Datensätze . . . 52
4.3.1 Automatische Annotation . . . 54
4.4 Spurerkennung mit gefilterten Canny-Algorithmus . . . 56
4.5 Spurerkennung mit faltendem neuronalen Netzwerk . . . 61
4.6 Experimente . . . 63
4.6.1 Laufzeit und Fehler vom gefilterten Canny-Algorithmus . . . 63
4.6.2 Laufzeit und Fehler vom faltenden neuronalen Netzwerk . . . 65
4.6.3 Auswertung der Ergebnisse . . . 67
4.7 Schlussfolgerungen . . . 69

5 Übertragung von Simulation auf Realität **71**
5.1 Einführung . . . 71
5.2 Zusammenbau der Modellfahrzeuge . . . 72
5.2.1 Konstruktion . . . 72
5.2.2 Hardware, Anpassungen und Erweiterungen . . . 73
5.3 Aufbau der Teststrecke . . . 75
5.4 Lenkverhalten, Spur-, Entfernungs- und Rotationserkennung . . . 76
5.5 Kommunikation zwischen Modellfahrzeugen . . . 79
5.6 Experimente . . . 81
5.6.1 Bildung der Rettungsgasse: Stockender Verkehr . . . 81
5.6.2 Bildung der Rettungsgasse: Hindernisse auf der Fahrbahn . . . 82
5.6.3 Bildung der Rettungsgasse: Platzen des Vorderreifens . . . 83
5.6.4 Bildung der Rettungsgasse: Stehender Verkehr . . . 84

5.6.5 Auswertung der Ergebnisse . . . 86
5.7 Schlussfolgerungen . . . 87

6 Objekterkennung auf Hardware mit beschränkten Ressourcen 88
6.1 Einführung . . . 88
6.2 Verwandte Arbeiten . . . 89
6.3 Datensätze . . . 90
6.3.1 Automatische Annotation . . . 93
6.3.2 Manuelle Annotation . . . 94
6.4 Funktionsweise . . . 95
6.4.1 Erkennung der Objekte ohne Hardwareerweiterung . . . 95
6.4.2 Erkennung der Objekte mit Hardwareerweiterung . . . 103
6.5 Experimente . . . 107
6.5.1 Sim-to-Real Übertragung . . . 108
6.5.2 Laufzeit auf Grafikkarte . . . 113
6.5.3 Laufzeit auf beschränkten Ressourcen ohne Hardwareerweiterung . 113
6.5.4 Laufzeit auf beschränkten Ressourcen mit Hardwareerweiterung . 115
6.5.5 Auswertung der Ergebnisse . . . 116
6.6 Schlussfolgerungen . . . 118

7 Kontrollsystem für Radarsensor auf Hardware mit beschränkten Ressourcen 120
7.1 Einführung . . . 120
7.2 Verwandte Arbeiten . . . 121
7.3 Datensätze . . . 122
7.3.1 Automatische Annotation . . . 124
7.3.2 Manuelle Annotation . . . 125
7.4 Funktionsweise . . . 126
7.4.1 Simulation . . . 127
7.4.2 Modellfahrzeuge . . . 129
7.4.3 Echte Fahrzeuge . . . 130
7.5 Experimente . . . 132
7.5.1 Auflösung und Genauigkeit . . . 132
7.5.2 Auflösung und Laufzeit . . . 133
7.5.3 Auswertung der Ergebnisse . . . 134
7.6 Schlussfolgerungen . . . 135

8 Zusammenfassung und Ausblick 137
8.1 Analyse der Innovationen . . . 137
8.2 Mögliche Erweiterungen der Arbeit . . . 139
8.3 Fazit . . . 140

Literaturverzeichnis 142

Sollte dies Kaffee sein, bringen Sie mir bitte Tee. Sollte dies Tee sein, bringen Sie mir bitte Kaffee.

Abraham Lincoln
1809 - 1865

Kapitel 1

Einleitung

1.1 Motivation

Künstliche neuronale Netzwerke finden immer mehr Anwendung bei bestimmten Problemen. Doch in dieser Arbeit stellen sich mehrere Fragen: Müssen ausschließlich die künstlichen neuronalen Netzwerke für bestimmte Probleme verwendet werden oder können die traditionellen Methoden bei den gleichen Problemen vergleichbare Ergebnisse liefern? Wie wirken sich die künstlichen neuronalen Netzwerke auf die Ressourcen im Vergleich zu traditionellen Methoden aus?

Der Schwerpunkt dieser Forschungsarbeit liegt auf der Entwicklung der Algorithmen und Software für stromsparende IoT-Geräte bezogen auf die Problematik der Rettungsgassenbildung auf Autobahnen mit autonomen Fahrzeugen. Auf den Autobahnen kommt es immer wieder zu unvorhersehbaren schwerwiegenden Unfällen. Dabei vergessen die Fahrer der Fahrzeuge teilweise eine Rettungsgasse für die Polizei- und Rettungsfahrzeuge zu bilden und behindern diese beim Erreichen des Unfallorts. Dabei ist für die Unfallbeteiligten jede Minute sehr wichtig. Die autonomen Fahrzeuge hingegen könnten die entwickelten Algorithmen zur Bildung der Rettungsgasse bei einem bestimmten Ereignis ausführen. Die Behebung dieser Problematik ist somit durch die autonomen Fahrzeuge möglich. Dementsprechend ist die Motivation dieser Forschungsarbeit dem Problem bei der Bildung der Rettungsgasse auf Autobahnen für die Polizei- und Rettungsfahrzeuge bei einem Unfall mit autonomen Fahrzeugen entgegenzuwirken. Dazu werden zwei Algorithmen erforscht und anschließend getestet. Die Evaluation der Algorithmen ist durch den in dieser Forschungsarbeit entwickelten Simulator möglich. Anschließend werden diese Algorithmen in der echten Umgebung und nicht nur in der Simulation überprüft. Um dem realen Problem entgegenzuwirken, werden echte autonome Modellfahrzeuge entwickelt und auf einer Teststrecke getestet. Außerdem ist die Motivation das Budget so gering wie möglich zu halten und dabei der Umwelt zur Liebe Strom und Geld zu sparen. Deswegen liegt der Fokus auf der Entwicklung der Software für stromsparende IoT-Geräte.

Dabei entstehen die folgenden Fragestellungen: Welche der entwickelten Algorithmen und Methoden können überhaupt auf der stromsparenden IoT-Hardware ausgeführt werden? Welche Laufzeiten können auf diesen IoT-Geräten erreicht werden? Können ebenfalls Echtzeitanwendungen auf der verwendeten IoT-Hardware implementiert werden? Die in diesem Kapitel aufgekommenen Fragestellungen können durch die verschiedenen Experimente in den nachfolgenden Kapiteln dieser Forschungsarbeit beantwortet werden. Die Thematik der beschränkten Ressourcen ist nicht nur bei den Modellfahrzeugen interessant. Beschränkte Ressourcen findet man heutzutage ebenfalls in Drohnen, smarten Überwachungskameras, Wildkameras und weiteren IoT-Geräten.

1.2 Aufbau der Arbeit

Im einleitenden Kapitel 1 wird auf die Motivation und den Hintergrund dieser Arbeit eingegangen. Kapitel 2 stellt zusätzlich die Grundlagen zu den in dieser Arbeit verwendeten Bereichen. Es werden die Grundlagen zu den Themen wie maschinelles Lernen, Computer Vision und autonomes Fahren erläutert. Diese Grundlagen werden in den weiteren Kapiteln dieser Forschungsarbeit verwendet beziehungsweise teilweise erweitert. Zusätzlich werden die Programmiersprachen, Plattformen, Bibliotheken und die verwendete Hardware dieser Arbeit beschrieben.

Anschließend beschreibt Kapitel 3 die Entwicklung eines Simulators in Unity und die Entwicklung von zwei verschiedenen Algorithmen für die Bildung der Rettungsgasse mit autonomen Fahrzeugen. Der erste Algorithmus bildet die Rettungsgasse bei stockendem Verkehr. Der zweite Algorithmus hingegen bei stehendem Verkehr. Durch die Entwicklung des Simulators können die entwickelten Algorithmen für die Bildung der Rettungsgasse veranschaulicht und überprüft werden. Der Vorteil in der simulierten Umgebung zu arbeiten ist, dass die entwickelten Methoden und Algorithmen, ohne einen praktischen aufwändigen experimentellen Aufbau, in der Simulation überprüft und getestet werden können. Zusätzlich enthält die simulierte Umgebung verschiedene Hilfsmittel, um zum Beispiel ein autonomes Fahrzeug ohne bildliche Auswertung in einem Spurverlauf zu halten. Das ist in der Praxis nicht der Fall. Diese Hilfsmittel stehen in der echten Umgebung nicht zur Verfügung. Somit wird in dem Kapitel 4 eine bildliche Spurerkennung für den in Kapitel 3 entwickelten Simulator entwickelt, um hinterher in Kapitel 5 auf die Modellfahrzeuge in der echten Umgebung umzusteigen und die entwickelten Algorithmen in der Praxis zu überprüfen. Dabei steht die Entwicklung von Software für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte im Vordergrund.

Das Kapitel 6 stellt eine Objekterkennung für Hardware mit beschränkten Ressourcen vor. Dabei werden ebenfalls verschiedene Modelle vorgestellt und der Einsatz einer Hardwareerweiterung für Hardware mit beschränkten Ressourcen untersucht. Dadurch können zum Beispiel die Objekte auf einer Fahrbahn in Echtzeit erkannt und klassifiziert werden. Als eine Erweiterung der Objekterkennung wird in Kapitel 7 eine Entfernungsmessung in einem zweidimensionalen Bild zum erkannten Objekt auf Hardware mit beschränkten Ressourcen vorgestellt. Dieses System ermöglicht eine bildliche Entfernungsmessung zum vorderen erkannten Objekt, durch Pixelmessungen und anschließender Umwandlung auf

die tatsächliche Entfernung, durchzuführen. Dieses kann ebenfalls als ein Kontrollsystem für einen Radarsensor bei autonomen Fahrzeugen verwendet werden. Zum Ende wird auf die einzelnen Innovationen dieser Arbeit eingegangen. Es werden zusätzlich mögliche Erweiterungen und die mögliche zukünftige Arbeit in diesem Bereich vorgestellt. Anschließend wird die vorgestellte Forschungsarbeit nochmal zusammengefasst.

Bei allen Unternehmungen muss vor deren
Beginn eine sorgfältige Vorbereitung stehen.

Marcus Tullius Cicero
106 – 43 BCE

Kapitel 2

Grundlagen

2.1 Einführung

In diesem Kapitel werden die Grundlagen der in dieser Forschungsarbeit verwendeten Thematiken vorgestellt. Die nachfolgenden Abschnitte stellen die Grundlagen zu maschinellem Lernen, inklusive der künstlichen neuronalen Netzwerke, faltenden neuronalen Netzwerke und Fehlerrückführung, vor. Die nachfolgenden Abschnitte geben einen guten Überblick über die Funktionsweise der künstlichen neuronalen Netzwerke. Dabei geht es nicht um die mathematische Tiefe, da die künstlichen neuronalen Netzwerke in dieser Forschungsarbeit nicht in der Forschung, sondern in der Anwendung verwendet werden. Zusätzlich wird auf das Thema der Computer Vision, inklusive dem Verständnis zur Spurerkennung, Canny-Algorithmus, Hough-Transformation und Objekterkennung, eingegangen. Ebenfalls werden die in dieser Forschungsarbeit verwendeten Programmiersprachen, Entwicklungsplattformen und Bibliotheken vorgestellt. Für den im nachfolgenden Kapitel 3 vorgestellten Simulator, wird die Entwicklungsumgebung Unity verwendet. Dazu werden auch einige Grundlagen vorgestellt. Am Schluss des Kapitels wird die in dieser Arbeit verwendete Hardware erläutert.

2.2 Maschinelles Lernen

Beim maschinellen Lernen (engl. Machine Learning) geht es um die Extrahierung von Wissen aus Daten. Dabei handelt es sich um ein Forschungsfeld in der Schnittmenge von Statistik und Informatik [72]. Die Methoden von maschinellem Lernen sind heutzutage nicht mehr wegzudenken, da diese in den letzten Jahren Teil des menschlichen Alltags geworden sind. Diese finden Anwendung in zum Beispiel automatischen Empfehlungen von Filmen, Nahrungsmitteln oder anderen Produkten. Zusätzlich findet maschinelles Lernen in der Erkennung und Zuordnung von Personen in Fotos Anwendung. Moderne Webseiten

von zum Beispiel Facebook, Amazon oder Netflix verwenden ebenfalls verschiedene Lernmodelle. Bei der Entwicklung intelligenter Systeme wurden oft von Hand kodierte Regeln in Form von Entscheidungen verwendet, um Daten zu verarbeiten oder Benutzereingaben anzupassen. Doch das Verwenden von Hand erstellter Regeln bringt einige Nachteile mit sich. Die Entscheidungslogik ist nur für ein bestimmtes Fachgebiet und eine Aufgabe konzipiert. Somit können kleine Veränderungen dazu führen, dass das gesamte System neu implementiert werden muss. Dagegen können die maschinellen Lernalgorithmen den Entscheidungsprozess durch Verallgemeinerung aus bekannten Beispielen automatisieren. An dieser Stelle spricht man vom überwachten Lernen (engl. Supervised Learning). Dabei stellt der Entwickler dem Algorithmus Paare von Eingabewerten und erwünschten Ausgabewerten zur Verfügung. Anhand dieser Werte findet der Algorithmus mit Hilfe von Fehlerrückführungsalgorithmen selbständig heraus, wie sich die gewünschte Ausgabe erstellen lässt. Dabei ist die Idee mit Hilfe von Trainingsdaten ein Modell zu entwickeln, um genaue Vorhersagen für neue unbekannte Daten zu treffen. Diese Methode ist eine der am häufigsten eingesetzte und erfolgreichste Art von maschinellem Lernen [72]. Im Wesentlichen gibt es zwei Arten von Aufgaben für das überwachte Lernen. Dabei handelt es sich um Klassifikation und Regression. Das Ziel bei der Klassifikation ist es, eine Klassenbezeichnung vorherzusagen. Dabei wird aus einer Auswahl vorgegebener Klassen entschieden. So kann zum Beispiel einem Objekt in einem Bild die Klasse Hund, Katze oder Maus zugeordnet werden. Bei der Regression spricht man von einer Vorhersage einer kontinuierlichen Größe, einer reellen Zahl. Zum Beispiel kann das jährliche Einkommen einer Person aus Bildungsgrad, Alter und Wohnort vorhergesagt werden [72].

2.2.1 Künstliche neuronale Netzwerke

Künstliche neuronale Netzwerke (engl. Artificial Neural Networks - ANNs oder KNNs) werden verwendet, um komplexe Muster in Daten zu finden und intelligente Systeme zu bauen. Die bekannteste Form der künstlichen neuronalen Netzwerke ist der mehrschichtige Perzeptron (engl. Multilayer Perceptron - MLP) [52]. Diese Netzwerke werden auch vollständig verbundene Netzwerke genannt. Tiefes Lernen (engl. Deep Learning) wird heute verwendet, um den Inhalt von Bildern, Text und Sprache zu verstehen. Ebenfalls haben diese Ansätze oft bei bestimmter Anwendung die traditionellen Methoden weit hinter sich gelassen. Die Systeme dazu erstrecken sich von Apps für mobile Geräte bis hin zum autonomen Fahren. Die künstlichen neuronalen Netzwerke sind an das riesige Netz der im menschlichen Nervensystem und Gehirn miteinander verbundenen Neuronen angelehnt. So enthält das vollständig verbundene Netzwerk Knoten (Neuronen), die mit allen Knoten der nachfolgenden Schicht verbunden sind. Jeder einzelne Knoten enthält eine Gewichtung w, die beim Training des künstlichen neuronalen Netzwerks durch die Fehlerrückführungsalgorithmen angepasst werden. Anschließend folgt für jede Schicht des künstlichen neuronalen Netzwerks eine Aktivierungsfunktion. Diese Aktivierungsfunktion σ aktiviert oder deaktiviert das Neuron anhand der Eingaben. Dadurch wird die Ausgabe jedes einzelnen Neurons gesteuert. Die häufig verwendeten Aktivierungsfunktionen sind logistische Funktion, SoftMax-Funktion, Sigmoidfunktion, Hyperbeltangens (tanh) und Rectified Linear Unit (ReLU) [52]. So kann das Beschriebene zu dieser Formel für eine

Schicht zusammengefasst werden [26]:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i\right) \tag{2.1}$$

Bei diesem Ansatz werden dem künstlichen neuronalen Netzwerk mitunter Millionen von Datenpunkten x zur Verfügung gestellt, um ein Muster in diesen Rohdaten zu erkennen. An dieser Stelle werden die Merkmale der Daten extrahiert und in eine besser nutzbare Form gebracht. Mit diesem Ansatz können komplexe Eigenschaften erfasst und gelöst werden. Dabei werden kleinere Informationssteine miteinander kombiniert, um schwierige Aufgaben wie die Klassifikation von Bildern zu lösen. Die Abbildung 2.1 zeigt ein Beispiel für die Klassifikation von handgeschriebenen Ziffern in Bildern aus der öffentlich verfügbaren MNIST-Datenbank (engl. Modified National Institute of Standards and Technology Database) [49]. Die Bilder haben eine Größe von 28 × 28 Pixel. Somit wird die Anzahl der Eingangsneuronen auf 784 gesetzt. Die Anzahl der Ausgabeneuronen beträgt 10. Dies entspricht den Ziffern von null bis neun [63].

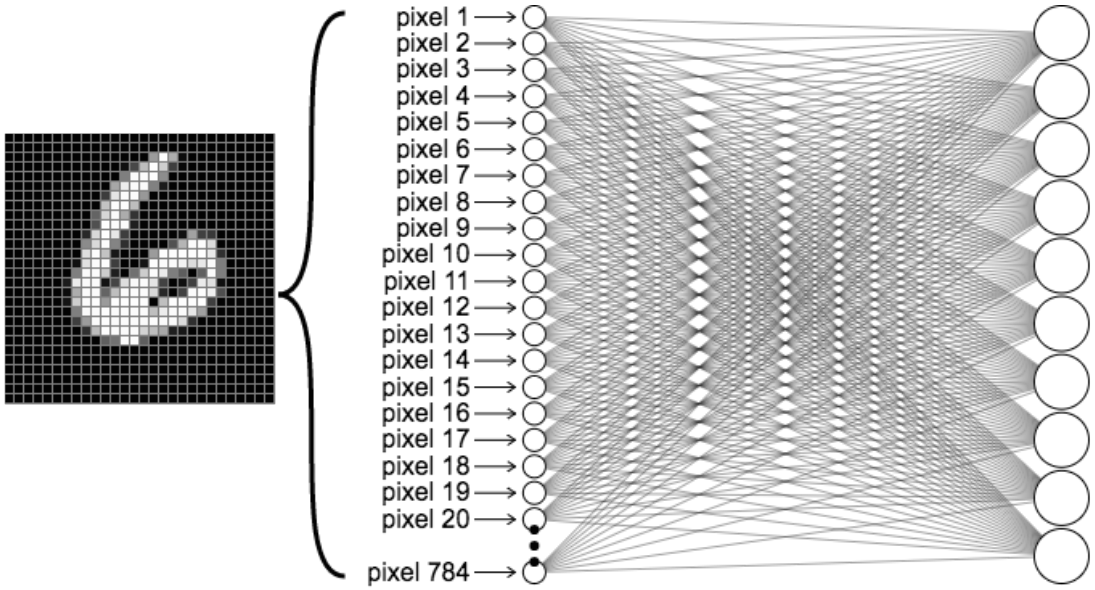


Abbildung 2.1: Künstliches neuronales Netz mit einer Schicht für die handgeschriebenen Zahlen in Bildern aus der MNIST-Datenbank [49]. Die Dimension der Eingabebilder ist 28 × 28 Pixel. Somit gibt es 784 Eingangsneuronen. Die 10 Ausgangsneuronen entsprechen 10 Klassen, also den 10 Ziffern von 0 bis 9 [63].

Die künstlichen neuronalen Netzwerke stechen auf dem Gebiet des Computersehens (engl. Computer Vision) besonders hervor [34]. Das Computersehen wird in Abschnitt 2.3 detaillierter erklärt. Die künstlichen neuronalen Netzwerke können auch zu „lang“ oder zu „kurz“ trainiert werden. Hier kann auch die Modellkomplexität für einen bestimmten

Anwendungsfall falsch gewählt werden. Dabei spricht man von einer Überanpassung (engl. Overfitting) oder Unteranpassung (engl. Underfitting) des neuronalen Netzwerks. Eine Überanpassung entsteht, wenn ein Modell zu sehr an die Trainingsdaten angepasst wird und dabei sehr gut für die Trainingsdaten funktioniert. Sobald das überangepasste Modell nach dem Training neue Daten bekommt, ist das Ergebnis nicht mehr zufriedenstellend. Eine Unteranpassung hingegen entsteht, wenn das Modell auch auf den Trainingsdaten schlecht abschneidet. Je komplexer das Modell wird, desto bessere Vorhersagen kann dieses Modell auf den Trainingsdaten treffen. Sobald dieses Modell zu komplex wird, konzentriert es sich zu sehr auf jeden einzelnen Datenpunkt in den Trainingsdatensätzen. Dadurch findet keine Verallgemeinerung des Datenmusters statt [73].

2.2.2 Faltende neuronale Netzwerke

Faltende neuronale Netzwerke werden auch Konvolutionsnetzwerke (engl. Convolutional Neural Networks - CNNs oder ConvNets) genannt. Diese Art von künstlichen neuronalen Netzwerken hat in den letzten Jahren eine Sonderstellung als besonders vielversprechende Art des tieferen Lernens erlangt. Die Abbildung 2.2 zeigt die Architektur eines faltenden neuronalen Netzwerks bildlich dar. Dabei werden aus dem Eingabebild durch die Konvolutionsschichten die Merkmale extrahiert. Im nächsten Schritt wird eine Klassifikation durch die vollständig verbundenen Schichten mit anschließender Wahrscheinlichkeitsverteilung durch die SoftMax-Aktivierungsfunktion durchgeführt [99].

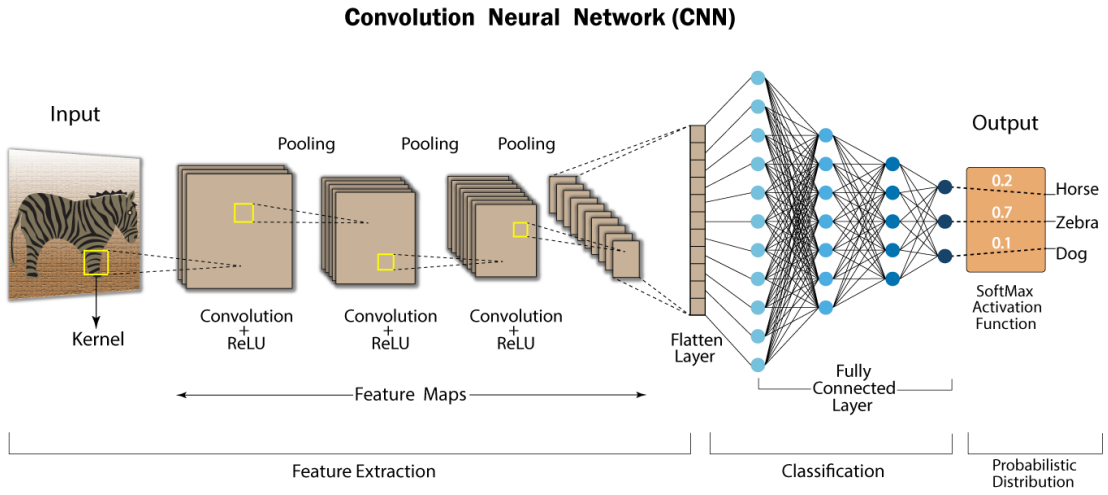


Abbildung 2.2: Architektur eines faltenden neuronalen Netzwerks inklusive der Merkmalsextraktion durch die Konvolutionsschichten, Klassifikation durch die vollständig verbundenen Schichten und anschließender Wahrscheinlichkeitsverteilung durch die SoftMax-Aktivierungsfunktion [99].

Der wesentliche Unterschied der vollständig verbundenen Netzwerke und der Konvolutionsnetzwerke ist das Verbindungsmuster der aufeinanderfolgenden Schichten. In einer vollständig verbundenen Schicht, wie der Name schon sagt, ist jeder Knoten mit allen

Knoten aus der vorherigen Schicht verbunden. In der Konvolutionsschicht eines CNNs hingegen ist jeder Knoten mit einer festen Anzahl an Knoten der nachfolgenden Schicht verbunden. Die Knoten der Konvolutionsschicht verwenden ebenfalls die gleichen Gewichte für diese Verbindungen. Diese Gewichte kann man sich als kleine „Fenster“, die sich über das Eingabebild bewegen, vorstellen. Diese „Fenster“ werden auch Filter genannt. Durch die gleichen Gewichte bekommt jede Konvolutionsschicht die gleichen Merkmale. Dabei wird dieselbe Berechnungsvorschrift auf unterschiedliche Teile des Bilds angewendet. In einem Bild kann der gleiche Inhalt an unterschiedlichen Stellen vorhanden sein. So kann zum Beispiel bei der Objekterkennung eine Katze unabhängig von der Lage, Rotationen und unterschiedlichen Lichtverhältnissen im Bild gefunden werden. Dabei spricht man von einer Invarianz [35]. Für die Implementierung einer Faltungsschicht beziehungsweise Konvolutionsschicht (engl. Convolutional Layer) des faltenden neuronalen Netzwerks müssen die Parameter für Ausgabe (engl. Output), Kernel (engl. Kernel), Schrittweite beim Verschieben (engl. Stride), Auffüllen (engl. Padding) und Aktivieren (engl. Activation) gesetzt werden. Dabei ist Output die Anzahl der Ausgangsfilter in der Faltung. Die Dimension des Kernels gibt die Höhe und Breite des 2D-Faltungsfensters an, das sich über die Pixel im Eingabebild bewegt. Stride spezifiziert die Schrittweite der Faltung entlang der Höhe und Breite und ist nahezu symmetrisch in der Dimension [103]. Padding ist der Parameter für das Auffüllen der Pixel am Randbereich des Bilds. Mit dem Parameter für die Aktivierung kann eine Aktivierungsfunktion angegeben werden. Diese bestimmt den Aktivierungszustand jedes einzelnen Neurons. Nach einer Konvolutionsschicht kann ebenfalls ein Zusammenlegen beziehungsweise Vereinigen (engl. Pooling) auf die Ausgabedaten angewendet werden. Dabei wird die Menge der weitergereichten Daten reduziert. Dadurch wird zusätzlich die Gesamtanzahl der Modellparameter drastisch verringert. Durch das Pooling entsteht eine räumliche Aggregation der Merkmale, die es dem CNN erlaubt die Invarianzen im Bild zu übersehen. Somit werden die berechneten Merkmale unempfindlich gegenüber kleineren Verschiebungen im Bild. Die Konvolutionsschichten können ebenfalls mit der Regularisierungstechnik namens Dropout ergänzt werden. Dabei wird das Netz gezwungen, die angelernte Darstellung über sämtliche Neuronen zu verteilen. Bei dieser Technik wird ein zufälliger, vorher bestimmter, Anteil an Knoten einer Schicht abgeschaltet. Dieses führt zu einer besseren Verallgemeinerbarkeit des Netzwerks [35].

2.2.3 Fehlerrückführung

Dieser Abschnitt präsentiert den für die Praxis wichtigsten Algorithmus zur Fehlerrückführung (engl. Backpropagation of Error). Dieser Algorithmus ist ein Feedback-Prozess, um die Gewichte beim Training des künstlichen neuronalen Netzwerks anzupassen. Nach der Berechnung der Netzausgabe (Vorwärtspropagieren) wird der Approximationsfehler für ein Trainingsbeispiel berechnet. Dieser ist für das Rückwärtspropagieren notwendig, um Schicht für Schicht die Gewichte rückwärts zu optimieren. Dieser Vorgang wird solange wiederholt, bis die Gewichte sich nicht mehr verändern oder eine Zeitschranke erreicht ist. Beim Lernen des künstlichen neuronalen Netzwerks wird oft mit einer höheren Lernrate η gestartet, die anschließend verkleinert wird. Außer den Eingabeneuronen berechnen

alle Neuronen ihren aktuellen Wert x_j nach der Vorschrift

$$x_j = \sigma\left(\sum_{i=1}^{n} w_{ji} x_i\right) \tag{2.2}$$

mit zum Beispiel der Sigmoidfunktion

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.3}$$

Anschließend werden die Gewichte entsprechend dem negativen Gradienten der über die Ausgabeneuronen summierten quadratischen Fehlerfunktion

$$E_p(w) = \frac{1}{2} \sum_{k \in Ausgabe} (t_k^p - x_k^p)^2 \tag{2.4}$$

für das Trainingsmuster p geändert:

$$\Delta_p w_{ji} = -\eta \frac{\partial E_p}{\partial w_{ji}}. \tag{2.5}$$

Daraus ergibt sich die Backpropagation-Lernregel

$$\Delta_p w_{ji} = \eta \delta_j^p x_i^p \tag{2.6}$$

mit

$$\delta_j^p = \begin{cases} x_j^p(1 - x_j^p)(t_j^p - x_j^p) & \text{falls } j \text{ Ausgabeneuron ist} \\ x_j^p(1 - x_j^p) \sum_k \delta_k^p w_{kj} & \text{falls } j \text{ verdecktes Neuron ist,} \end{cases} \tag{2.7}$$

die ebenfalls als verallgemeinerte Deltaregel bezeichnet wird [27]. Zusätzlich gibt es verschiedene Methoden, um einen bestimmten Fehler zu messen. Diese können zum Beispiel die mittlere quadratische Abweichung oder der mittlere absolute Fehler sein. Dieses Fehlermaß kann für jeden verschiedenen Anwendungszweck unterschiedlich sein. Ein in dieser Forschungsarbeit verwendetes Fehlermaß ist der mittlere absolute Fehler (engl. Mean Absolute Error - MAE) [119]. Dieser wird in Kapitel 4 und 7 verwendet und ist wie folgt definiert:

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |y_i - x_i| \tag{2.8}$$

Dabei handelt es sich um das arithmetische Mittel der absoluten Fehler mit y_i als vorhergesagten Wert und x_i als tatsächlichen Wert. Der beschriebene Ablauf des Lernens des künstlichen neuronalen Netzwerks ist als Pseudocode im nachfolgenden Algorithmus 1 angegeben.

Algorithmus 1: Backpropagation [27]

Eingabe : Trainingsbeispiele, Lernrate η
Ausgabe : Anpassung der Gewichte w

initialisiere alle Gewichte w_j zufällig
tue
  für *alle* $(q^p, t^p) \in$ *Trainingsbeispiele* **tue**
    lege den Anfragevektor q^p für die Eingabeschicht an
    `/* Vorwärtspropagieren */`
    für *alle Schichten ab der ersten verdeckten aufwärts* **tue**
      für *alle Neuronen der Schicht* **tue**
        berechne die Aktivierung mit $x_j = \sigma(\sum_{i=1}^{n} w_{ji} x_i)$
    berechne den quadratischen Fehler mit $E_p(w)$
    `/* Rückwärtspropagieren */`
    für *alle Lagen von Gewichten ab der letzten abwärts* **tue**
      für *jedes Gewicht* w_{ji} **tue**
        berechne $w_{ji} = w_{ji} + \eta \delta_j^p x_i^p$
bis w *konvergiert oder Zeitschranke erreicht*

2.3 Computer Vision

Das Computersehen (engl. Computer Vision) oder maschinelles Sehen umfasst verschiedene Methoden zur Erfassung, Verarbeitung und Analyse von digitalen Bildern in einem Computer. Das maschinelle Sehen ist dem recht jungen Gebiet der Computervisualistik zuzuordnen. Dieses vereint die Computer Informatik mit den digitalen Bildern (Visualisierung) [84]. Visualisierungen beinhalten die grafische Darstellung von Daten und Wissen. Dabei spielen künstlerische, ästhetische und ergonomische Aspekte inklusive des Designs von Eingabegeräten eine Rolle. Wird zum Beispiel ein digitales Bild mit einer digitalen Kamera aufgenommen, so kann anschließend diese Bildaufnahme auf dem Computer bearbeitet, gespeichert und angezeigt werden. Die Technik der Bildaufnahme und der Speicherung der digitalen Bilder ist an dieser Stelle eine komplette Wissenschaft für sich. Dabei kann die Computergrafik helfen. Hierbei spricht man von einer Wissenschaft der Erzeugung von Bildern im Computer für Menschen. Diese Grafiken reichen von fotorealistischen und nicht fotorealistischen Bildern über Animationen, Filmen bis hin zur virtuellen Realität. Maschinelles Sehen behandelt hingegen den umgekehrten Weg der Bilder aus der Realität in den Computer. Dabei stehen die Theorie und Technik der Bildverarbeitung und Bildanalyse im Vordergrund. Die Bildverarbeitung und Bildanalyse lässt sich unterteilen in Bildvorverarbeitung, Detektion elementarer Merkmale wie Kanten und geraden Linien, Segmentierung, Analyse elementarer Formen und Identifikation von Objekten. Das Ziel der Bildvorverarbeitung ist es die Bilder für den menschlichen Betrachter angenehmer darzustellen. Dabei spricht man von dem Entfernen von Rot in den Augen und die Optimierung der Helligkeit und des Kontrastes im Bild. Diese Optimierung ist für

den menschlichen Betrachter zwar schöner, kann aber bei der anschließenden Analyse im Rechner sogar empfindlich stören. Somit muss an dieser Stelle zwischen dem Nutzen einer Bildaufnahme für das Fotoalbum oder Analyse im Computer unterschieden werden. Die Bildanalyse kann hierbei als ein spezielles Teilgebiet der künstlichen Intelligenz angesehen werden [84].

2.3.1 Spurerkennung

Bei der Spurerkennung handelt es sich in erster Linie um eine automatisierte Identifizierung der eigenen Fahrspur des Fahrzeugs. Nachdem der Spurverlauf erkannt wurde, kann das autonome Fahrzeug automatisch in der Fahrspur gehalten werden. Zusätzlich können die Fahrspuren der Nachbarfahrzeuge untersucht werden. Die Entwicklung der Spurerkennung in Kapitel 4 orientiert sich an die autonome Fahrzeugindustrie. Denn in der Industrie für autonome Fahrzeuge gibt es Fahrzeuge, die bereits eigenständig und ohne Fahrer fahren können. Dazu müssen diese Fahrzeuge den Spurverlauf exakt erkennen, um von der Fahrbahn nicht abzukommen und die Insassen nicht zu verletzen [69]. Die Spurmarkierungen sind als durchgezogene oder gestrichelte Linien links und rechts auf der Fahrbahn markiert und sind für das menschliche Auge direkt sichtbar. Doch für die autonomen Fahrzeuge ist es nicht so einfach. Diese Linien der Spur müssen von dem autonomen Fahrzeug von anderen Linien im Bild unterschieden und getrennt werden. Durch diese Trennung der Linien in einem zweidimensionalen Bild kann die Mitte der Fahrbahn im nächsten Schritt errechnet werden. Die Spurerkennung ist nicht neu und wird seit längerem in der Fahrzeugindustrie erforscht. Seit einiger Zeit gibt es in einigen Fahrzeugen bereits serienmäßig den Spurassistenten (engl. Lane Departure Warning Systems - LDWS) oder den Spurhalteassistenten (engl. Lane Keeping Assist Systems - LKAS). Dieser warnt den Fahrer, wenn das Fahrzeug von der Fahrbahn abkommt oder kann sogar eine Zeitlang das Fahrzeug eigenständig in der Fahrspur halten. Zum Beispiel bei Nissan Motors seit 2001 [44], bei Citroën seit 2005 [86] und bei Audi seit 2007 [4]. Das Problem der akademischen Forschung ist, dass diese Algorithmen und Vorgehen, die sich bereits in der Fahrzeugindustrie für autonome Fahrzeuge etabliert haben, unter Verschluss gehalten werden und nicht frei zugänglich sind. Dies ist natürlich wegen dem Konkurrenzkampf der Hersteller der Fall. Aus diesem Grund müssen teilweise eigene Algorithmen und Vorgehen in dem akademischen Bereich erforscht und entwickelt werden. Der Fahrspurverlauf kann mit den Ansätzen des maschinellen Lernens, zum Beispiel mit den künstlichen neuronalen Netzwerken oder den faltenden neuronalen Netzwerken, erkannt werden. Dieses ist ebenfalls mit den traditionellen Methoden, zum Beispiel durch eine Kantenerkennung mit dem Canny-Algorithmus und anschließender Hough-Transformation, möglich. Diese Algorithmen werden in den nachfolgenden Abschnitten erklärt.

2.3.2 Canny-Algorithmus

Dieser Abschnitt präsentiert den Canny-Algorithmus [15], der für die Kantenerkennung in dem Kapitel 4 eingesetzt wurde. Dieser Kantenoperator gilt auch heute noch als „State

of the Art“ unter den Kantenerkennungsalgorithmen. Die Canny-Methode versucht gleichzeitig drei Ziele zu erreichen. Dabei wird versucht die echten Kanten möglichst zuverlässig zu detektieren, die Position der Kanten präzise zu bestimmen und die Anzahl falscher Kantenmarkierungen zu minimieren. Einfache Kantenoperatoren basieren typischerweise auf den Ableitungen erster Ordnung und nachfolgender Schwellwertoperation. Diese erfüllen die vorgestellten Ziele in der Regel nicht. Der Canny-Algorithmus ist im Kern ebenfalls ein Gradientenverfahren. Dieser benutzt zur Lokalisierung der Kanten auch die Nulldurchgänge der zweiten Ableitungen. Diese befinden sich an jeder Stelle, wo die ersten Ableitungen ein lokales Maximum oder Minimum aufweisen. Diese Kantenerkennung kann allgemein in folgende drei Schritte zusammengefasst werden [13]:

1. **Vorbereitung:** Das Eingabebild wird mit einem Gaußfilter einer bestimmten Größe geglättet, da die Kantenerkennung für Rauschen im Bild anfällig ist. Dadurch wird ebenfalls die Skalenebene des Kantendetektors spezifiziert, welche das Ergebnis der verschiedenen gerichteten Filter enthält. Aus dem geglätteten Bild wird für jede Position der x/y-Gradient inklusive dessen Betrag und Richtung berechnet.

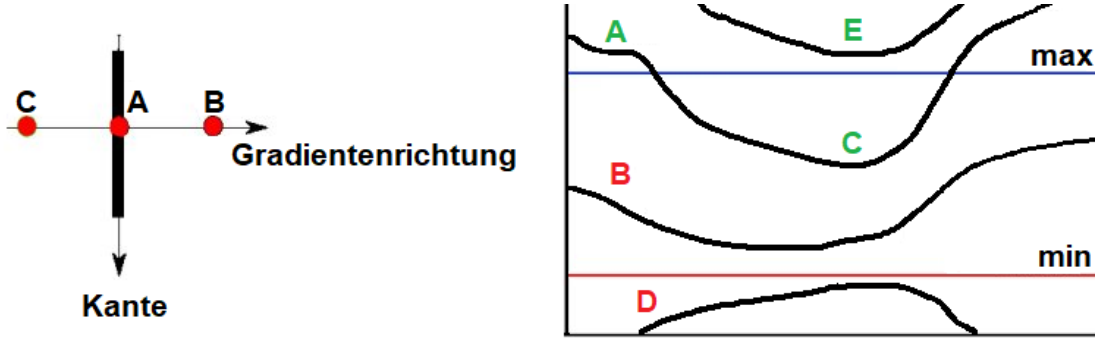
2. **Kantenlokalisierung:** Alle Positionen (Pixel) werden als Kantenpunkte markiert, an denen der Betrag des Gradienten ein lokales Maximum entlang der zugehörigen Gradientenrichtung aufweist. Dabei werden die nicht als Kante markierten Pixelpositionen entfernt. Dieses veranschaulicht die Abbildung 2.3a. Wie man erkennen kann liegt der Punkt A auf der Kante in vertikaler Richtung. Die Gradientenrichtung verläuft orthogonal zu der Kante. Die Punkte B und C liegen in Gradientenrichtung. Somit wird der Punkt A mit den Punkten B und C auf ein lokales Maximum überprüft. Wenn dies der Fall ist, wird dieser Punkt für die nächste Stufe berücksichtigt, andernfalls wird er unterdrückt (auf 0 gesetzt). Anschließend ist das Ergebnis ein Binärbild mit dünnen Kanten [78].

3. **Kantenselektion und -verfolgung:** Abschließend werden unter Verwendung eines Hysterese-Schwellwerts zusammenhängende Ketten von Kantenelementen gebildet. Hierbei wird anhand von zwei Schwellwerten min und max entschieden, welche der gefundenen Kanten wirklich eine Kante ist und welche nicht. Alle Kanten mit einem Intensitätsgradienten von mehr als max sind echte Kanten. Alle Kanten unter dem Wert von min sind keine echten Kanten und werden daher verworfen. Die Kanten in dem Bereich von min und max werden basierend auf deren Verbindungen zugeordnet. Wenn eine Kante sich in dem Schwellwertbereich von min und max befindet und mit einer echten Kante verbunden ist, gilt diese Kante ebenfalls als eine Kante. Analog dazu werden die nicht mit sicheren Kanten verbundenen Kanten verworfen [78].

   Die Abbildung 2.3b stellt dieses Vorgehen bildlich dar. Die Kanten A und E sind über dem Schwellwert max und sind echte Kanten. Die Kante D befindet sich unter dem Schwellwert min und wird verworfen. Die Kante B befindet sich zwischen min und max. Diese ist mit keiner echten Kante verbunden und wird somit verworfen. Die Kante C befindet sich ebenfalls in dem Schwellwertbereich von min und max.

Diese ist mit der echten Kante A verbunden und wird daher als eine echte Kante klassifiziert.

Der in diesem Abschnitt beschriebene Canny-Kantenerkennungsalgorithmus kann durch die OpenCV-Bibliothek in Python realisiert werden [78]. Mehr zu der OpenCV-Bibliothek wird in dem Abschnitt 2.5.7 erläutert. Ein Beispielbild der Canny-Kantenerkennung zeigt die Abbildung 2.5a im nachfolgenden Abschnitt.



(a) Gradienten- und Kantenrichtung. (b) Kantenselektion und -verfolgung.

Abbildung 2.3: Veranschaulichung der Kantenlokalisierung, -selektion und -verfolgung (orientiert an [78]).

2.3.3 Hough-Transformation

Die Hough-Transformation ist eine wichtige mathematische Methode, um verschiedene geometrische Formen in Bildern zu finden. Mit geometrischer Form sind die Formen wie Geraden, Kreise oder Ellipsen gemeint. Diese Formen sollten sich in einer mathematischen Form darstellen lassen. Der Ausgangspunkt für die Findung der Koordinaten dieser Formen ist ein zweidimensionales Binärbild [85]. Diese Methode kann die Form erkennen, auch wenn diese gebrochen oder etwas verzerrt ist. Die Hough-Transformation wird unter dem Beispiel einer Geraden beziehungsweise Linie erläutert. Durch die Erkennung der Häufungen von Punkten in der Parameterdarstellung können Linien im Bild gefunden und extrahiert werden. Eine Linie kann mathematisch im kartesischen Koordinatensystem als $y = mx + b$ dargestellt werden. Bei der Hough-Transformation werden die Linien im Polarkoordinatensystem ausgedrückt. Somit kann eine Gerade in dem Polarkoordinatensystem wie folgt mathematisch dargestellt werden:

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right) x + \left(\frac{\rho}{\sin\theta}\right) \tag{2.9}$$

Umgeformt kann die Gerade in Parameterdarstellung als $\rho = x\cos\theta + y\sin\theta$ dargestellt werden. Dabei ist ρ der senkrechte Abstand vom Ursprung zur Linie und θ der Winkel, der durch diese senkrechte Linie und die horizontale Achse gebildet wird. Somit kann

jede Linie durch die Variablen ρ und θ dargestellt werden. Die nachfolgende Abbildung 2.4 beschreibt bildlich die Darstellung der Linien durch ρ und θ. Angenommen das Eingabebild enthält eine horizontale Linie in der Mitte des Bilds. So wird zuerst der erste Punkt der Linie betrachtet. Die x- und y-Koordinaten dieses Punkts sind bekannt. Somit können verschiedene Werte für $\theta \in \{0, 1, \ldots, 180\}$ eingesetzt werden. Dabei werden alle möglichen Geraden ermittelt, die durch diesen einzelnen Bildpunkt laufen. Anschließend werden diese Geraden in einem Akkumulator als (ρ, θ)-Paar mit dem Wert 1 gespeichert. Bei diesem Vorgang wird ein Akkumulator mit den verschiedenen ρ und θ Paaren erzeugt [14]. Nachfolgend wird der nächste Punkt betrachtet. Dafür werden ebenfalls die verschiedenen ρ und θ Paare bestimmt und der Wert an dieser Position inkrementiert. Abschließend können Häufungen von Punkten an den verschiedenen (ρ, θ)-Positionen betrachtet werden, um eine geeignete Linie zu finden. Dabei wird der Akkumulator nach maximalen Werten durchsucht [89].

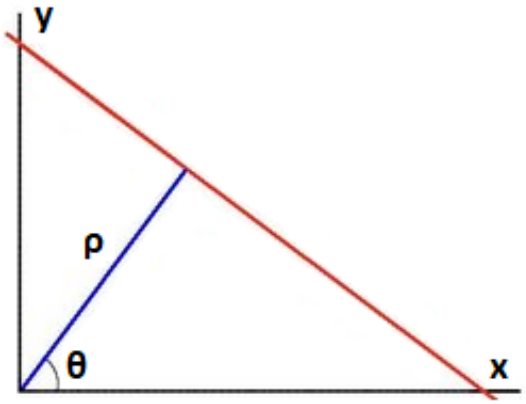
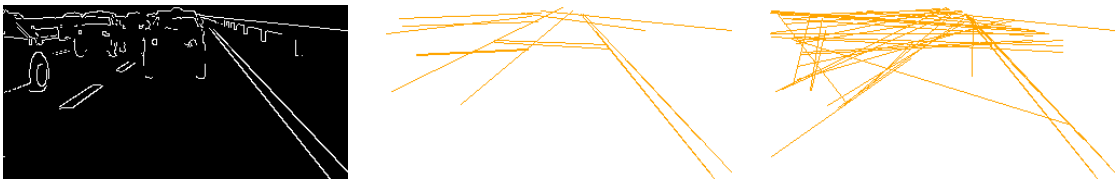


Abbildung 2.4: Veranschaulichung der Hough-Transformation inklusive der Darstellung der Linien durch ρ und θ.

Bei der Hough-Transformation kann ebenfalls ein Schwellwert angegeben werden. Dieser Schwellwert ist die maximale Anzahl von Schnittpunkten einer Linie, um diese zu erkennen [81]. In den vergangenen Jahren wurde die Hough-Transformation optimiert. Die probabilistische Hough-Transformation (engl. Probabilistic Hough Transform) verwendet nicht alle Bildpunkte, sondern nur eine zufällige Teilmenge von Punkten, die für die Linienerkennung ausreicht. Somit werden viel weniger Berechnungen durchgeführt und höhere Laufzeiten erreicht. Die probabilistische Hough-Transformation enthält zwei weitere Parameter für die Erkennung der Linien. Der Parameter für die minimale Länge der Linien (engl. Min Line Length) steuert die Erkennung der Linien anhand der minimalen Länge. Dieser stellt die Mindestlänge der Linie, die erkannt werden sollte, dar. Der Parameter für den maximalen zulässigen Abstand zwischen den Liniensegmenten (engl. Max Line Gap) fasst mehrere Linien zu einer einzigen Linie und behandelt diese als eine einzelne Linie [89]. Diese erweiterte Hough-Transformation wird in dem Kapitel 4 dieser Forschungsarbeit verwendet [16]. Die Abbildungen 2.5b und 2.5c zeigen die Findung der Linien anhand der eingeführten Parameter aus dem Canny-Kantenerkennungsbild (Abb.

2.5a). Die in diesem Abschnitt beschriebene Hough-Transformation kann ebenfalls durch die OpenCV-Bibliothek in Python realisiert werden [81]. Die OpenCV-Bibliothek enthält ebenfalls Implementierungen zu der probabilistischen Hough-Transformation [89].



(a) Kantenbild. Schwellwert 1 (min) = 350, Schwellwert 2 (max) = 400.

(b) Hough-Transformation. Länge (min) = 100, Abstand (max) = 30.

(c) Hough-Transformation. Länge (min) = 1, Abstand (max) = 320.

Abbildung 2.5: Probabilistische Hough-Transformation mit verschiedenen Parametern auf dem Canny-Kantenerkennungsbild mit einer Größe von 320 × 160 Pixel aus Kapitel 4. Der Parameter für den Schwellwert wurde auf 15 gesetzt.

2.3.4 Objekterkennung

Die Objekterkennung ist dem Bereich der Bildverarbeitung des maschinellen Sehens zuzuordnen. Dabei wird versucht in jedem einzelnen Bild die Objekte zu lokalisieren und zu klassifizieren. Bei diesem Gebiet der Bildverarbeitung wird das einzelne Bild in wichtige Teile unterteilt und auf Merkmale untersucht. Anschließend wird jedes Teil des Bilds einer Klasse der Objekte zugeordnet. So kann zum Beispiel die Position einer Katze oder eines Hunds im Bild gefunden werden [98]. Dieser Abschnitt beschreibt die Objekterkennung unter dem Beispiel von autonomen Fahrzeugen. Die autonomen Fahrzeuge besitzen mehrere Sensoren und Kameras, unter anderem auch eine Kamera, gerichtet auf die Fahrbahn. Mithilfe dieser Kamera können verschiedene Objekte, wie Menschen, Fahrzeuge oder Tiere, im Sichtfeld der Kamera auf der Fahrbahn als ein zweidimensionales Bild geliefert werden. Anschließend müssen diese Objekte in diesem Bild gefunden und klassifiziert werden. Zusätzlich kann die Position eines Objekts auf der Fahrbahn ermittelt werden, um Kollisionen zu vermeiden. Dabei kann eine zusätzliche Implementierung der Spurerkennung helfen. Damit zum Beispiel die Kollisionserkennung flüssig funktionieren kann, ist hierbei eine Echtzeitobjekterkennung die Voraussetzung. Genau aus diesem Grund konzentriert sich das Kapitel 6 auf die Echtzeitobjekterkennung von individuellen Objekten auf einer Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte. Für die Objekterkennung wird meistens eine rechenstarke Hardware benötigt, wie zum Beispiel eine Grafikkarte, da die Bildverarbeitung sehr rechenintensiv ist. Zusätzlich wird die Objekterkennung mit den faltenden neuronalen Netzwerken verarbeitet. Diese sind für die gute Verarbeitung der Bilder bekannt. Darüber hinaus haben sich die ConvNets bei der Objekterkennung inklusive der Konturfindung erwiesen. Zusätzlich wird seit längerer Zeit die Objekterkennung in der Fahrzeugindustrie erforscht. Seit 2014 gibt es zum Beispiel bei Tesla bereits die Erkennung von Passanten, Fahrrädern oder anderen

Fahrzeugen [118]. Auch hier ergibt sich das Problem der akademischen Forschung, dass diese Algorithmen und Vorgehen, die sich bereits in der Fahrzeugindustrie für autonome Fahrzeuge etabliert haben, unter Verschluss gehalten werden und nicht frei zugänglich sind. So müssen teilweise eigene Algorithmen und Vorgehen bei der Objekterkennung in dem akademischen Bereich erforscht und entwickelt werden. Eine sehr häufig verwendete Metrik für die Objekterkennung ist die mittlere durchschnittliche Genauigkeit (engl. Mean Average Precision - mAP). Diese Metrik wird in dem Kapitel 6 für die Objekterkennung verwendet und ist wie folgt definiert [40]:

$$mAP = \frac{1}{n} \sum_{k=1}^{n} \int_0^1 p_k(r) dr \tag{2.10}$$

Die neuesten Forschungsarbeiten enthalten in der Regel nur Ergebnisse für den COCO-Datensatz mit der COCO mittleren durchschnittlichen Genauigkeit [40]. Diese misst die Genauigkeit von verschiedenen Objektdetektoren wie zum Beispiel Faster R-CNN, SSD, EfficientDet oder CenterNet. In dem Kapitel 6 wird hauptsächlich die SSD (engl. Single Shot Detector) Architektur verwendet, da diese schnell und präzise funktioniert. Die Formel für die Berechnung der mittleren durchschnittlichen Genauigkeit basiert auf den Untermetriken wie Konfusionsmatrix (engl. Confusion Matrix), Schnittmenge über Vereinigung (engl. Intersection Over Union - IoU), Präzision (engl. Precision) und Abruf (engl. Recall). Um eine Konfusionsmatrix zu erstellen werden weitere vier Attribute benötigt. Wahr positiv (engl. True Positive - TP) ergibt sich, wenn das Modell eine Klasse vorhergesagt hat und diese mit der echten Klasse übereinstimmt. Bei wahr negativ (engl. True Negative - TN) kann das Modell die Klassenbezeichnung nicht bestimmen und diese gehört ebenfalls nicht zu den echten Klassen. Falsch positiv (engl. False Positive - FP) entsteht, wenn das Modell eine Klasse vorhersagt, diese aber nicht Teil der echten Klassen ist. Bei falsch negativ (engl. False Negative - FN) kann das Modell eine Klasse nicht voraussagen, welche aber zu den echten Klassen gehört. Die IoU misst die Überlappung zwischen dem echten minimal umgebenden Rechteck (engl. Bounding Box) und der vorhergesagten Box des Modells. Dabei wird gemessen, wie stark sich die vorhergesagten Koordinaten mit den tatsächlichen echten Koordinaten überschneiden. Ein höherer IoU-Wert zeigt an, dass die vorhergesagten Koordinaten der Box zu den echten Koordinaten der Box ähnlicher sind. Dies wird in der Abbildung 2.6 veranschaulicht. Für den IoU-Wert kann anschließend ein Schwellwert (engl. Threshold) gesetzt werden, um die für die Konfusionsmatrix notwendigen Attribute zu bestimmen [91]. Dieses zeigt der Vergleich der Abbildungen 2.6a und 2.6b. Anschließend können die Metriken Präzision p (Gl. 2.11) und Abruf r (Gl. 2.12) wie folgt bestimmt werden:

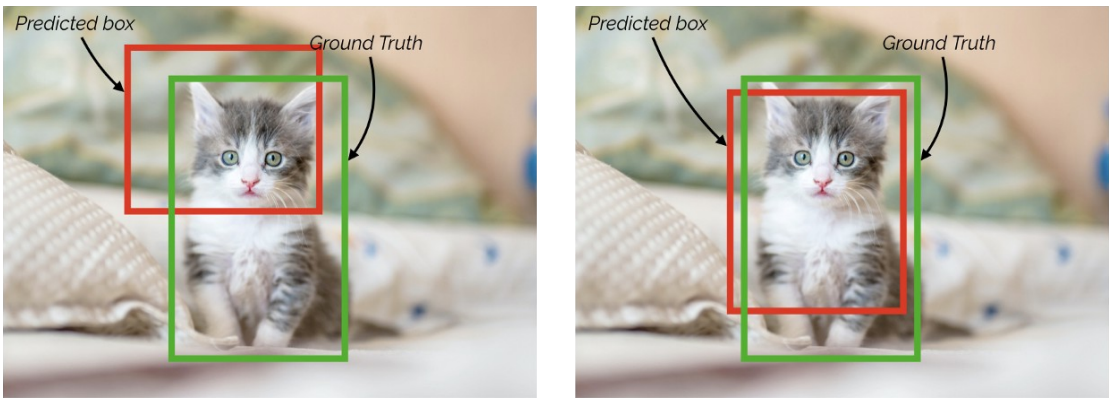
$$p = \frac{TP}{TP + FP} \tag{2.11}$$

$$r = \frac{TP}{TP + FN} \tag{2.12}$$

So ergibt sich die allgemeine Definition für die durchschnittliche Präzision (engl. Average Precision - AP) mit der Fläche unter der Präzisions-Recall-Kurve $p_k(r)$ in Gleichung 2.10.

Diese Berechnung erfolgt pro Klasse k. Zum Schluss kann die mittlere durchschnittliche Genauigkeit für alle Klassen n bestimmt werden [40]. Die Berechnung der mittleren durchschnittlichen Genauigkeit (mAP) kann in folgende Schritte zusammengefasst werden [91]:

1. Erzeugung der Vorhersageergebnisse mit Hilfe des Modells.
2. Umwandlung der Vorhersageergebnisse in Klassenbezeichnungen.
3. Berechnung der Konfusionsmatrix (TP, FP, TN, FN).
4. Berechnung der Präzisions- und Recall-Metriken.
5. Berechnung der Fläche unter der Präzisions-Recall-Kurve.
6. Berechnung der durchschnittlichen Genauigkeit (AP).
7. Berechnung des Mittelwerts von der durchschnittlichen Genauigkeit über alle Klassen.



(a) Falsch positiv (False Positive - FP). IoU $\approx 0,3$.

(b) Wahr positiv (True Positive - TP). IoU $\approx 0,7$.

Abbildung 2.6: Darstellung der notwendigen Attribute für die Konfusionsmatrix (IoU-Schwellwert = 0,5) [91].

2.3.5 Transformation in die Vogelperspektive

In diesem Abschnitt wird die Technik der Transformation in die Vogelperspektive (engl. Bird's Eye View Transformation) präsentiert. Diese Transformation ist in dem Kapitel 7 von großer Bedeutung. Dabei ist das Ziel das Eingabebild aus einer Seitenansicht in eine Draufsicht umzuwandeln (Abb. 2.7). Bei dem Beispiel des autonomen Fahrens wird versucht, die Fahrbahnmarkierungen links und rechts auf der eigenen Fahrbahn parallel auszurichten. Diese Technik kann unter dem Begriff der digitalen Bildverarbeitung als

geometrische Bildmodifikation eingeordnet werden. Die Transformation in die Vogelperspektive kann in drei Schritte unterteilt werden. Als erstes wird das Eingabebild in einem verschobenen Koordinatensystem dargestellt. Anschließend wird das Bild gedreht und auf eine zweidimensionale Ebene projiziert. Dadurch, dass das Eingabebild als eine zweidimensionale Matrix mit den verschiedenen Pixelwerten dargestellt wird, kann jede Position des Bilds an eine neue Position verschoben werden. Die Rotation eines Bilds kann durch eine Matrixmultiplikation realisiert werden [114]. Durch die Methoden der Bildgeometrie kann anschließend zwischen den verschiedenen Koordinatensystemen einer abgebildeten Szene gewechselt werden.

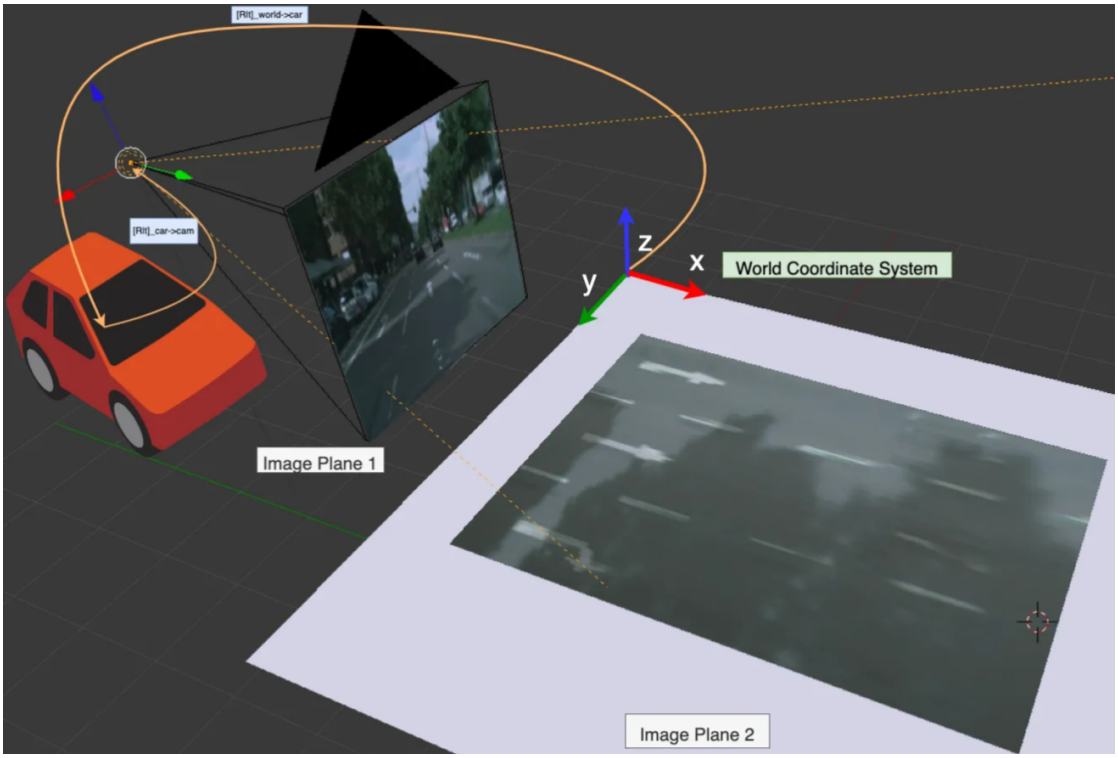
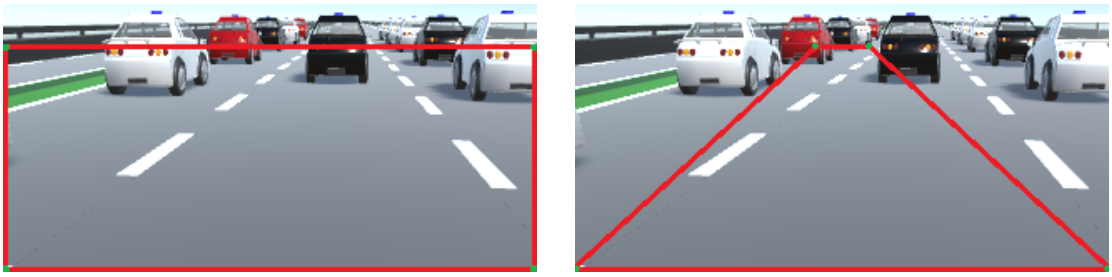


Abbildung 2.7: Seitenansicht der Kamera aus dem Fahrzeug auf die Fahrbahn und die umgewandelte Draufsicht auf die Fahrbahn aus der Vogelperspektive [100].

Angewendet auf den Simulationsdaten in Abbildung 2.8 aus dem Simulator in Kapitel 3 müssen als erstes vier Punkte im Bild für Quelle (engl. Source) *src* festgelegt werden. Dabei spricht man von einem Bereich von Interesse (engl. Region of Interest - ROI). Diese Punkte sind grünfarbig in Abbildung 2.8a dargestellt. Das Eingabebild beträgt hierbei 320 Pixel in der Breite und 160 Pixel in der Höhe. Bei diesem Beispiel sind es die Punkte $src_{xmin}(0,0)$, $src_{xmax}(320,0)$, $src_{ymin}(0,135)$, $src_{ymax}(320,135)$. Nach der Festlegung der Punkte für den Bereich von Interesse, wird das Bild auf exakt diesen Bereich zugeschnitten. Zusätzlich müssen die Koordinaten für das Zielbild (engl. Destination) *dst* angegeben werden. Diese Koordinaten sind $dst_{xmin}(0,0)$, $dst_{xmax}(320,0)$, $dst_{ymin}(144,135)$, $dst_{ymax}(176,135)$ und

sind in der Abbildung 2.8b dargestellt. Dabei werden die y-Koordinaten für dst_{ymin} und dst_{ymax} verändert, um eine Streckung beziehungsweise Stauchung zu erzeugen. Durch die Veränderung dieser Parameter wird versucht, die Fahrbahnmarkierungen links und rechts auf der eigenen Fahrbahn parallel auszurichten.



(a) Bereich von Interesse (ROI) von dem Eingabebild *src*.

(b) Bereich von Interesse (ROI) von dem Zielbild *dst*.

Abbildung 2.8: Darstellung der Transformation in die Vogelperspektive an Simulationsdaten aus dem Simulator in Kapitel 3.

Diese Transformationsparameter können experimentell gefunden werden und sind für jede Ausrichtung der Kamera unterschiedlich. An dieser Stelle beträgt die x-Koordinate bei dst_{ymin} 45 % und die x-Koordinate bei dst_{ymax} 55 % der ursprünglichen Breite. Diese Parameter sind ebenfalls von dem Bereich von Interesse abhängig. Nach der Findung der Parameter kann die Umrechnungsmatrix M aus den vier Paaren von *src*- und *dst*-Koordinaten erzeugt werden [79]. Dabei wird eine 3×3 Matrix wie folgt errechnet:

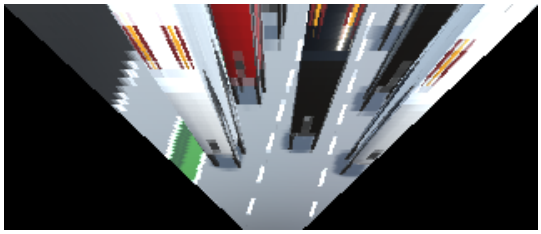
$$\begin{bmatrix} t_i x'_i \\ t_i y'_i \\ t_i \end{bmatrix} = M \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \text{ mit } dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2, 3 \tag{2.13}$$

Anschließend kann eine perspektivische Transformation auf das Eingabebild mit der zuvor erzeugten Matrix (Gl. 2.13) angewendet werden [80]. Dabei ist Destination *dst* das transformierte Ausgabebild in der Vogelperspektive:

$$dst(x, y) = src\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}}\right) \tag{2.14}$$

Die Abbildung 2.9 zeigt anschließend das Ergebnis der Transformation in die Vogelperspektive. Wie man erkennen kann ist das Zielbild (Abb. 2.9b) wie erwartet in der Höhe kleiner als das Eingabebild (Abb. 2.9a). Die in diesem Abschnitt beschriebene Transformation kann durch die OpenCV-Bibliothek in Python realisiert werden. Mehr zu der OpenCV-Bibliothek wird in dem Abschnitt 2.5.7 erläutert.

(a) Eingabebild aus der Kamera des autonomen Fahrzeugs mit 320 × 160 Pixel.



(b) Transformiertes Eingabebild in die Vogelperspektive mit 320 × 135 Pixel.

Abbildung 2.9: Darstellung der Transformation in die Vogelperspektive an Simulationsdaten aus dem Simulator in Kapitel 3.

2.4 Autonomes Fahren

Dieser Abschnitt beinhaltet die Grundlagen und den historischen Hintergrund zum autonomen Fahren. Um einen geeigneten Simulator in Kapitel 3 für Rettungsgassenbildung und Unfallsimulationen mit autonomen Fahrzeugen auf Autobahnen praxisnah zu implementieren, wurde die Technologie der autonomen Fahrzeuge praxisorientiert untersucht. Dazu wurden die Spurerkennung, Rotationserkennung und die Entfernungsmessung zu anderen Fahrzeugen beziehungsweise anderen Objekten aus der Praxis angeschaut. Zusätzlich wurde die Kommunikation der autonomen Fahrzeuge aus der Praxis untersucht. Somit konnte sich diese Forschungsarbeit bei der Entwicklung an den aktuellen Stand der Technik aus der autonomen Fahrzeugindustrie halten. Die Grundlagen und den historischen Hintergrund zum autonomen Fahren beschreiben die nachfolgenden Abschnitte.

2.4.1 Was ist autonomes Fahren?

Seit der Erfindung des Autos kursiert der Begriff *Automobil*, welches durch die Begriffsbildung aus dem griechischen, das Selbstbewegliche bedeutet. Dadurch wurde der Fahrer ohne Unterstützung von Pferden mobil. Doch durch den Umstieg auf die Autos, ging in der Begriffsbildung *Automobil* durch die fehlenden Pferde eine gewisse Form von Autonomie des Gefährts verloren [69]. Durch das lange Training und die Dressur der Kutschpferde konnten diese erlernen sich an einfache Gesetze und Ordnung auf der Straße zu halten. So konnten die Kutschpferde zum Beispiel den Hindernissen auf der Fahrbahn ausweichen oder sich an dem Spurverlauf, durch das Gras des Wegerandes, orientierten und somit die Spur halten. Dadurch ergab sich eine gewisse Autonomie des Gespanns. So konnte ein nicht mehr voll fahrtauglicher Kutscher inklusive des Fuhrwerks sicher nach Hause gebracht werden. Doch durch den Umstieg auf die Autos, war diese Fähigkeit der Hindernisvermeidung und Spurhaltung im Fahrzeug nicht mehr vorhanden. Somit beschäftigt sich der Begriff *autonomes Fahren* mit der Wiederherstellung der Autonomie im Fahrzeug, der die historische Form noch weit übertreffen wird [69].

2.4.2 Lenkverhalten, Spur-, Entfernungs- und Rotationserkennung

An dieser Stelle wird auf das Lenkverhalten, die Spurerkennung, Distanzerkennung, Rotationserkennung und auf die in der Praxis verwendete Sensorik eingegangen. Um das Lenkverhalten der autonomen Fahrzeuge in der Praxis zu bestimmen, wird für diese Fahrzeuge eine Trajektorie bestimmt. Dabei wird ein Pfad, welches das Fahrzeug in den nächsten Sekunden folgen soll, erstellt. Bei diesem Bewegungsplan wird ebenfalls die Geschwindigkeit für das autonome Fahrzeug optimiert [120]. In der Praxis wird ebenfalls oft das lineare Einspurmodell verwendet. Dadurch wird das komplizierte Gebiet des Fahrverhaltens vereinfacht und das Fahrverhalten von zweiachsigen, vierrädrigen Kraftfahrzeugen kann beschrieben werden. Zusätzlich können die kinematischen Größen und Kräfte bei dem Lenk- und Störverhalten beim Seitenwind analysiert werden [70]. Die Spurerkennung beziehungsweise das Halten des Fahrzeugs auf der Fahrbahn wird in der Regel über die Bilderkennung durchgeführt. Dies ist durch die, in der Frontscheibe des Fahrzeugs verbaute Videokamera, möglich. Zusätzlich kann die Erkennung von Ampeln durch diese Kamera erreicht werden. Die Wahrnehmung der Umgebung wird durch einen drehbaren Sensor namens Lidar (engl. Light Detection and Ranging) erledigt. Dieser Sensor ist auf dem Dach des Fahrzeugs installiert, um eine 360° Umgebungssicht bis zu einer Entfernung von 60 Metern für die Fahrzeuge zu erzeugen. Dadurch kann anschließend eine 3D-Kartierung durchgeführt werden, wodurch eine 3D-Umgebung um das fahrende Fahrzeug erstellt werden kann. Die Abstandssensoren (Radar) sind für das automatische Einparken und die Ermittlung des Abstands zum vorderen Fahrzeug notwendig. Der Positionsschätzer (engl. Position Estimator) wird verwendet, um die Erkennung der Rotation des Fahrzeugs und die Ermittlung der Position des Fahrzeugs durchzuführen [5]. Die nachfolgende Abbildung 2.10 fasst diese Sensoren nochmal in einem Bild zusammen.

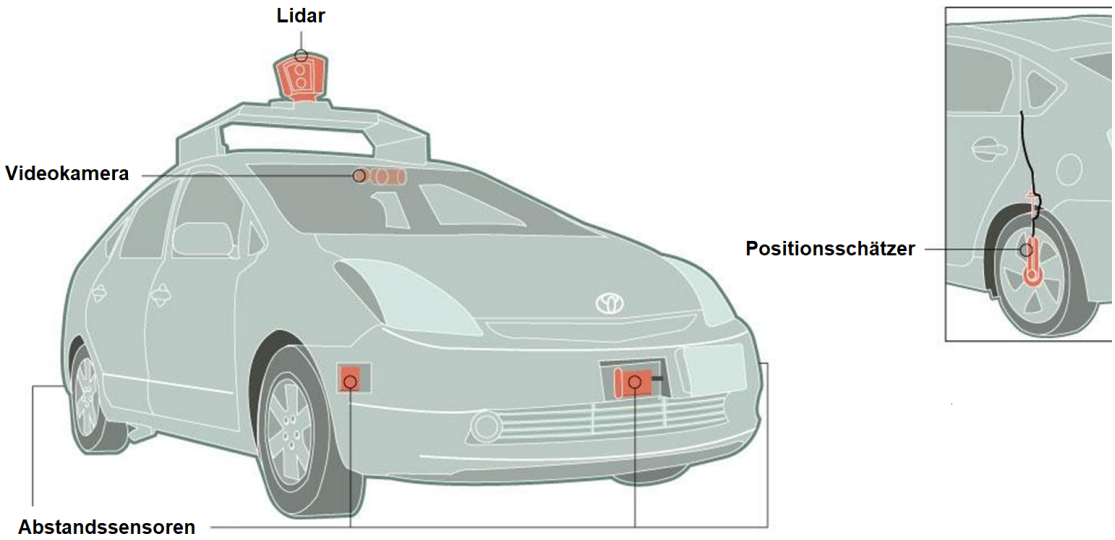
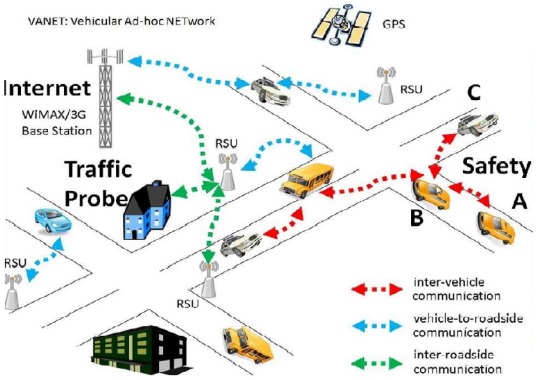


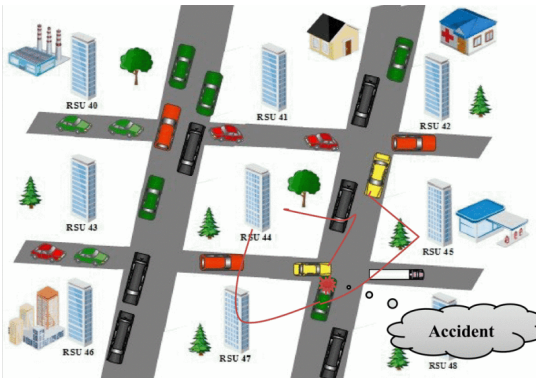
Abbildung 2.10: Sensoren eines autonomes Fahrzeugs (orientiert an [5]).

2.4.3 Kommunikation zwischen autonomen Fahrzeugen

Bei der Kommunikation zwischen autonomen Fahrzeugen spricht man in der Praxis oft von drei Kommunikationsmöglichkeiten. Die erste ist die Fahrzeug-zu-Infrastruktur (engl. Vehicle-to-Infrastructure - V2I) Kommunikation. Dadurch kann zum Beispiel die aktuelle Farbe der Ampel an das ankommende Fahrzeug übermittelt werden. Die zweite Option ist die Fahrzeug-zu-Fahrzeug (engl. Vehicle-to-Vehicle - V2V) Kommunikation. Auf diese Weise kann ein Unfallfahrzeug die anderen Fahrzeuge in der Umgebung benachrichtigen. Die dritte Möglichkeit ist die Infrastruktur-zu-Infrastruktur (engl. Infrastructure-to-Infrastructure - I2I) Kommunikation. Die I2I-Kommunikation kann genutzt werden, um Ampelanlagen intelligent zu schalten. Dadurch können Staus vermieden werden [109]. Um die Kommunikation zu ermöglichen, wird ein Fahrzeug-Ad-hoc-Netzwerk (engl. Vehicular-Ad-Hoc-Network - VANet) aufgebaut [33]. Bei dem Fahrzeug-Ad-hoc-Netzwerk handelt es sich um ein mobiles Funknetz mit Funkknoten und Funkstrecken (engl. Mobile-Ad-hoc-Network - MANet), dessen Knotenpunkte für die Nachrichtenverteilung die Fahrzeuge selbst sind. Die nachfolgende Abbildung 2.11 veranschaulicht die Architektur eines Ad-hoc-Netzwerks (Abb. 2.11a) und ein Beispiel für die Reaktion auf Unfälle (Abb. 2.11b) der autonomen Fahrzeuge in der Praxis.



(a) Architektur eines Ad-hoc-Netzwerks für Fahrzeuge [23]. Dadurch können die Fahrzeuge untereinander Nachrichten austauschen (rot) und über Straßenrandgeräte zusätzlich über z.B. Unfälle benachrichtigt werden (blau). Die Straßenrandgeräte können die Nachrichten über das Internet z.B. an die Rettungskräfte weiterleiten (grün).



(b) Reaktion auf Unfälle im Straßenverkehr [32]. Das Unfallfahrzeug erzeugt und versendet eine CAM-Nachricht. Diese Nachricht wird an andere Fahrzeuge in dieser Gefahrenzone verschickt. Zusätzlich wird die Nachricht an das Straßenrandgerät (RSU) weitergeleitet. So können z.B. die Verkehrsstaus reduziert werden.

Abbildung 2.11: Architektur eines Ad-hoc-Netzwerks und die Kommunikation der Fahrzeuge durch On-Board-Units (OBUs) und Road-Side-Units (RSUs) bei einem Unfall.

Diese Fahrzeuge verfügen über Kommunikationsgeräte (engl. On-Board-Unit - OBU), die für die Kommunikation zwischen den einzelnen Fahrzeugen und den Straßenrandgeräten (engl. Road-Side-Unit - RSU) zuständig sind [32]. Die Fahrzeuge erstellen sogenannte

CAM-Nachrichten (engl. Cooperative Awareness Messages - CAMs). Diese Nachrichten sind Statusinformationen über den Verkehrsfluss, Fahrzeugposition, Fahrgeschwindigkeit, Fahrtrichtung, Fahrzeugstatus und vieles mehr. CAM-Nachrichten werden zwischen Fahrzeugen über die Fahrzeug-zu-Fahrzeug (V2V) Kommunikation oder über die Straßenrandgeräte (RSUs) über die Fahrzeug-zu-Infrastruktur (V2I) Kommunikation mit der Verkehrsleitzentrale ausgetauscht. Diese Nachrichten werden einmal pro Sekunde versendet [21].

2.4.4 Stufen der Automatisierung

Dieser Abschnitt gibt einen Überblick über die Automatisierungsstufen der autonomen Fahrzeuge von der US-Bundesbehörde für Straßen- und Fahrzeugsicherheit (engl. National Highway Traffic Safety Administration - NHTSA) [59]. Diese werden in fünf Stufen unterteilt. Die Klassifizierung der Stufen hängt von den im Fahrzeug verwendeten Technologien und dem Eingriff des Fahrers in den Fahrprozess ab [113]. In der nachfolgenden Tabelle 2.1 werden diese Stufen beschrieben.

Tabelle 2.1: Stufen der Automatisierung von der US-Bundesbehörde für Straßen- und Fahrzeugsicherheit (engl. National Highway Traffic Safety Administration - NHTSA) [59].

Stufe	**Automatisierung**
0	Keine Autonomie des Fahrzeugs, Fahrer hat die Kontrolle
1	Das Fahrzeug gibt dem Fahrer Informationen/Warnungen, Fahrer hat informierte Kontrolle
2	Fahrzeug integriert Erkennung/Reaktion, Fahrer ist bereit, die Kontrolle zu übernehmen
3	Vollständig autonomes Fahrzeug, Fahrer übernimmt im Notfall die Kontrolle
4a	Vollständig autonomes Fahrzeug, Insassen müssen nicht in der Lage sein zu fahren
4b	Vernetzte und kooperierende Fahrzeuge, Optimierter Systembetrieb und passives Fahrerlebnis

Das selbstfahrende Fahrzeug könnte zum Beispiel ab der zweiten Automatisierungsstufe eine Bildung der Rettungsgasse für die Polizei- und Rettungsfahrzeuge bei einem stehenden Verkehr auf einer Autobahn, wie im nachfolgenden Kapitel 3 beschrieben, durchführen. Ab dieser Stufe hat das Fahrzeug eine integrierte Erkennung und Reaktion. Der Fahrer ist bereit die Kontrolle zu übernehmen. Ab der dritten Stufe der Automatisierung könnte das autonome Fahrzeug eine Rettungsgassenbildung bei einem stockenden Verkehr durchführen. Dabei ist das Fahrzeug vollständig autonom und der Fahrer übernimmt nur im Notfall die Kontrolle.

2.5 Programmiersprachen, Plattformen und Bibliotheken

In den nachfolgenden Abschnitten werden die in dieser Forschungsarbeit verwendeten Programmiersprachen, Entwicklungsplattformen und Bibliotheken vorgestellt.

2.5.1 Python

Dieser Abschnitt stellt die Hauptprogrammiersprache dieser Forschungsarbeit vor. Dabei handelt es sich um Python. Diese Programmiersprache erschien im Jahr 1991 und ist neben Perl und Ruby zu einer der beliebtesten interpretierten Programmiersprachen geworden [61]. Durch die zahllosen Frameworks für die Entwicklung von dynamischen Webseiten, Webanwendungen oder Webservices (engl. Webframeworks) lassen sich damit ebenfalls Webseiten gestalten. Aus diesem Grund hat die Beliebtheit von Python im Jahr 2005 stark zugenommen. Seit 2009 ist die Programmiersprache Python eine der wichtigsten Sprachen für die Datenwissenschaft, das maschinelle Lernen und die allgemeine Softwareentwicklung im akademischen und industriellen Bereich geworden. Mit Python können ebenfalls Daten analysiert, zusammengefasst, ausgewertet und visualisiert werden. Ebenfalls ein großer Vorteil der Programmiersprache Python ist die leichte Integration von C-,C++- und Fortran-Code. Dadurch können zum Beispiel Fortran- und C-Bibliotheken für die lineare Algebra, Optimierung oder schnelle Fourier-Transformation verwendet werden. Der Nachteil der Programmiersprache Python ist die langsame Ausführung von Code gegenüber einer kompilierten Programmiersprache wie Java oder C++. Ebenso gibt es seit 2001 einen interaktiven Python-Interpreter namens IPython. Dieser bietet selbst keine Rechen- oder Datenanalysewerkzeuge. Durch interaktives Python wird ein Execute-Explore-Workflow anstelle des typischen Edit-Compile-Run-Workflows unterstützt. Dadurch kann die Produktivität in der Softwareentwicklung erhöht werden. Seit 2014 wurde IPython erweitert und bietet durch das Jupyter-Notebook-Projekt das Ausführen, Debuggen und Testen vom entwickelten Python-Code in einem Webbrowser [61]. Die Spurerkennung in Kapitel 4, die Objekterkennung in Kapitel 6 und die Entfernungsmessung in Kapitel 7 wurden ebenfalls in dieser Forschungsarbeit mit Python 3 beziehungsweise IPython entwickelt.

2.5.2 C#

Die Programmiersprache C# (C-Sharp) ist für viele verschiedene Anwendungen geeignet. Bei dieser Programmiersprache handelt es sich um eine Mehrzweck-Programmiersprache (engl. General Purpose Language). Damit lassen sich unterschiedliche Probleme von Anwenderprogrammen, Web-Applikationen und Hardware-Programmierung lösen. Die C# Programmiersprache ist von Microsoft entwickelt und kommt überwiegend in einer Windows-Umgebung zum Einsatz. Zusätzlich bestehen mittlerweile immer mehr Möglichkeiten C#-Programme für die Betriebssysteme macOS, Linux, Android oder iOS zu implementieren. Somit ist C# universell einsetzbar. C# ist eine Weiterentwicklung von C und weist ebenfalls Ähnlichkeiten zu Java und Einflüsse von C++, Haskell und Delphi auf [8].

2.5.3 Unity

Unity ist eine Echtzeit-Entwicklungsumgebung zur Entwicklung und Gestaltung von Computerspielen (engl. Game Engine) des Unternehmens Unity. Diese Computerspiele können für verschiedene Betriebssysteme wie Windows, Linux und macOS entwickelt werden. Normalerweise ist Unity nicht kostenlos und enthält verschiedene Lizenzmodelle. Sobald Unity zur Übungs- und Lernzwecken verwendet wird, kann diese Entwicklungsumgebung kostenfrei verwendet werden. Zur Programmierung der logischen Spielabläufe wird die zuvor vorgestellte Programmiersprache C# verwendet. Die entwickelten Computerspiele können sowohl zweidimensional (2D) als auch dreidimensional (3D) sein. In der Entwicklungsumgebung Unity können bereits einfache bereitgestellte Objekte (Flächen und Körper) verwendet werden und somit die benötigte Umwelt erstellt werden. Diese Objekte können zum Beispiel einfache geometrische Figuren wie Würfel, Kugel (Ellipse), Kapsel und unter anderem Zylinder sein. Alle diese Objekte können in der Höhe, in der Breite und in der Länge skaliert werden. Ebenfalls werden vorgefertigte komplexe Objekte (engl. Assets) in großer Anzahl im Unity Asset Store angeboten. Eine Vielzahl dieser Elemente für die Spielprogrammierung ist ebenfalls kostenfrei [108]. Die nachfolgende Abbildung 2.12 zeigt die Unity Entwicklungsumgebung auf einen Blick.

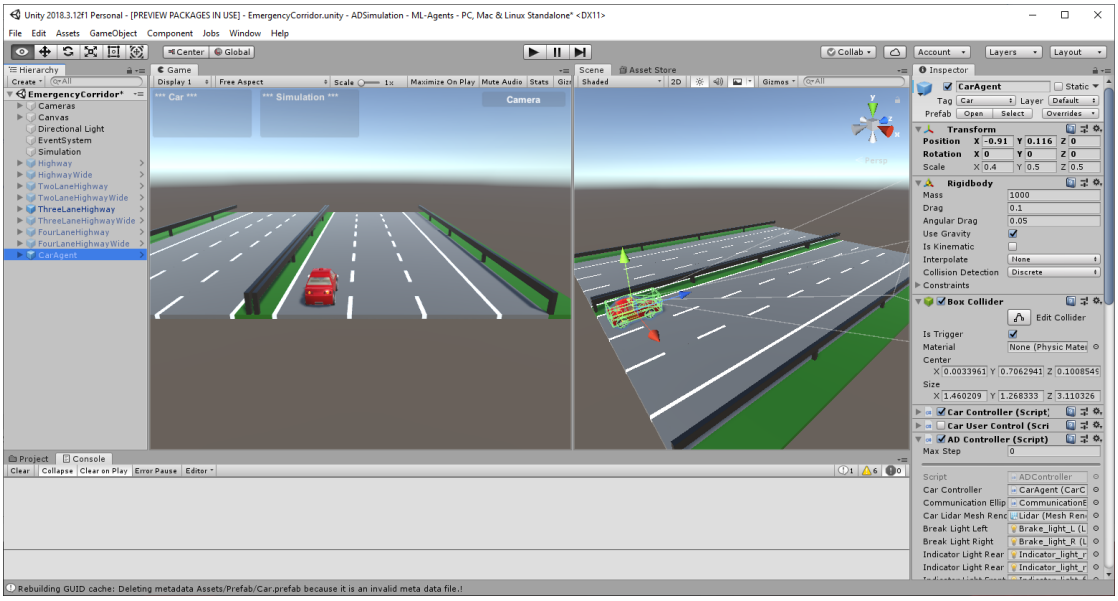


Abbildung 2.12: Darstellung der Unity Echtzeit-Entwicklungsumgebung von dem Simulator aus Kapitel 3.

In der linken Spalte findet man die Objekte des erstellten Projekts. Darauffolgend sind mittig das Spiel (engl. Game) und die Szene (engl. Scene) dargestellt. In der rechten Spalte ist der Inspektor (engl. Inspector) dargestellt. Dieser enthält zum Beispiel einige Informationen und Einstellungen über den Namen, die Position, die Rotation und die entwickelten C#-Skripe des Objekts. Sobald der Knopf zum Ausführen betätigt wird,

wird der entwickelte Programmcode ausgeführt. Die Programmlogik wiederholt sich jede Bildsequenz (engl. Frame). Der Spielablauf kann ebenfalls pausiert und eine Bildsequenz weiter abgespielt werden. In Unity können ebenfalls verschiedene Hilfswerkzeuge verwendet werden. Eine Technik zur Bestimmung des Abstands eines Objekts sind Strahlen (engl. Raycasts). Die Raycasts kann man sich als eine Linie in eine bestimmte Richtung mit einer bestimmten Länge vorstellen. Sobald ein simuliertes Objekt von dieser Linie getroffen wird, sind Informationen zu dem Namen des Objekts und die Entfernung zu diesem Objekt in einem dreidimensionalen Raum verfügbar. So lässt sich feststellen, welches Objekt getroffen wurde und wie weit es vom Ursprung des Strahls entfernt ist (Abb. 2.13).

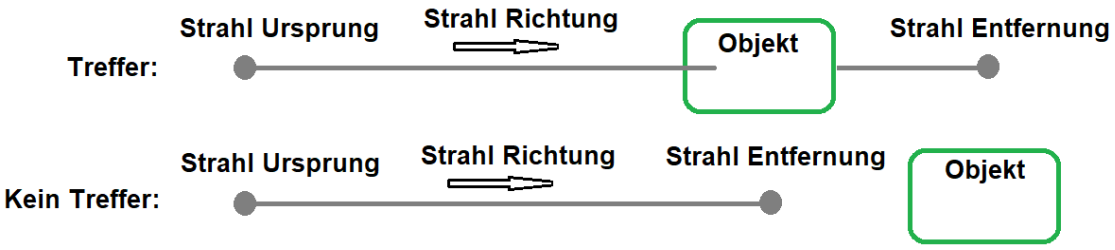


Abbildung 2.13: Darstellung von Treffer und kein Treffer eines Objekts mit den Raycasts in Unity.

2.5.4 Unity ML-Agents Toolkit

Das Unity ML-Agents (engl. Machine Learning Agents) Toolkit stellt eine Sammlung unterschiedlicher Werkzeuge für das maschinelle Lernen innerhalb einer Spiel-Engine (engl. Game-Engine) zur Verfügung. Dadurch können intelligente künstliche Agenten durch bestärkendes Lernen (engl. Reinforcement Learning) und Imitationslernen (engl. Imitation Learning) geschaffen werden. Zusätzlich bietet dieses Toolkit eine Schnittstelle zwischen der Unity Echtzeit-Entwicklungsplattform und der Programmiersprache Python. Dies ermöglicht es Python-basierte Bibliotheken einzusetzen. Diese beiden Umgebungen sind ebenfalls synchronisiert [47]. Durch ML-Agents lässt sich zum Beispiel die in Kapitel 4 vorgestellte Spurerkennung ausführen. So kann das Bild der Fahrspur in der Unity Simulation aufgenommen und an Python zur Weiterverarbeitung weitergegeben werden. Anschließend kann, durch die Auswertung in Python, der Agent in der Unity Entwicklungsumgebung synchron gesteuert werden.

2.5.5 TensorFlow

Für die Entwicklung verschiedener KI-Systeme wird in Kapitel 4 und 6 die TensorFlow-Bibliothek verwendet. TensorFlow ist ein Software-Framework für numerische Berechnungen, die auf Datenflussgraphen beruhen. Zusätzlich ist TensorFlow eine Programmierschnittstelle zum Beschreiben und Implementieren verschiedener maschineller Lernalgorithmen. Der meist verbreitete Ansatz ist die Verwendung von tiefen künstlichen

neuronalen Netzwerken (engl. Deep Learning Networks). Die eigentliche Kernkomponente von TensorFlow ist in der Programmiersprache C++ geschrieben. Allerdings ist die von den meisten Forschern und Datenanalytikern genutzte Programmierschnittstelle in Python geschrieben. Diese ist ebenfalls am weitesten entwickelt. Die in C++ entwickelte Schnittstelle bietet eine API auf einer niedrigeren Ebene. Somit lassen sich Algorithmen, zum Beispiel im Embedded-Bereich, effizienter ausführen. Dank TensorFlow können die tiefen künstlichen neuronalen Netzwerke baukastenartig aufgebaut werden. TensorFlow unterstützt ebenfalls viele Optimierungsverfahren, mit den automatisch, anhand der vom Benutzer angegebenen Verlustfunktion, differenziert werden kann. Zusätzlich können eigene Werkzeuge entwickelt oder vorhandene, wie zum Beispiel, ein Feedback-Prozess namens Backpropagation zur Optimierung der Gewichte der Deep-Learning-Netzwerke, verwendet werden [34].

2.5.6 TensorFlow Object Detection API

Die TensorFlow Objekterkennungsprogrammierschnittstelle (engl. TensorFlow Object Detection API) ist ein Open-Source-Framework, welches auf der TensorFlow-Bibliothek aufgebaut ist [122]. Dieses Framework erleichtert die Erstellung, das Training und den Einsatz von Objekterkennungsmodellen. Diese Modelle für maschinelles Lernen sind in der Lage, mehrere Objekte in einem einzigen Bild präzise zu lokalisieren und zu klassifizieren. Allerdings ist dieses Thema eine große Herausforderung in der Computer Vision. Die TensorFlow Object Detection API unterstützt sowohl TensorFlow 1 (TF 1) als auch TensorFlow 2 (TF 2). Ebenfalls sind bereits vortrainierte Modelle verfügbar. Diese Modelle können zusätzlich angepasst und erweitert werden. In dem TensorFlow 1 Erkennungsmodellzoo (engl. Detection Model Zoo) sind mehrere dieser ConvNet-Modelle für TensorFlow 1 zu finden [92]. In dem TensorFlow 2 Erkennungsmodellzoo stehen ebenfalls einige zur Verfügung [18]. Die TensorFlow 2 Object Detection API Modelle können zusätzlich in TensorFlow Lite konvertiert werden [105]. Zum Zeitpunkt der Entwicklung werden nur die SSD-Modelle unterstützt. Ebenso sind die in TensorFlow 1 Modellzoo und TensorFlow 2 Modellzoo zur Verfügung gestellten Modelle nur für die TensorFlow 1 oder TensorFlow 2 Hauptversion entwickelt. Diese verschiedenen TensorFlow 1 und TensorFlow 2 Modelle sind nicht interoperabel [38]. Die TensorFlow Object Detection API wurde für die Objekterkennung in dem Kapitel 6 verwendet.

2.5.7 OpenCV

OpenCV (engl. Open Source Computer Vision) ist eine sehr häufig verwendete native und plattformübergreifende Bibliothek, um verschiedene Probleme in dem maschinellen Sehen zu bewältigen. Diese enthält verschiedene Implementierungen von Algorithmen zu Computersehen, maschinellem Lernen und Bildverarbeitung. Hinter den oberflächlichen OpenCV-Funktionen verbergen sich sehr leistungsfähige interne Funktionen, die auf Recheneffizienz ausgelegt sind. Zusätzlich können diese Algorithmen auf mehreren Prozessoren oder Grafikkarten ausgeführt werden. Außerdem können verschiedene Methoden zur Bildgeometrie genutzt werden, um verschiedene Ansichten einer abgebildeten Szene

zu vermitteln [94].

2.6 Verwendete Hardware

Dieser Abschnitt präsentiert die verwendete Hardware dieser Forschungsarbeit. Für die Entwicklung und die Experimente wurde ausschließlich die in den nächsten Abschnitten aufgelistete Hardware verwendet.

2.6.1 Dell G3 15 3590 Notebook

Für die Entwicklung und einige Experimente dieser Forschungsarbeit in Abschnitt 4.6 und Abschnitt 6.5.2 wurde das Dell G3 15 3590 Notebook verwendet. Dieses hat einen Intel i7-9750H Prozessor (engl. Central Processing Unit - CPU), 16 GB Arbeitsspeicher (engl. Random Access Memory - RAM), 256 GB SSD Festplatte und eine NVIDIA GeForce GTX 1660 Ti Grafikkarte (engl. Graphics Processing Unit - GPU) verbaut. Diese Hardware war für die Entwicklung mit Unity und Python vollkommen ausreichend.

2.6.2 Raspberry Pi

In Kapitel 6 und 7 wird das Thema der Entwicklung der Software für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte eingeführt. Dafür wurden der Raspberry Pi 3 B und der Raspberry Pi 4 B als IoT-Geräte mit beschränkten Ressourcen verwendet. Diese Einplatinencomputer, die von der britischen Raspberry Pi Foundation aus den Komponenten von Android-Smartphones entwickelt wurden, haben lediglich ein leistungsschwaches Ein-Chip-System (engl. System-on-a-Chip - SoC) von Broadcom mit einer Arm-CPU verbaut. Dieser Prozessor ist mit einem leistungsschwachen Grafikprozessor kombiniert [51]. Diese Hardwarekomponenten sind auf dem Board fest verlötet und können nicht ausgetauscht werden. Ebenfalls können diese Einplatinencomputer eine grafische Benutzeroberfläche darstellen und Filme in einer Full-HD-Auflösung mit 30 Bildern pro Sekunde (engl. Frames per Second - FPS) darstellen. Ebenfalls durch das Verbauen der Raspberry Pi Kamera kann der Raspberry Pi zum Beispiel als Überwachungskamera, mobile Actionkamera oder für Langzeitaufnahmen genutzt werden. Außerdem gibt es inzwischen mehrere Modellgenerationen, die weitestgehend kompatibel sind. Für den Raspberry Pi gibt es verschiedene Betriebssysteme. Das beliebteste und auf die Leistungsfähigkeit des Raspberry Pis zugeschnittene Version ist Raspbian. Der Name dieser Linux-Distribution setzt sich aus den Begriffen *Raspberry* und *Debian* zusammen [30]. Zusätzlich wird der Raspberry Pi als ein IoT-Gerät für verschiedene Anwendungen eingesetzt. Dieser kann zum Beispiel als eine Wetterstation, eine Steuerungsanlage für Heizungen und Solaranlagen oder eine Heimautomation fungieren. Ebenfalls ist der Einsatz des Raspberry Pis als intelligentes Spielzeug-Fahrzeug oder Spielzeug-Roboter sehr beliebt. Durch die Linux-Distribution können verschiedene Programmiersprachen auf diesem Minicomputer ausgeführt werden [51]. Eine der in dieser Forschungsarbeit verwendeten Programmiersprachen ist Python. In Kapitel 6 und 7 werden einige Lauf-

zeitmessungen der in dieser Forschungsarbeit entwickelten Systeme auf den Raspberry Pis durchgeführt. Die Hardware der verwendeten IoT-Geräte sieht wie folgt aus:

- Raspberry Pi 3 B mit ARM Cortex-A53 1,2 GHz Prozessor (CPU), 1 GB Arbeitsspeicher (RAM), USB 2.0, 8 GB SD Speicherkarte mit Raspbian.
- Raspberry Pi 4 B mit ARM Cortex-A72 1,5 GHz Prozessor (CPU), 8 GB Arbeitsspeicher (RAM), USB 3.0, 16 GB SD Speicherkarte mit Raspbian.

2.6.3 Intel Neural Compute Stick 2

Die Laufzeitmessungen in Abschnitt 6.5.4 wurden auf der Intel Neural Compute Stick 2 (NCS2) Hardwareerweiterung durchgeführt [43]. Diese Hardwareerweiterung hat eine Intel Movidius Myriad X Bildverarbeitungseinheit (engl. Vision Processing Unit - VPU) mit 700 MHz Grundtaktfrequenz verbaut. Dieser Mikroprozessor wurde speziell Entwickelt, um die Ausführung der Anwendungen mit künstlichen neuronalen Netzwerken zu beschleunigen. Dabei liegt der Schwerpunkt auf der Hardware mit geringem Stromverbrauch und der Entwicklung der Bildverarbeitung in Echtzeit [42]. Durch die implementierte Software, können auf dieser Hardwareerweiterung die OpenVINO-Modelle verarbeitet werden. Die Intel NCS2 Hardwareerweiterung enthält ebenfalls 4 GB Arbeitsspeicher (engl. Synchronous Dynamic Random Access Memory - SDRAM) und die OpenVINO-Modelle ausführen zu können [41].

2.6.4 Google Colab

Für das Training der ConvNet-Modelle in Kapitel 6 hat die Hardware des Dell G3 15 3590 Notebooks nicht mehr ausgereicht. Aus diesem Grund wurden die Dienste von Google Colaboratory (Google Colab) verwendet [31]. In Google Colab wird ein virtueller Computer jedem Google-Benutzer zugewiesen. Dadurch kann ein eigener Python-Code in einem Browser durch die Verwendung von IPython geschrieben und ausgeführt werden. Zusätzlich ist keine Konfiguration dieser virtuellen Computer notwendig, was ein sehr großer Vorteil ist. Ebenfalls können verschiedene Dateien hochgeladen und verschiedene Bibliotheken auf diesem virtuellen Computer installiert werden. Zusätzlich ist die Nutzung von Google Colab kostenlos. Dazu ist lediglich ein Google-Account notwendig. Für das Training der ConvNet-Modelle in Kapitel 6 wurde ein virtueller Computer mit einem Intel Xeon 2.30 GHz Prozessor (CPU), 26 GB Arbeitsspeicher (RAM) und einer NVIDIA Tesla P100-PCIE-16GB Grafikkarte (GPU) verwendet.

Nichts ist für mich mehr Abbild der Welt und des Lebens als der Baum. Vor ihm würde ich täglich nachdenken, vor ihm und über ihn...

Christian Morgenstern
1871 – 1914

Kapitel 3

Entwicklung eines Simulators in Unity

3.1 Einführung

Auf den Straßen und Autobahnen passieren immer wieder unvorhersehbare Unfälle. Auf den Autobahnen kann es teilweise zu schwerwiegenden Unfällen kommen, da die Fahrzeuge sich auf den Autobahnen mit hoher Geschwindigkeit bewegen. Dadurch kommt es zu Staus und Straßensperrungen. Für diesen Fall gibt es eine Regelung der Rettungsgassenbildung für die zum Unfallort ankommenden Einsatzfahrzeuge. Doch das ist leider nur die Theorie. Die Fahrer der Kraftfahrzeuge vergessen teilweise in einem Stau eine Rettungsgasse für die Polizei- und Rettungsfahrzeuge zu bilden. Eine Bildung der Rettungsgasse zu einem späteren Zeitpunkt ist in einigen Fällen nicht mehr möglich. Wenn die Fahrzeuge zum Stehen kommen, sind diese in der Regel zu nah bei einander. Zusätzlich ergibt sich das Problem, dass selbst wenn nach dem Stehen der Fahrzeuge eine Rettungsgasse gebildet werden kann und ein Einsatzfahrzeug vorbeifährt, die Fahrer der Kraftfahrzeuge diese Rettungsgasse wieder schließen. Dies führt immer wieder zu Behinderung weiterer Polizei- beziehungsweise Rettungsfahrzeuge. Das Problem ist nicht neu und wird schon seit längerem diskutiert. Dieses ergibt sich nicht nur auf den Autobahnen, sondern auch in den Städten. Aktuell gibt es keine optimale Lösung dafür. Manche Länder greifen mit hohen Geldstrafen für die Behinderung der Einsatzfahrzeuge durch [12]. Um diesem Problem entgegenzuwirken wurde bereits ein Warnsystem für Rettungsfahrzeuge (engl. Emergency Vehicle Warning System) vorgestellt [9]. Um dieses Problem komplett zu beseitigen muss die Rettungsgasse automatisch von den bald für den Straßenverkehr zugelassenen autonomen Fahrzeugen gebildet werden. Denn die Frage ist nicht ob, sondern wann die autonomen Fahrzeuge auf unsere Straßen kommen.

Zusätzlich bemüht sich die Automobilindustrie für autonome Fahrzeuge jeden Tag, die Sicherheit dieser Fahrzeuge zu verbessern und somit die durch die Fahrzeuge verursachten

Unfälle so weit wie möglich zu reduzieren. Dennoch können selbst bei autonomen Fahrzeugen die Unfälle auf unvorhersehbare technische Mängel oder Softwarefehler zurückgeführt werden. Um die Algorithmen für die Rettungsgassenbildung für die autonomen Fahrzeuge entwerfen und testen zu können, ist zuerst ein geeigneter Simulator von Nöten. Somit wird in diesem Kapitel die Entwicklung eines Simulators in Unity vorgestellt. Dieser Simulator wurde erstellt, um das Verhalten autonomer Fahrzeuge bei Unfällen auf einer Autobahn zu simulieren und zu analysieren. Dieses Vorgehen ermöglicht eine komplett freie Implementierung des Verhaltens der Fahrzeuge gegenüber einem Verkehrsflusssimulator. Zum Zeitpunkt der Entwicklung gibt es in dem Bereich der Rettungsbildung für autonome Fahrzeuge auf Autobahnen noch keine wissenschaftliche akademische Forschung.

In den nachfolgenden Abschnitten werden die verwandten Arbeiten zu den Verkehrsflusssimulatoren, den verfügbaren Projekten in Unity in dem Bereich des autonomen Fahrens und der Rettungsgassenbildung für autonome Fahrzeuge vorgestellt. Die Simulation wird inklusive der Umwelt und Objekte, die Erkennung der Fahrspur, die Erkennung des Abstands zum nachfolgenden Fahrzeug und die Erkennung der Drehung des Fahrzeugs im Simulator vorgestellt. Da die echten autonomen Fahrzeuge ebenfalls während der Fahrt kommunizieren, werden zwei Techniken der Fahrzeug-zu-Fahrzeug Kommunikation in Unity eingeführt. Zusätzlich werden in diesem Kapitel zwei Algorithmen für die Bildung der Rettungsgasse auf einer Autobahn für autonome Fahrzeuge entwickelt und vorgestellt. Diese Algorithmen werden anschließend in dem entwickelten Simulator, durch die Durchführung einiger Experimente, getestet. Durch die Entwicklung des Simulators für die Bildung der Rettungsgasse mit autonomen Fahrzeugen können die Algorithmen ohne einen aufwändigen experimentellen Aufbau entwickelt, getestet und verbessert werden. Der implementierte Simulator und die entwickelten Algorithmen sollen dazu beitragen, die derzeitigen Probleme bei der Bildung einer Rettungsgasse für Polizei- und Rettungsfahrzeuge auf Autobahnen durch die kommenden autonomen Fahrzeuge zu beseitigen. Oft können die Rettungsfahrzeuge die Unfallstelle nicht erreichen und werden durch andere Verkehrsteilnehmer behindert. So kann die Hilfe für die Unfallbeteiligten unter Umständen zu spät kommen.

3.2 Verwandte Arbeiten

Dieser Abschnitt präsentiert die verwandten Arbeiten zu dem in Unity entwickelten Simulator und den entwickelten Algorithmen für die Bildung der Rettungsgasse mit autonomen Fahrzeugen auf Autobahnen. Bevor mit der Implementierung des Simulators begonnen wurde, wurde nach einem etablierten Verkehrsflusssimulator gesucht. Einige Verkehrsflusssimulatoren sind bereits verfügbar: *PTV Vissim* [87], *MATSim* [68], *SUMO* [6], *MAINSIM* [20]. Leider kann das Verhalten des Fahrzeugs nicht beliebig programmiert werden. Außerdem kann die Position des Fahrzeugs nicht frei verändert und somit die Fahrzeuge nicht beliebig gesteuert werden. Aus diesem Grund wurde mit der Spielentwicklungssoftware Unity ein eigener Simulator für autonome Fahrzeuge für die Bildung der Rettungsgasse für die Polizei- und Rettungsfahrzeuge auf Autobahnen implementiert. Eine eigene Implementierung erlaubt es das Verhalten des autonomen Fahrzeugs und

die Kommunikation zwischen den autonomen Fahrzeugen zu entwickeln. Somit kann die Erkennung der Fahrspur, Erkennung des Abstands zwischen den autonomen Fahrzeugen, Erkennung der Drehung der Fahrzeuge und Erkennung der Unfälle im Vergleich zu einem Verkehrsflusssimulator implementiert werden. Ebenfalls kann die Autobahn auch beliebig gestaltet werden. Zum Beispiel können mehrere Fahrspuren hinzugefügt werden, die in ihrer Breite variieren können.

Zusätzlich gibt es bereits mehrere Projekte in Unity, die sich mit der Simulation von autonomen Fahrzeugen beschäftigen. So können beispielsweise Straßen mit Fahrzeugen, Fußgängern und autonomen Fahrzeugen mit Sensoren mit *SimViz* erstellt werden [76]. Einige Projekte beinhalten sogar selbstfahrende Fahrzeuge inklusive der Bilderkennung der simulierten Umgebung. Zum Beispiel kann *AirSim*, ein Open-Source-Simulator mit realistischen Umgebungen und Fahrzeugdynamiken für autonome Systeme, verwendet werden [77]. Neben den Unity Projekten gibt es ebenfalls *CARLA*, einen Open-Source-Simulator für die Forschung zum autonomen Fahren. Dieser Simulator wurde von Grund auf entwickelt, um die Entwicklung, das Training und die Validierung von autonomen städtischen Fahrsystemen zu unterstützen. Neben dem offenen und verfügbaren Quellcode bietet dieser Simulator digitale Assets wie Stadtgrundrisse, Gebäude und Fahrzeuge, die zu dem Zweck der Forschung erstellt wurden und kostenlos verwendet werden können [22].

Der Fokus dieser Forschungsarbeit bei der Entwicklung eines Simulators liegt bei der Bildung einer Rettungsgasse mit autonomen Fahrzeugen auf Autobahnen. Daher wird die Möglichkeit der Erstellung von verschiedenen Autobahnen mit Leitplanken links und rechts benötigt. Zusätzlich ist die Erstellung von mehreren Fahrspuren auf einer Autobahn, die in der Breite variieren, von Nöten. Aus diesem Grund wird ein eigener Simulator in dieser Forschungsarbeit implementiert. Laut der Recherchen gab es zum Zeitpunkt der Entwicklung noch keine veröffentlichten Forschungsarbeiten auf dem Gebiet der automatischen Bildung einer Rettungsgasse mit autonomen Fahrzeugen. Es gab lediglich mehrere Patente [19], [66], [93] für Warn- und Kommunikationssysteme für die Bildung der Rettungsgasse durch die Fahrzeug-zu-Fahrzeug Kommunikation. Dadurch können zum Beispiel die Informationen über die Route und das Einsatzfahrzeug an die anderen Verkehrsteilnehmer weitergeleitet werden.

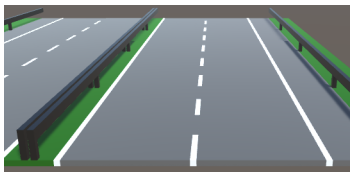
3.3 Simulation in Unity

Ausgehend davon, wie die autonomen Fahrzeuge in der Praxis funktionieren, konnten die Anforderungen an eine geeignete Implementierung des Simulators abgeleitet werden. Mehr Informationen gibt der Abschnitt 2.4. Die nachfolgenden Abschnitte beschreiben somit die entwickelte Umwelt inklusive der Objekte, des Lenkverhaltens, der Spurerkennung, der Rotationserkennung und der Entfernungsmessung der autonomen Fahrzeuge in dem Simulator. Ebenfalls wird auf die Kommunikation zwischen den simulierten autonomen Fahrzeugen eingegangen und zwei verschiedene Methoden in Unity vorgestellt.

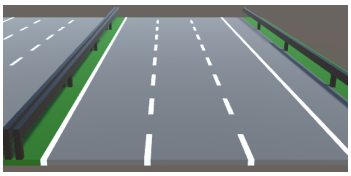
3.3.1 Umwelt und Objekte

In der Entwicklungsumgebung Unity können bereits einfache bereitgestellte Objekte verwendet werden und somit die benötigte Umwelt erstellt werden. Mehr dazu beschreibt der Abschnitt 2.5.3. Für die Umwelt mit verschiedenen Autobahnen wurden Würfelobjekte, die anschließend zu Quadern skaliert wurden, verwendet. Somit konnten die verschiedenen Quader an verschiedene Positionen in der simulierten Umgebung gesetzt werden. Die Farbe dieser Objekte wurde ebenfalls verändert, je nach dem zu welcher Kategorie dieses Objekt gehört (Abb. 3.1).

(a) Zweispurige reguläre Autobahn.



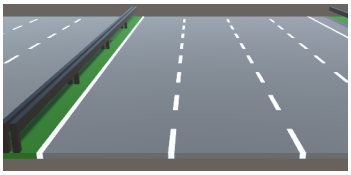
(b) Zweispurige extra breite Autobahn.



(c) Dreispurige reguläre Autobahn.

(d) Dreispurige extra breite Autobahn.

(e) Vierspurige reguläre Autobahn.



(f) Vierspurige extra breite Autobahn.

Abbildung 3.1: Simulierte Autobahn mit verschiedener Anzahl und Breite der einzelnen Fahrspuren.

So ergeben sich die schwarzen Leitplanken, die grauen Fahrbahnen, die weißen Fahrbahnmarkierungen und die grünen Wiesen. Die Farben der Objekte können in Unity ebenfalls durch Texturen ausgetauscht werden, falls dies notwendig ist. Für die in Abschnitt 3.5 durchgeführten Experimente wurden Autobahnen mit verschiedener Anzahl der Fahrspuren und verschiedener Spurbreite entwickelt. Somit kann anschließend die Bildung der Rettungsgasse durch autonome Fahrzeuge im Simulator auf einer zweispurigen Autobahn (Abb. 3.1a), auf einer dreispurigen Autobahn (Abb. 3.1c) und auf einer vierspurigen Autobahn (Abb. 3.1e) durchgeführt werden. Diese Autobahnen sind ebenfalls in einer breiten Spurvariante verfügbar (Abb. 3.1b, 3.1d und 3.1f). Um den Implementierungsaufwand geringer zu halten, wurde die Vorlage für die Spielprogrammierung *Car'Toon : The Sport Car with interior* aus dem Unity Asset Store verwendet [111]. Diese enthält bereits die Implementierungen für die manuelle Lenkung des Fahrzeugs, die automatische Beschleunigung, den Bremsvorgang und die Physik eines Fahrzeugs. So konnte diese Vorlage erfolgreich als autonomes simuliertes Fahrzeug eingesetzt werden. Die Farbe des Fahrzeugs ist standardmäßig rot (Abb. 3.2). Die Geschwindigkeit eines Objekts ist in Unity ebenfalls angegeben. Diese ist in Unity Einheit pro Sekunde (ue/s) angegeben

[102]. In dieser Arbeit entspricht eine Unity Einheit zwei Metern. Somit kann die Geschwindigkeit des simulierten Fahrzeugs in Meter pro Sekunde (m/s) beziehungsweise Kilometer pro Stunde (km/h) gemessen werden.

(a) Heckansicht. (b) Seitenansicht. (c) Frontansicht.

Abbildung 3.2: Autonomes simuliertes Fahrzeug aus der Vorlage *Car'Toon : The Sport Car with interior* aus dem Unity Asset Store [111].

Zusätzlich wurde die maximale Geschwindigkeit des simulierten Fahrzeugs auf 30 ue/s beziehungsweise 60 m/s oder 216 km/h gesetzt. Bei der Erzeugung verschiedener Umwelten wurde aus einer Farbpalette die Farbe für das Fahrzeug in der Simulation zufällig zwischen weiß, schwarz, grau und rot gewählt. Bei der Umsetzung der automatischen Rettungsgassenbildung ist aufgefallen, dass sich das verwendete Auto bei einer Notbremsung nicht lenken ließ, da alle vier Räder des Autos beim Bremsen komplett blockierten. Um eine Rettungsgasse bilden zu können, muss bei einer Notbremsung nicht nur gebremst, sondern auch gleichzeitig gelenkt werden. In der Praxis ist dies nur mit einem Antiblockiersystem (engl. Anti-Lock Braking System - ABS) möglich. Daher wurde dieses System ebenfalls in diesem Simulator eigenhändig implementiert. Ein Mensch kann in einer Gefahrsituation nicht gleichzeitig bremsen und an die Bildung der Rettungsgasse denken. Zum Vergleich: Das autonome Fahrzeug ist dazu in der Lage, da es nicht nervös und hektisch gegenüber dem Menschen reagiert und nur die entwickelten Algorithmen Schritt für Schritt ausführt. Um die in Abschnitt 3.4 vorgestellten Algorithmen für die Bildung der Rettungsgasse zu überprüfen, wurden ebenfalls einige simulierte Tiere aus dem Bauernhoftiere-Sortiment *Farm Animals Set* aus dem Unity Asset Store verwendet [112].

(a) Linke Seitenansicht: Kuh, Schwein und Schaf. (b) Rechte Seitenansicht: Kuh, Schwein und Schaf. (c) Herde verschiedener Tiere (Wildwechsel).

Abbildung 3.3: Simulierte Tiere auf einer Autobahn aus der Vorlage *Farm Animals Set* aus dem Unity Asset Store [112].

Dieses Sortiment enthält mehrere modellierte Bauernhoftiere wie Enten, Hühner, Kühe, Schweine und Schafe. In der Simulation wurden allerdings nur die Vorlagen für Kühe, Schweine und Schafe genutzt. Die vorherige Abbildung 3.3 veranschaulicht die Bauernhoftiere aus dem Unity Asset Store. Nachdem die Objekte erfolgreich modelliert und einige vorgefertigte Modelle hinzugefügt werden konnten, konnte eine Autobahn mit generierten autonomen Fahrzeugen inklusive den Hindernissen durch verschiedene Tiere auf der Fahrbahn erstellt werden. Dies stellt die Abbildung 3.4 bildlich dar. Die verschiedenen simulierten Tiere stellen einen beispielhaften unerwarteten Wildwechsel auf einer Autobahn vor. Dies ist nur eine Möglichkeit von Hindernissen auf einer Autobahn. Ein Unfall mit mehreren Fahrzeugen oder ein ausgebrochenes Lastkraftfahrzeug kann ebenfalls ein Hindernis auf einer Autobahn sein.

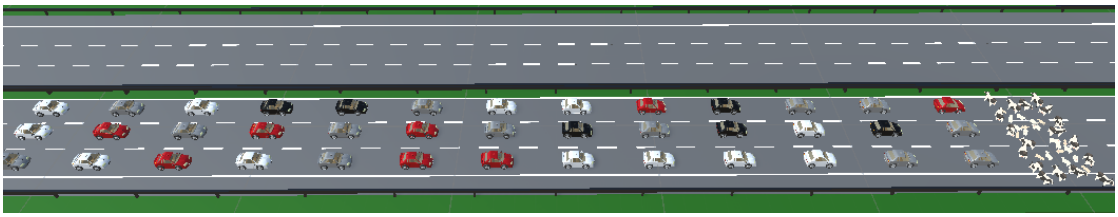


Abbildung 3.4: Dreispurige Autobahn in der Simulation mit 39 generierten autonomen Fahrzeugen und Hindernissen durch verschiedene Tiere auf der Fahrbahn.

3.3.2 Lenkverhalten, Spur-, Entfernungs- und Rotationserkennung

In diesem Abschnitt geht es um das Lenkverhalten, die Erkennung der Fahrspur, die Erkennung des Abstands zum nachfolgenden Fahrzeug und die Erkennung der Drehung des Fahrzeugs im vorgestellten Simulator. Wie bereits erwähnt ist in dem verwendeten Fahrzeug aus dem Unity Asset Store eine manuelle Lenkung des Fahrzeugs implementiert. Somit kann das Fahrzeug von dem Benutzer von Hand gesteuert werden. Allerdings wird an dieser Stelle die automatische Lenkung für autonome Fahrzeuge benötigt. Zur automatischen Steuerung des Lenkverhaltens wurde in dem Simulator eine Trajektorie für die autonomen Fahrzeuge entwickelt und getestet. Diese Trajektorie wurde in diesem Fall durch einfache Ansätze der Funktionsapproximation bestimmt. Dabei wurden nur die Einflüsse von der Geschwindigkeit betrachtet und das Lenk- und Störverhalten bei Seitenwind ausgelassen. Zusätzlich wird hierbei von trockenen Straßen ausgegangen. Um genügend Daten für die Berechnung der Trajektorie zu beschaffen, wurde das Fahrzeug manuell in dem Simulator bei verschiedenen Geschwindigkeiten gesteuert. Anschließend konnte aus diesen Daten eine Funktion 3.1 für das Lenkverhalten (lv) anhand der Fahrgeschwindigkeit (g) durch nicht lineare Regression für $0 \leq g \leq 30$ gefunden werden:

$$lv(g) = \begin{cases} -0.000006g^3 + 0.0004g^2 - 0.0109g + 0.175, & g \in [0, 30] \\ 0.005, & \text{sonst} \end{cases} \tag{3.1}$$

Durch diese Funktion ist sichergestellt, dass das simulierte autonome Fahrzeug bei einer hohen Geschwindigkeit auf einer Autobahn nicht stark lenken kann. Somit wird die Kipp- und Schleudergefahr des Fahrzeugs vermieden. Durch diese Approximation kann das physikalische Verhalten des Fahrzeugs überwacht und gesteuert werden. Ein höherer Wert für die Geschwindigkeit ergibt einen kleineren Wert bei der Lenkung des Fahrzeugs. Ebenfalls ist bei der Berechnung der Lenkung die Geschwindigkeit auf die Werte zwischen 0 und 30 ue/s begrenzt.

Die Erkennung der Fahrspur wurde im ersten Schritt der Einfachheit halber anders implementiert als aus der Praxis bekannt (Spurerkennung durch eine Videokamera). Um die Breite der Fahrspur zu bestimmen und das simulierte Fahrzeug in der Spur zu halten, wurde auf die Technik der Raycasts in Unity zurückgegriffen. Der Abschnitt 2.5.3 gibt mehr Informationen zu den Raycasts. Diese Technik ersetzt an dieser Stelle die bildliche Spurerkennung mit einer Videokamera. Für die Erkennung der Fahrspur wurden zwei Raycasts verwendet (Abb. 3.5a). Beide sind von der Mitte des Fahrzeugs nach vorne und leicht nach unten gerichtet (Abb. 3.5b). Mit Raycasts lässt sich so nicht nur die Fahrspur erkennen, sondern es können auch die Abstandssensoren (Radar) simuliert werden. Wie bereits erwähnt, ermittelt der Radarsensor den Abstand zum vorderen Fahrzeug in der Praxis. Die verschiedenen Sensoren aus der Praxis werden in dem Abschnitt 2.4.2 ausführlicher beschrieben. Die in Abbildung 3.5c dargestellten neun nach vorne gerichteten Strahlen sind notwendig, um den Abstand zum nachfolgenden simulierten Fahrzeug zu bestimmen.

(a) Draufsicht der Raycasts auf der dritten Spur.

(b) Seitenansicht der Raycasts.

(c) Alle Raycasts auf der Fahrbahn.

Abbildung 3.5: Raycasts eines autonomen Fahrzeugs zur Fahrspur- und Abstandserkennung in dem Simulator. Zwei für die Fahrspurerkennung und neun für die Abstandserkennung.

Die implementierte Rotationserkennung des eigenen Fahrzeugs beinhaltet die Überwachung der Rotation pro Spielsequenz. Diese Implementierung simuliert den Sensor zur Rotationsbestimmung aus der Praxis. Sobald eine große Drehung in einer kurzen Zeit stattgefunden hat, kann von einer unerwarteten Drehung des Fahrzeugs ausgegangen

werden. Diese Rotationserkennung hilft zum Beispiel, das Platzen eines Reifens während der Fahrt zu simulieren. Mehr zu diesem Experiment wird in Abschnitt 3.5.3 vorgestellt. Neben der Drehung wird auch die Kollision des Fahrzeugs überwacht. Diese Methode wird von Unity bereitgestellt und wird automatisch aufgerufen, wenn ein Spielobjekt mit einem anderen Spielobjekt kollidiert. Sobald einer dieser beiden Unfälle erkannt wird, führt das simulierte autonome Fahrzeug eine Notbremsung durch, schaltet die Warnblinkleuchten ein und benachrichtigt die simulierten autonomen Fahrzeuge in der Umgebung. Dieses Vorgehen wird im nachfolgenden Abschnitt 3.3.3 ausführlicher erläutert. Sobald der Abstand durch die neun Raycasts zum nachfolgenden Fahrzeug gemessen werden kann, errechnet eine Funktion (Gl. 3.3) auf Basis der Fahrgeschwindigkeit den Sicherheitsabstand, der eingehalten werden muss, um eine Kollision zu vermeiden. Auf der Grundlage eigener Vorversuche konnte für diese Simulation ebenfalls eine effektive Funktion (Gl. 3.2) für den Bremsweg durch nicht lineare Regression gefunden werden. Um an die notwendigen Daten für die nicht lineare Regression zu kommen, wurden in der Simulation mehrere Notbremsungen aus unterschiedlichen Fahrgeschwindigkeiten durchgeführt. Somit ergibt sich für $0 \leq g \leq 30$ die folgende Funktion:

$$bw(g) = 0.0003g^3 + 0.0009g^2 + 0.1587g, \quad g \in [0, 30] \tag{3.2}$$

Diese Funktion liefert den Bremsweg (bw) in Abhängigkeit von der Fahrgeschwindigkeit (g) des Fahrzeugs. Dabei handelt es sich jedoch nur um den Bremsweg, der auf jeden Fall eingehalten werden muss. In der endgültig implementierten Funktion (Gl. 3.3) wurde zum Bremsweg (bw) ein linearer Wert (a) für den Sicherheitsabstand (sa) hinzugefügt. Somit ergibt sich die folgende Funktion für den Sicherheitsabstand (sa):

$$sa(g) = bw(g) + a \tag{3.3}$$

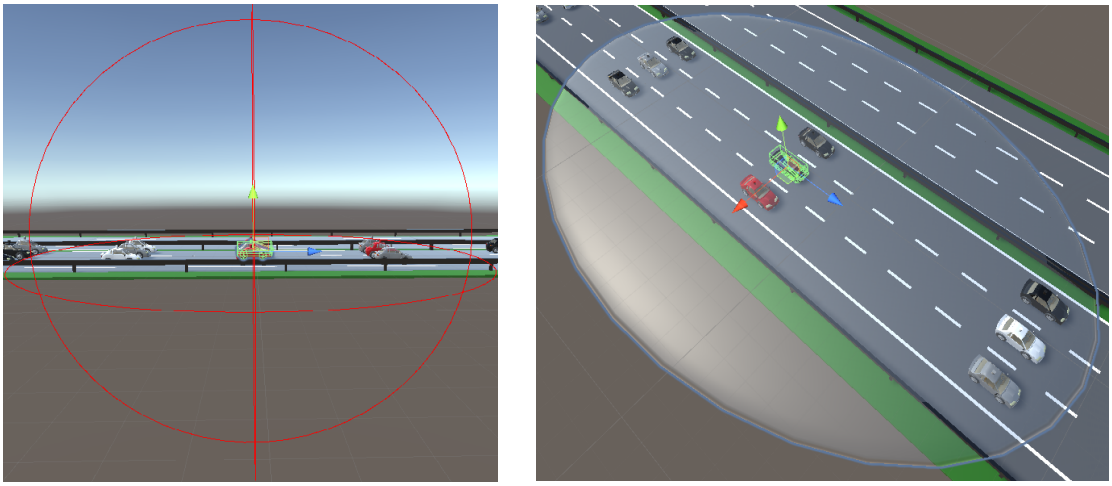
Bei dem Skalar (a) handelt es sich um einen Wert von $1,6$ Unity Einheiten. Dieser entspricht der Länge des simulierten Fahrzeugs in Unity. Somit ist nach der Gefahrenbremsung der Abstand des Fahrzeugs zum vorderen Objekt genau eine Fahrzeuglänge.

3.3.3 Kommunikation zwischen simulierten Fahrzeugen

Die Kommunikation bei autonomen Fahrzeugen ist sehr wichtig. Was für uns Menschen die Warnung durch Warnblinklichter oder der Bremslichter ist, ist für das autonome Fahrzeug die elektronische Kommunikation. Wichtig hierbei ist, dass die Kommunikation von Fahrzeug zu Fahrzeug stattfindet. Das heißt dies geschieht ohne eine zentrale Stelle, wie zum Beispiel einem Masten, denn dieser kann ausfallen oder manipuliert werden. Durch die Kommunikation der autonomen Fahrzeuge kann die Koordination der Fahrzeuge realisiert werden. Wenn zum Beispiel alle Fahrzeuge unerwartet stehen bleiben müssen, dann stellt sich die Frage, wer was als erster macht. Hierbei ist es ebenfalls schwierig die Regeln abzubilden. Wie Verhalten sich die Fahrzeuge bei einem Unfall durch zum Beispiel einen technischen Defekt? Wer fährt vorbei? Wer hält an? Sogar für einen Menschen ist es in

manchen Situationen schwer zu entscheiden, wie man sich in gewissen Situationen richtig verhält. Durch die Kommunikation wird ebenfalls herausgefunden, welches Fahrzeug sich auf welcher Fahrspur befindet und wie viele Spuren die Autobahn insgesamt hat. Die Fahrzeuge kennen ihre eigene Fahrspur, Fahrtrichtung und die maximalen Fahrspuren der Autobahn. Dies wird an die Fahrzeuge in der Umgebung kommuniziert. Durch eindeutige Identifikationsnummern (IDs) wissen diese Fahrzeuge ebenfalls, welches Fahrzeug sich vor ihnen, hinter ihnen, links und rechts befindet.

Die Kommunikation zwischen Spielobjekten kann in Unity auf zwei verschiedenen Wegen implementiert werden. Beide Vorgehen wurden für die Kommunikation zwischen den simulierten autonomen Fahrzeugen ausprobiert. Als erstes wurde die in Unity zur Verfügung gestellte Methode `Physics.OverlapSphere(center, radius)` ausprobiert. Dadurch wird eine Kugel mit einem übergebenen Zentrum und einem übergebenen Radius erstellt. Diese Methode liefert anschließend die Objekte, die sich in der erstellten Kugel befinden. Dadurch kann schnell ermittelt werden, welche Fahrzeuge sich in der Nähe beziehungsweise in der Umgebung befinden. An dieser Stelle wird auch das eigene Fahrzeug mitgeliefert. Dieses muss dann natürlich herausgefiltert werden. Die nachfolgende Abbildung 3.6a zeigt die `OverlapSphere`-Methode im Einsatz.



(a) Die Kommunikation der Fahrzeuge durch eine erzeugte Kugel.

(b) Die Kommunikation der Fahrzeuge durch eine erzeugte Ellipse.

Abbildung 3.6: Kommunikation zwischen autonomen Fahrzeugen im Unity Simulator.

Das Zentrum dieser Kugel wurde auf das Zentrum des eigenen Fahrzeugs gesetzt. Auf der vorherigen Abbildung 3.6a kann ebenfalls erkannt werden, dass die Kugel auch die Objekte oberhalb und unterhalb des Fahrzeugs erkennen würde. Das Versenden der Nachrichten nach oben beziehungsweise nach unten ist in diesem Fall nicht notwendig. Dies entspricht auch nicht ganz der Realität, da man in der Praxis das Signal nach oben beziehungsweise nach unten nicht verschwenden würde. Das Signal könnte zum Beispiel durch Richtantennen in bestimmte Richtungen gelenkt werden. Dadurch können bei gleicher Sendeleistung größere Distanzen überwunden werden. Aus diesem

Grund wurde eine skalierte Kugel (Ellipse) für die Kommunikation implementiert. Die vorherige Abbildung 3.6b stellt diese Ellipse grafisch dar. Die Kommunikationsellipse ist ein Spielobjekt, das dem Fahrzeug als Unterobjekt zugeordnet ist. Dieses enthält die Methoden `OnTriggerEnter(Collider)` und `OnTriggerExit(Collider)`. Zusätzlich wurde diese Ellipse um eine Zuordnungstabelle mit den kollidierten Objekten erweitert. Die erwähnten Methoden werden automatisch beim Eintreten beziehungsweise Verlassen einer Kollision aufgerufen. Dies passiert im Hintergrund, somit ist die Zuordnungstabelle mit den kollidierten Fahrzeugen immer aktuell. Bei diesen Kollisionen spricht man von einer haptischen und nicht von einer physischen Kollision. Sobald also ein Unfall beziehungsweise eine physische Kollision des Fahrzeugs erkannt wird, führt das Fahrzeug eine Notbremsung durch, schaltet das Warnblinklicht ein und benachrichtigt die Fahrzeuge inklusive der eigenen Identifikationsnummer, Nachrichtnamen, Fahrtrichtung und Position des Fahrzeugs in der Umgebung.

Da die simulierten autonomen Fahrzeuge nicht wirklich über die Luft kommunizieren wurden für den Nachrichtenaustausch eigene Nachrichtenobjekte implementiert. Das Format für den Nachrichtenaustausch ist wie folgt definiert: `Message(VersenderID, AusgehendeNachricht, RichtungUnfallauto, PositionUnfallauto, EmpfaengerID, EingehendeNachricht)`. Wenn die ID des Empfängers `EmpfaengerID` und die eingehende Nachricht `EingehendeNachricht` nicht gesetzt sind, werden alle Fahrzeuge in der Umgebung benachrichtigt. Die eingehende Nachricht `EingehendeNachricht` wird bei einer Weiterleitung aus der ausgehenden Nachricht `AusgehendeNachricht` übernommen. Dieses Vorgehen gibt die Möglichkeit Nachrichtenobjekte inklusive notwendigen Informationen zwischen den Fahrzeugen auszutauschen. Die Tabelle 3.1 zeigt die verschiedenen Nachrichtennamen inklusive des Typs und der Beschreibung auf einen Blick.

Tabelle 3.1: Überblick der Nachrichtennamen inklusive des Typs und der Beschreibung bei der Kommunikation zwischen autonomen Fahrzeugen im Simulator.

Name	**Typ**	**Beschreibung**
`SPIN`	Drehung	Drehung des Fahrzeugs
`COLLISION`	Kollision	Zusammenstoß des Fahrzeugs
`FORWARD`	Weiterleitung	Weiterleitung der ankommenden Nachricht
`RESCUE`	Rettung	Benachrichtigung der Rettungsfahrzeuge über einen Unfall
`LANE`	Fahrspur	Eigene Fahrspur des Fahrzeugs
`TOTALLANES`	Gesamtspuren	Gesamtspuren der Autobahn
`LOCALMAP`	Lokale Karte	Aufbau der lokalen Karte
`POSITION`	Position	Eigene Spur des Fahrzeugs und ID des vorderen Fahrzeugs
`CHANGESTATE`	Zustandsänderung	Änderung des Zustands (Rettungsgassenbildung stehender Verkehr)

Bei den Nachrichten kann ebenfalls zwischen Drehung `SPIN`, Unfall beziehungsweise Kollision `COLLISION`, Weiterleitung `FORWARD`, Rettung `RESCUE`, Fahrspur `LANE`, Gesamt-

spuren `TOTALLANES`, lokale Karte `LOCALMAP`, Position `POSITION` und Zustandsänderung `CHANGESTATE` unterschieden werden. Die letzten drei Nachrichtentypen werden für die Rettungsgassenbildung beim stehenden Verkehr benötigt. Die Fahrzeuge, die eine Nachricht `SPIN` oder `COLLISION` von einem Unfallfahrzeug bekommen, entscheiden wie diese auf die Nachricht reagieren. Dabei werden die Fahrtrichtung und die Position dieser Fahrzeuge verglichen. Nur die Fahrzeuge, die sich in die gleiche Richtung bewegen und sich hinter dem Unfallfahrzeug befinden, bremsen und leiten diese Nachricht mit dem Namen `FORWARD` an die Fahrzeuge in der Umgebung (hinteren Fahrzeuge) weiter. Alle in der Nähe befindlichen Fahrzeuge, die keine Notbremsung durchführen müssen, da es für diese Fahrzeuge keine Gefahr besteht, nehmen die Nachricht an und verständigen die Einsatzkräfte mit der Nachricht `RESCUE`. Sobald ein autonomes Fahrzeug die Gesamtspuren der Autobahn wissen möchte, kann dieses die Nachricht `TOTALLANES` versenden. Die Fahrzeuge in der Umgebung antworten mit der Angabe ihrer eigenen Fahrspur in der Nachricht `LANE`.

Für die Bildung der Rettungsgasse bei einem stehenden Verkehr wird durch die Kommunikation zwischen den simulierten Fahrzeugen eine lokale Karte der Fahrzeuge in der Umgebung aufgebaut. Um die Informationen jedes Fahrzeugs anzufragen wird der Nachrichtenname `LOCALMAP` versendet. Die Fahrzeuge in der Umgebung antworten mit dem Nachrichtennamen `POSITION`. Diese Nachricht enthält die Informationen über die aktuelle Fahrspur des Fahrzeugs und die Identifikationsnummer des vorderen Fahrzeugs auf derselben Spur. Durch diese Informationen kann jedes Fahrzeug eine eigene lokale Karte mit den Fahrzeugen in der Umgebung aufbauen. Für den Aufbau einer lokalen Karte wird als Datenstruktur ein Wörterbuch mit Fahrspuren als Schlüssel und Listen als Wert verwendet. Diese Listen enthalten die Identifikationsnummern der einzelnen Fahrzeuge auf der einzelnen Fahrspur. Die nachfolgende Tabelle 3.2 zeigt ein Beispiel für eine lokale Karte auf einer zweispurigen Autobahn. Die Unfallfahrzeuge stehen in Reihe 0. Diese Fahrzeuge beteiligen sich nicht an dem Aufbau der lokalen Karte.

Tabelle 3.2: Beispiel für eine lokale Karte mit zwölf autonomen simulierten Fahrzeugen und zwei Unfallfahrzeugen auf einer zweispurigen Autobahn. Die Werte in den einzelnen Reihen sind als Identifikationsnummer (ID) der autonomen simulierten Fahrzeuge angegeben. Die Unfallfahrzeuge sind mit der ID 0 gekennzeichnet.

Spur	**Reihe 0**	**Reihe 1**	**Reihe 2**	**Reihe 3**	**Reihe 4**	**Reihe 5**	**Reihe 6**
1	0	11	9	7	5	3	1
2	0	12	10	8	6	4	2

Für das Ändern des Zustands für die Ausführung der Rettungsgassenbildung bei einem stehenden Verkehr, wird die Nachricht `CHANGESTATE` versendet. Zusätzlich beim Ankommen einer Nachricht färbt sich der Lidar, der bisher ohne Funktion ist, auf dem Dach des Fahrzeugs gelb. Dies ist nur dafür da, um das Ankommen der Nachrichten für das menschliche Auge sichtbar zu machen. Sobald Nachrichten weitergeleitet werden, färbt sich dieser grün. Beim Austauschen der Fahrspuren zwischen den Fahrzeugen, nimmt

der Lidarsensor die Farbe Blau an. Nach der Erzeugung der lokalen Karte des Fahrzeugs färbt sich der Lidar des Fahrzeugs weiß. Ebenfalls werden die Nachrichten zur Kontrolle in Unity gespeichert und aktuell gehalten. Jedes Mal, sobald eine Nachricht empfangen wird, werden die aktuell ankommenden Daten übernommen.

Einige Vorexperimente zeigen, dass die Kommunikation zwischen den Fahrzeugen in der Umgebung mit der Ellipse als Spielobjekt schneller gegenüber der Kugel funktioniert. Die Nachrichten können somit schneller an die Fahrzeuge in der Umgebung übermittelt werden. Dies hängt damit zusammen, dass die haptische Erkennung von kollidierten Fahrzeugen in der Nähe sich im Hintergrund befindet und zu jedem Zeitpunkt der Ausführung des Simulators immer aktuell ist. Bei der Kommunikation durch eine Kugel hingegen, werden jedes Mal beim Versenden der Nachrichten, die Fahrzeuge in der Umgebung errechnet. Bei vielen Nachrichten pro Fahrzeug, benötigt dies natürlich mehr Zeit.

3.4 Algorithmen

Wie bereits erwähnt ist im Falle eines Unfalls die Bildung der Rettungsgasse für Polizei- und Rettungsfahrzeuge auf Autobahnen sehr wichtig. Sobald die autonomen Fahrzeuge auf unsere Straßen kommen, könnte das Risiko für unvorhersehbare Unfälle gesenkt werden. Doch durch unvorhersehbare technische Mängel oder Softwarefehler können die Unfälle leider nicht komplett vermieden werden. Um der angesprochenen Problematik entgegenzuwirken und Unfälle auf den Autobahnen zu reduzieren, werden in diesem Abschnitt zwei Algorithmen für die Bildung der Rettungsgasse mit autonomen Fahrzeugen für die Polizei- und Rettungsfahrzeuge im Simulator vorgestellt. Diese Algorithmen werden für den in diesem Kapitel vorgestellten Simulator entwickelt und anschließend getestet. Wie das Kapitel 5 zeigt, können diese Algorithmen für die Rettungsgassenbildung ebenfalls in der Praxis eingesetzt werden.

3.4.1 Bildung der Rettungsgasse: Stockender Verkehr

Der Algorithmus 2 wurde für die Bildung der Rettungsgasse bei einem stockenden Verkehr entwickelt. Um in dem Simulator eine Rettungsgasse mit autonomen Fahrzeugen bilden zu können, müssen diese Fahrzeuge wissen, auf welcher Fahrspur sich diese Fahrzeuge befinden. Zusätzlich ist auch die Information über die maximalen Fahrspuren der Autobahn von Nöten. Beispielsweise auf einer Autobahn mit zwei oder mehr Fahrspuren, fahren die Fahrzeuge auf der äußersten (letzten) Fahrspur näher an die Mittelleitplanke heran. Die Fahrzeuge auf der vorletzten Fahrspur fahren dagegen auf die rechte Fahrbahnmarkierung ihrer Fahrspur. Um die Position des linken oder rechten Vorderrads mit den Fahrbahnmarkierungen zu vergleichen, wird im Simulator ebenfalls auf die Technik der Raycasts zurückgegriffen. Daraus lässt sich ableiten, in welche Richtung (links oder rechts) das Fahrzeug fahren muss, um eine Rettungsgasse zu bilden. Sobald die Fahrgeschwindigkeit unter einen bestimmten Wert fällt (z.B. 30 km/h), bilden die Fahrzeuge automatisch einen Rettungskorridor. Das bedeutet, dass bei einem stockenden Verkehr oder einem Stau immer eine Rettungsgasse gebildet ist. Der Algorithmus 2 zur Bildung der Rettungsgasse bei einem stockenden Verkehr funktioniert nach einem

einfachen Prinzip. Die letzte Fahrspur befindet sich an der Mittelleitplanke. Somit wird beim Auffahren auf die Autobahn mit der Nummerierung der Fahrspuren begonnen. Die aktuelle Fahrgeschwindigkeit muss über 0 km/h liegen, um den Algorithmus für die Bildung der Rettungsgasse bei einem stockenden Verkehr bilden zu können. Zusätzlich muss die aktuelle Fahrspur des Fahrzeugs kleiner oder gleich den Gesamtspuren auf der Autobahn sein. Dabei muss auch die Anzahl der Gesamtspuren über eins betragen.

Algorithmus 2: Bildung der Rettungsgasse: Stockender Verkehr

Eingabe : Geschwindigkeit > 0, AktuelleSpur $\leq$ MaxSpuren, MaxSpuren > 1, G $\in [1, 30]$

Ausgabe : Steuerung des Fahrzeugs

```
1  für jeden einzelnen Schritt tue
2      wenn Geschwindigkeit ≤ G und AktuelleSpur = MaxSpuren dann
3          erkenne die linke Fahrbahnmarkierung
4          bewege das Fahrzeug nach links, bis es die linke Fahrbahnmarkierung
             berührt
5          halte das linke Vorderrad auf der linken Fahrbahnmarkierung
6      sonst wenn Geschwindigkeit ≤ G und AktuelleSpur = MaxSpuren − 1
         dann
7          erkenne die rechte Fahrbahnmarkierung
8          bewege das Fahrzeug nach rechts, bis es die rechte Fahrbahnmarkierung
             berührt
9          halte das rechte Vorderrad auf der rechten Fahrbahnmarkierung
10     sonst
11         erkenne den Fahrspurverlauf
12         bewege das Fahrzeug in die Mitte der Fahrspur
13         halte das Fahrzeug in der Mitte der Fahrspur
```

3.4.2 Bildung der Rettungsgasse: Stehender Verkehr

Der Algorithmus 3 wurde für die Bildung der Rettungsgasse bei einem stehenden Verkehr entwickelt. Aus unvorhersehbaren Gründen kann es vorkommen, dass die Fahrzeuge ohne der Bildung der Rettungsgasse zum Stehen kommen und somit ein Korridor für die Polizei- und Rettungsfahrzeug nicht gebildet ist. Anschließend könnten die Fahrer dieser Fahrzeuge auf Knopfdruck eine Rettungsgasse bilden. Das in diesem Abschnitt vorgestellte Vorgehen wird als erstes für die Simulation entwickelt und im Abschnitt 3.5.4 getestet. Für die Durchführung des Algorithmus 3 ist es erforderlich, eine lokale Karte für jedes einzelne Fahrzeug zu erzeugen. Das heißt, jedes Fahrzeug enthält die Informationen über die Position jedes einzelnen Fahrzeugs inklusive der Fahrspur in der Umgebung. Durch das Austauschen eigener Identifikationsnummer, eigener Spurposition und der Identifikationsnummer des vorderen Fahrzeugs, kann ebenfalls die Reihenfolge der Fahrzeuge auf den verschiedenen Fahrspuren in der lokalen Karte erstellt werden.

Algorithmus 3: Bildung der Rettungsgasse: Stehender Verkehr

Eingabe : StartGeschwindigkeit $= 0$, AktuelleFrontDistanz > 0, MinFrontDistanz $= 3$, AktuelleSpur $\leq$ MaxSpuren, MaxSpuren > 1, Zustand ≥ 0

Ausgabe : Steuerung des Fahrzeugs

```
1  versende Nachricht LOCALMAP
2  empfange Nachrichten POSITION
3  baue eine lokale Karte der Fahrzeuge und deren Vorderfahrzeuge auf
4  wenn AktuelleSpur = MaxSpuren oder AktuelleSpur = MaxSpuren − 1 dann
5      wenn Das erste Fahrzeug in der Fahrspur dann
6          Zustand = EXEC_BACKWARD
7      sonst
8          Zustand = WAIT
9  für jeden einzelnen Schritt tue
10     unterscheide Zustand tue
           /* Warten auf Ausführung                                    */
11         Fall WAIT tue
12             bewege das Fahrzeug nicht
13             warte auf Nachricht CHANGESTATE zum Wechseln des Zustands
           /* Ausführung, ggf. Rückwärtsfahren und Zustand wechseln    */
14         Fall EXEC_BACKWARD tue
15             wenn AktuelleFrontDistanz < MinFrontDistanz dann
16                 bewege das Fahrzeug gerade nach Hinten
17                 überprüfe dabei den hinteren Abstand
18             setze Zustand = EXEC_FORWARD
           /* Ausführung, nur vorwärts fahren                          */
19         Fall EXEC_FORWARD tue
20             verkleinere den Sicherheitsabstand zum vorderen Fahrzeug
21             erhöhe das Lenkverhalten
22             bewege das Fahrzeug nach vorne
23             führe Bildung der Rettungsgasse: Stockender Verkehr aus
24             wenn Ausführung abgeschlossen dann
25                 versende CHANGESTATE mit Zustand = EXEC_BACKWARD an
                    das hintere Fahrzeug
           /* Normalzustand, nur vorwärts fahren                       */
26         Fall NORMAL_FORWARD tue
27             erhöhe den Sicherheitsabstand zum vorderen Fahrzeug
28             verkleinere das Lenkverhalten
29             bewege das Fahrzeug nach vorne
30             führe Bildung der Rettungsgasse: Stockender Verkehr aus
           /* Keine gültige ID des Fahrzeugs oder Unfallfahrzeug       */
31         sonst tue
32             bewege das Fahrzeug nicht
```

Anschließend kann entschieden werden, wer mit der Ausführung der Rettungsgasse beginnt. Die Ausführung der Rettungsgassenbildung bei einem stehenden Verkehr erfolgt über verschiedene Zustände. So kann systematisch die Bildung der Rettungsgasse Reihe für Reihe durchgeführt werden. Die Fahrzeuge, die den Unfallfahrzeugen am nächsten sind, beginnen mit der Bildung der Rettungsgasse. Sobald das ausführende Fahrzeug die Bildung der Rettungsgasse abgeschlossen hat, benachrichtigt dieses das Fahrzeug dahinter in der gleichen Fahrspur. In der Regel ist auf deutschen Autobahnen das Rückwärtsfahren verboten. Dies besagt der Paragraf 18 Absatz 7 (§ 18 Abs. 7) der Straßenverkehrs-Ordnung (StVO) [11]. Doch bei einem unvorhersehbaren Unfall muss in einer Notsituation eine Ausnahme gemacht werden, um die Rettungsfahrzeuge durchzulassen. Dabei kann nach dem Prinzip „Not kennt kein Gebot“ gehandelt werden. Der vorherige Algorithmus 3 stellt das Vorgehen für die Rettungsgassenbildung bei einem stehenden Verkehr auf einen Blick dar.

3.5 Experimente

Die nachfolgenden Experimente wurden zur Kontrolle der Funktionalität des entwickelten Simulators und der entwickelten Algorithmen für die Bildung der Rettungsgasse bei einem stockenden und stehenden Verkehr für autonome Fahrzeuge durchgeführt. Diese Experimente beinhalten die Rettungsgassenbildung bei einem stockenden Verkehr, bei Hindernissen auf der Fahrbahn, beim Platzen des Vorderreifens und bei einem stehenden Verkehr.

3.5.1 Bildung der Rettungsgasse: Stockender Verkehr

Die Rettungsgasse wird immer dann gebildet, wenn die Fahrgeschwindigkeit unter einen bestimmten Wert, zum Beispiel 30 km/h, fällt. Bei einem stockenden Verkehr bilden die Fahrzeuge automatisch eine Rettungsgasse. Dies wird in der nachfolgenden Abbildung 3.7a veranschaulicht.

(a) Öffnen einer Rettungsgasse.

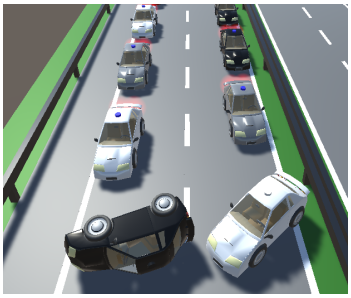
(b) Schließen einer Rettungsgasse.

Abbildung 3.7: Bildung einer Rettungsgasse bei einem stockenden Verkehr auf einer regulären zweispurigen Autobahn.

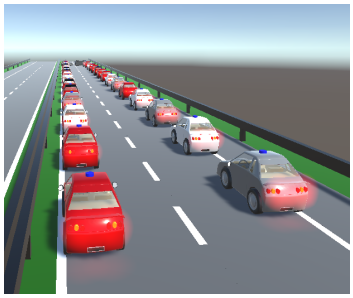
Ebenfalls kann man erkennen, dass die Fahrzeuge bis zu einer bestimmten Geschwindigkeit alle geordnet in Reihen fahren. Dies passiert durch das Halten des Sicherheitsabstands zum vorderen Fahrzeug. So wird die Straße komplett ausgenutzt. Damit ist auch sichergestellt, dass die Fahrzeuge rechtzeitig zum Stehen kommen, ohne einen Unfall zu verursachen und somit die Insassen nicht zu verletzten. Sobald die Entfernung zum nachfolgenden Fahrzeug sich vergrößert, beschleunigen die Fahrzeuge automatisch. Wenn die Geschwindigkeit größer wird, vergrößern die Fahrzeuge automatisch den Sicherheitsabstand und schließen die Lücke für die Rettungsfahrzeuge (Abb. 3.7b).

3.5.2 Bildung der Rettungsgasse: Hindernisse auf der Fahrbahn

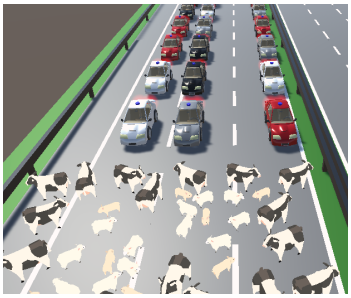
Dieses Experiment zeigt die Bildung einer Rettungsgasse der autonomen simulierten Fahrzeuge bei der Annäherung an Hindernisse auf einer Autobahn. Dies könnte zum Beispiel ein Unfall, der die gesamte Autobahn blockiert oder ein unerwarteter Wildwechsel durch die Weidetiere, sein. Die nachfolgenden Abbildungen 3.8a und 3.8b zeigen die Bildung der Rettungsgasse im Falle eines Unfalls auf einer zweispurigen Autobahn.



(a) Auf einer regulären zweispurigen Autobahn (Frontansicht).



(b) Auf einer regulären zweispurigen Autobahn (Rückansicht).



(c) Auf einer regulären dreispurigen Autobahn (Frontansicht).

(d) Auf einer regulären dreispurigen Autobahn (Rückansicht).

(e) Auf einer breiten dreispurigen Autobahn (Frontansicht).

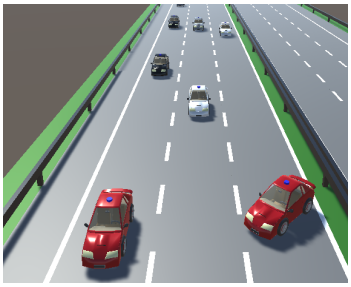
(f) Auf einer regulären vierspurige Autobahn (Frontansicht).

Abbildung 3.8: Bildung einer Rettungsgasse mit simulierten autonomen Fahrzeugen bei Hindernissen auf Autobahnen mit verschiedener Anzahl und Breite der Fahrspuren.

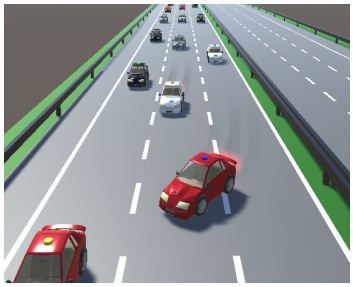
Auf einer dreispurigen Autobahn wurde die automatische Rettungsgassenbildung im Falle eines Hindernisses ebenfalls getestet. Dies ist in den Abbildungen 3.8c und 3.8d zu sehen. Die simulierten Fahrzeuge kennen durch die Verwendung von Raycasts die Breite der Fahrspur. So kann diese Breite bei der Bildung der Rettungsgasse berücksichtigt werden. Daher kann die Breite der Fahrspur variieren. Dies ist in Abbildung 3.8e zu sehen. Um sicherzustellen, dass die Simulation auch für mehr als drei Fahrspuren auf einer Autobahn funktioniert, wurde eine vierspurige Autobahn ebenfalls erfolgreich getestet (Abb. 3.8f).

3.5.3 Bildung der Rettungsgasse: Platzen des Vorderreifens

Um herauszufinden, ob die Rettungsgasse zu jedem unerwarteten Zeitpunkt gebildet werden kann, wurde ein Unfall bei der Fahrt simuliert. Dadurch konnte festgestellt werden, ob eine Rettungsgasse von den autonomen Fahrzeugen bei jeder Gelegenheit gebildet werden kann. Die Simulation dieses Experiments erfolgte zufällig, um jeden möglichen Fall abzudecken. Das Fahrzeug für den Unfall, die Rotation in Grad und die Fahrgeschwindigkeit des Fahrzeugs bei dem Unfall wurden aus diesem Grund zufällig festgelegt. Diese Simulation soll das Platzen eines Vorderreifens bei der Fahrt vormachen. Die nachfolgende Abbildung 3.9 veranschaulicht dieses Experiment. Die Fahrzeuge, die sich vor dem Unfallfahrzeug befinden, fahren weiter und benachrichtigen die Rettungskräfte (Abb. 3.9a). Die anderen Fahrzeuge, hinter dem Unfallfahrzeug, bleiben stehen (Abb. 3.9b), um einen Aufprall zu vermeiden. In der Abbildung 3.9b kann man ebenso erkennen, dass das Lidar der Fahrzeuge sich gelb gefärbt hat. Dies veranschaulicht das Ankommen einer Nachricht. Durch diese Experimente konnte das unvorhersehbare und unvorstellbare Verhalten des Fahrzeugs durch die Physik gezeigt werden. Vor allem war die Reaktion des Fahrzeugs auf die Einstellung der Rotation immer unterschiedlich. Manchmal rutscht das Fahrzeug meterweit über die Fahrbahn oder kollidiert teilweise mit den Leitplanken, je nach Einstellung der zufälligen Rotation und der gewählten Fahrspur und der Geschwindigkeit. Dieses ergibt sich aus den G-Kräften, die von der Richtung und Drehung abhängig sind. Die Abbildung 3.9c zeigt die erfolgreich gebildete Rettungsgasse beim Platzen des Vorderreifens.



(a) Linkes rotes Fahrzeug fährt vorbei.



(b) Durchführung der Gefahrenbremsung.

(c) Erfolgreich gebildete Rettungsgasse.

Abbildung 3.9: Simulation eines geplatzten Vorderreifens während der Fahrt eines autonomes Fahrzeugs auf einer regulären dreispurigen Autobahn.

3.5.4 Bildung der Rettungsgasse: Stehender Verkehr

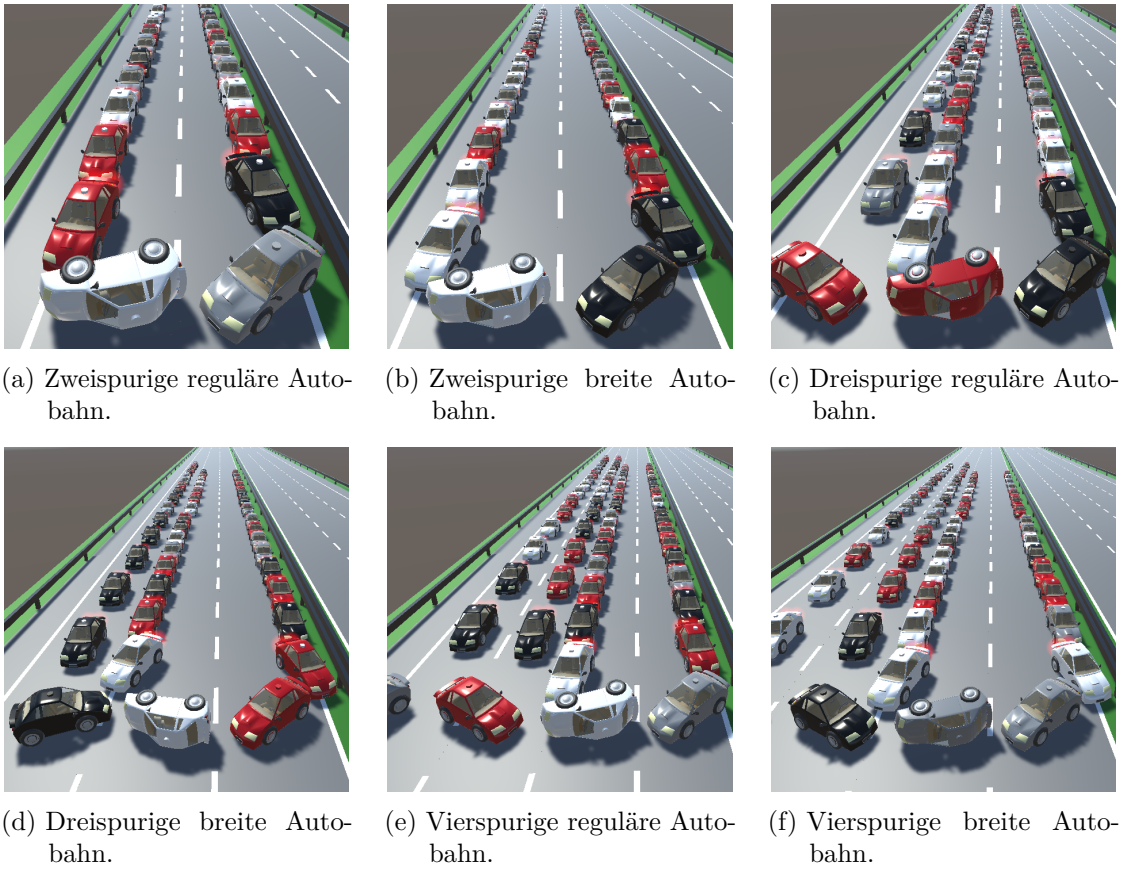
Dieses Experiment zeigt die Bildung einer Rettungsgasse der autonomen simulierten Fahrzeuge bei einem stehenden Verkehr. Wie bereits erwähnt benötigen die Fahrzeuge für die Bildung der Rettungsgasse bei einem stehenden Verkehr eine lokale Karte, welche durch die Kommunikation zwischen den Fahrzeugen erzeugt werden kann. So können systematisch Reihe für Reihe die Fahrzeuge eine Rettungsgasse für die Polizei- und Rettungsfahrzeuge bilden. Die nachfolgende Abbildung 3.10 zeigt die systematische Ausführung dieses Algorithmus.

(a) Erste Reihe. (b) Zweite Reihe. (c) Dritte Reihe.

Abbildung 3.10: Bildung einer Rettungsgasse beim simulierten Unfall auf einer regulären zweispurigen Autobahn beim stehenden Verkehr.

Auf der Abbildung 3.10a kann man erkennen, dass die Fahrzeuge, die den Unfallautos am nähesten sind, mit der Bildung der Rettungsgasse beginnen. Zusätzlich erkennt man, dass diese Fahrzeuge nach hinten Fahren, um beim Ausrangieren genügend Platz vor sich zu haben. Wie die Abbildung 3.10b zeigt, auch die Fahrzeuge in der zweiten Reihe haben noch nicht ausreichend Platz für die Bildung der Rettungsgasse. Aus diesem Grund fahren diese Fahrzeuge ebenfalls zuerst nach hinten. Die Fahrzeuge in der dritten Reihe hingegen, haben ausreichend Platz und fahren nicht mehr nach hinten (Abb. 3.10c). Das ergibt sich aus der Verkürzung des Abstands zu den vorderen Fahrzeugen im Algorithmus.

Durch die im Abschnitt 3.3.3 vorgestellte Ellipse für die Kommunikation zwischen den simulierten Fahrzeugen (Abb. 3.6b) werden nicht alle Fahrzeuge über die Bildung der Rettungsgasse informiert. Somit kann die Rettungsgasse in einem kleinen Kreis kommuniziert und gebildet werden. Durch die Verkürzung des Abstands der vorderen Fahrzeuge haben, zum Beispiel, die Fahrzeuge ab der sechsten Reihe genug Platz für eine Bildung der Rettungsgasse bei der Fahrt. So können diese gemeinsam eine Bildung der Rettungsgasse bei einem stockenden Verkehr durchführen. Der Algorithmus für die Bildung der Rettungsgasse bei einem stehenden Verkehr wurde ebenfalls auf verschiedenen simulierten Autobahnen mit unterschiedlicher Anzahl an Fahrspuren und Fahrspurbreiten getestet. Dies zeigt die nachfolgende Abbildung 3.11.



(a) Zweispurige reguläre Autobahn.
(b) Zweispurige breite Autobahn.
(c) Dreispurige reguläre Autobahn.
(d) Dreispurige breite Autobahn.
(e) Vierspurige reguläre Autobahn.
(f) Vierspurige breite Autobahn.

Abbildung 3.11: Bildung einer Rettungsgasse bei simulierten Unfällen auf Autobahnen mit verschiedener Anzahl und Breite der Fahrspuren beim stehenden Verkehr.

3.5.5 Auswertung der Ergebnisse

Nach der Durchführung aller Experimente in der Simulation, können die Ergebnisse ausgewertet werden. Dabei kann man sagen, dass jedes durchgeführte Experiment zur vollsten Zufriedenheit durchgeführt werden konnte. Die Bildung der Rettungsgasse konnte zu jedem Zeitpunkt, beim stockenden Verkehr, bei Hindernissen auf der Fahrbahn, beim Platzen des Vorderreifens und beim stehenden Verkehr, erfolgreich von den simulierten autonomen Fahrzeugen durchgeführt werden. Das heißt die Algorithmen konnten erfolgreich in der Simulation eingesetzt werden. Zusätzlich bestätigt sich durch diese Experimente die Funktionalität des entwickelten Simulators. Außerdem zeigen die Experimente, dass für die Bildung der Rettungsgasse einige Parameter sehr wichtig sind. Die Gesamtanzahl der Fahrspuren und die eigene Fahrspurnummer des autonomen Fahrzeugs auf der Autobahn müssen bekannt sein. Ohne diese Parameter kann die Rettungsgasse mit autonomen Fahrzeugen auf einer Autobahn nicht mehr gebildet werden. Die Gesamtanzahl der

Fahrspuren auf der Autobahn wird durch die Kommunikation zwischen den Fahrzeugen im Simulator gefunden. Die eigene Fahrspurnummer wird im Fahrzeug fest eingetragen, kann aber auch beim Auffahren auf die Autobahn mitgezählt werden.

Wie bereits in Abschnitt 2.4.4 vorgestellt, hängt die Einstufung der Automatisierung von den in dem Fahrzeug verwendeten Technologien und dem notwendigen Eingriff des Fahrers in den Fahrprozess ab. Die Bildung der Rettungsgasse bei einem stockenden Verkehr könnte ab der dritten Stufe der Automatisierung automatisch von dem autonomen Fahrzeug durchgeführt werden. Ab dieser Stufe ist dieses Fahrzeug vollkommen autonom. Der Fahrer greift nur in einer Gefahrsituation in den Fahrprozess ein. Bei der Bildung der Rettungsgasse bei einem stehenden Verkehr hingegen, kann der Algorithmus bereits ab der zweiten Stufe der Automatisierung automatisch auf Knopfdruck durchgeführt werden. Dabei hat das Fahrzeug die Erkennung und die Reaktion integriert und der Fahrer ist bereit die Kontrolle zu übernehmen. Die einzige Voraussetzung ist hierbei das einheitliche System für die Kommunikation zwischen den autonomen Fahrzeugen in der Umgebung für die Erzeugung der lokalen Karte, die für die Bildung der Rettungsgasse bei einem stehenden Verkehr unerlässlich ist.

3.6 Schlussfolgerungen

Die Motivation für die Ausarbeitung der Thematik der Rettungsgassenbildung ist, dass Menschenleben rechtzeitig gerettet werden und dass jede Sekunde bei einem Unfall wichtig sein kann. Das unnötige Behindern der Polizei- und Rettungsfahrzeuge zum Unfallort soll verringert beziehungsweise komplett vermieden werden. Dies kann in Zukunft durch die für den Straßenverkehr zugelassenen autonomen Fahrzeuge realisiert werden. Aus diesem Grund wurde in diesem Kapitel ein Simulator in Unity inklusive der Algorithmen für die Bildung der Rettungsgasse implementiert und vorgestellt. Dieser Simulator dient ebenfalls dazu, das Verhalten von autonomen Fahrzeugen bei unvorhersehbaren Unfällen im Autobahnverkehr zu demonstrieren. Die simulierten Fahrzeuge bewegen sich autonom. Sie beschleunigen automatisch, halten den Sicherheitsabstand ein, passen den Sicherheitsabstand an ihre Geschwindigkeit an und passen den Rettungskorridor an die Breite der Fahrbahn an. Wenn die Geschwindigkeit unter einen bestimmten Wert fällt, wird automatisch eine Rettungsgasse geöffnet. Dies ist bei einem langsam fließenden Verkehr wichtig. Übersteigt die Geschwindigkeit einen bestimmten Wert, wird die Rettungsgasse automatisch geschlossen. Die Kommunikation zwischen den Fahrzeugen ist sowohl in der Simulation als auch in der Praxis sehr wichtig. Die Fahrzeuge tauschen Nachrichten über die Situation auf der Straße aus. Dies ermöglicht es, Entscheidungen zu treffen und die Richtung und die Position des Unfallfahrzeugs weiterzugeben. Ebenfalls wie die Experimente zeigen, kann durch die Kommunikation eine Bildung der Rettungsgasse beim stehenden Verkehr durchgeführt werden. Zusätzlich wurden verschiedene Testfälle bei den Experimenten zufällig generiert, um möglichst viele verschiedene Verhaltensweisen abzudecken.

In dem Kapitel 5 wird eine Übertragung von der entwickelten Simulation auf die Realität vorgestellt. Es wird geplant von den in diesem Kapitel vorgestellten simulierten autonomen

Fahrzeugen auf die autonome Modellfahrzeuge in der echten Umgebung umzusteigen. Somit ist die Idee, die in diesem Kapitel entwickelten Algorithmen, auf Modellfahrzeugen in der echten Welt zu überprüfen. In dem Simulator wird die Fahrspurerkennung mit Hilfe von Raycasts durchgeführt. Doch in der Praxis gibt es diese Hilfsmittel nicht. Daher ist der nächste Schritt die Implementierung einer Fahrspurerkennung mit einer Videokamera durch eine Bilderkennung, wie dies auch aus der autonomen Automobilindustrie bekannt ist. Dabei wird der Vergleich zwischen den traditionellen Methoden und den Methoden des maschinellen Lernens in Betracht gezogen. Diese Thematik wird in dem nachfolgenden Kapitel 4 vorgestellt.

Das Experiment, dem nicht eine Theorie, d.h. eine Idee vorausgeht, verhält sich zur Naturforschung wie das Rasseln einer Kinderklapper zur Musik.

Justus von Liebig
1803 – 1873

Kapitel 4

Spurerkennung

4.1 Einführung

Dieses Kapitel präsentiert zwei verschiedene Ansätze zur Erkennung des Spurverlaufs der eigenen Fahrbahn in einem zweidimensionalen Bild. Diese Spurerkennung ist für den in Kapitel 3 entwickelten Simulator von Nöten, um das Halten des Fahrzeugs in der Fahrbahn durch eine Videokamera zu realisieren. Dadurch ist die spätere, im nächsten Kapitel 5 beschriebene, Übertragung von der Simulation auf die Realität möglich. Bei der Entwicklung dieser Spurerkennung stellt sich ebenfalls die Forschungsfrage: *Bieten die traditionellen Methoden eine ausreichende Funktionalität angewendet auf eine bildliche Spurerkennung oder muss hierbei auf die künstlichen neuronalen Netzwerke ausgewichen werden?*

Um diese Frage zu beantworten werden in diesem Kapitel zwei Methoden, der gefilterte Canny-Algorithmus und ein faltendes neuronales Netzwerk (ConvNet), für die Spurerkennung in dem zuvor vorgestellten Simulator entwickelt, implementiert und evaluiert. Zu Beginn wird auf die verwandten Arbeiten dieser Thematik eingegangen. Anschließend wird der gefilterte Canny-Algorithmus für die Spurerkennung vorgestellt, Schritt für Schritt erklärt und ausgewertet. Dieser wird speziell für die Fahrspurerkennung entwickelt. Zusätzlich wird eine optimale Konfiguration der Parameter für das faltende neuronale Netzwerk gefunden. Die Netzwerkarchitektur des ConvNets wird ebenfalls vorgestellt und erklärt. Für das überwachte Lernen des ConvNets sind bekanntlich viele annotierte Trainingsdaten notwendig. Diese annotierten Trainingsdaten werden mit dem, in Kapitel 3 vorgestellten, Simulator für Rettungsgassen- und Unfallsimulationen erzeugt und automatisch annotiert. Das Vorgehen für das Erzeugen automatisch annotierter Trainingsdaten wird ebenfalls in diesem Kapitel vorgestellt. Diese beiden entwickelten Systeme der Spurerkennung werden zusätzlich verglichen, um das bessere und schnellere System für die Spurerkennung für den entwickelten Simulator zu finden. Durch die in diesem Kapitel beschriebenen Experimente wird der Vergleich der Laufzeit und des Fehlers

der Algorithmen präsentiert. Die Laufzeit und der Fehler der beiden Vorgehensweisen, abhängig von der Bildgröße, wird ebenfalls angegeben [55]. Wie bereits erwähnt ist das Ziel dieser Forschungsarbeit von der zuvor entwickelten Simulation auf die echten Modellautos umzusteigen, um die entwickelten Algorithmen in Kapitel 3 und in diesem Kapitel zu übertragen und zu überprüfen. Dies beschreibt das nächste Kapitel 5. Dabei wird ein ähnliches Verhalten der Modellfahrzeuge in der echten Umgebung, wie zu vor in der simulierten Umgebung, erwartet.

4.2 Verwandte Arbeiten

Es gibt einige wissenschaftliche Arbeiten, die sich mit der Erkennung der Fahrspur beschäftigen. Zum Beispiel gibt es eine Fahrspurerkennung und Fahrspurverfolgung mit B-Snake [117] oder eine robuste Fahrspurerkennung und Fahrspurverfolgung in anspruchsvollen Szenarien [50]. Ebenfalls gibt es Arbeiten, die mit Hilfe von dem Canny-Algorithmus und der Hyperbelannäherung (engl. Hyperbola Fitting) die Fahrspur erkennen [3]. Ziemlich aktueller Ansatz für die Spurerkennung ist das Einbeziehen von Informationen aus früheren Bildern. Das kann durch das Kombinieren von dem faltenden neuronalen Netzwerk (CNN) und dem rekurrenten neuronalen Netzwerk (RNN) realisiert werden [125]. Ebenfalls gibt es einen Algorithmus zur Erkennung von Straßenfahrzeugen in einem Sicherheitsassistenten auf der Grundlage von künstlicher Intelligenz [17]. Sobald dreidimensionale Informationen verfügbar sind, kann zwischen Straße und Hindernissen unterschieden werden [75]. Solche 3D-Informationen können zum Beispiel mithilfe von Lidarsensoren beschafft und ausgewertet werden [7]. Ein weiterer Ansatz die 3D-Informationen für die Spur- oder Objekterkennung zu beschaffen ist die so genannte Stereokamera. Diese enthält zwei Kameras in einem bestimmten Abstand, ähnlich wie die menschlichen Augen. Diese Kamera liefert zwei Bilder. Durch die beiden Bilder kann die Tiefe des Bildes bestimmt werden, um hinterher zum Beispiel zwischen Straßen, Menschen, Autos und Häusern zu unterscheiden [60]. Viele der wissenschaftlichen Forschungsarbeiten zur Spurerkennung der Fahrspur verzichten auf die Angabe der Laufzeit und der Hardware der entwickelten Algorithmen. Ebenfalls wird teilweise die Auflösung der Eingabebilder nicht angegeben. Somit können die Methoden nicht verglichen werden! Der Ansatz dieser Forschungsarbeit ist es in diesem Kapitel weitere Methoden zu entwickeln, um die Fahrspur in dem bereits in Kapitel 3 vorgestellten Simulator und anschließend in der Realität zu erkennen. Dadurch soll das Gleichgewicht zwischen der Genauigkeit und der Geschwindigkeit der Algorithmen gefunden werden.

4.3 Datensätze

Dieser Abschnitt präsentiert die verschiedenen Datensätze, die für die Spurerkennung im folgenden Kapitel verwendet wurden. Durch die schnelle Erzeugung der Trainingsdaten mit dem in Kapitel 3 vorgestellten Simulator, können Bilder mit verschiedener Breite und Höhe generiert und automatisch annotiert werden. Dieses Vorgehen wird im nachfolgenden Abschnitt 4.3.1 genauer erläutert. Es wurden folgende Auflösungen erstellt und getestet

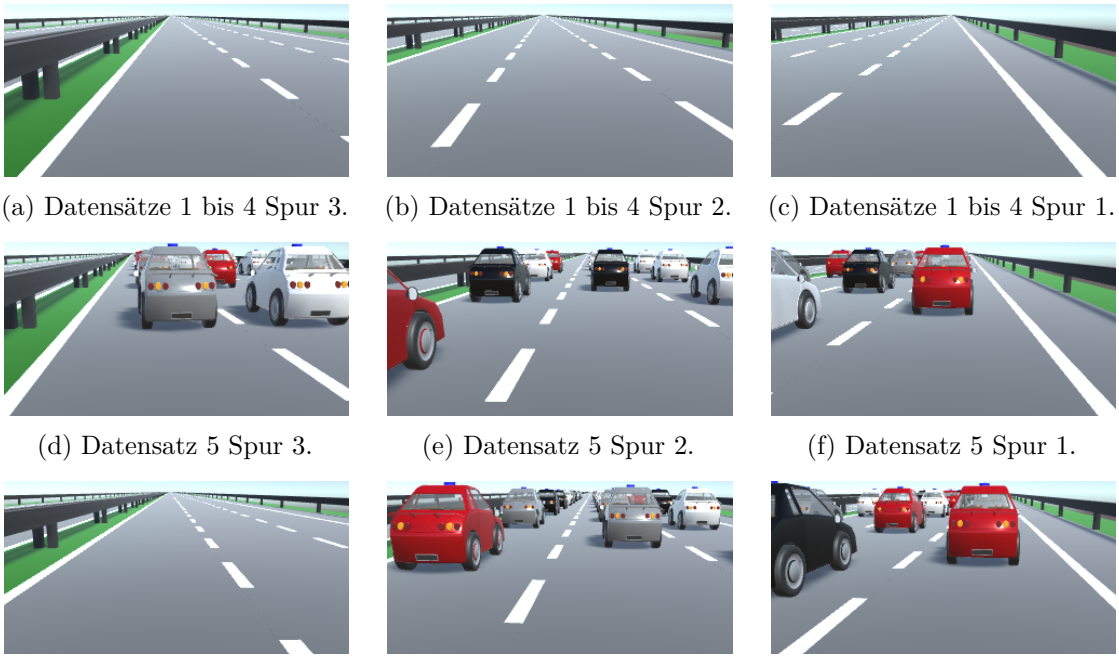
(Breite × Höhe): 160 × 80, 320 × 160, 640 × 160 und 640 × 320 Pixel. Insgesamt wurden 54 Tausend annotierte Bilddateien erstellt. Diese sind in der nachfolgenden Tabelle 4.1 im Überblick dargestellt.

Tabelle 4.1: Erstellte Datensätze für die Spurerkennung in dem Simulator. Die Auflösung der Bilder ist in Pixel (Breite × Höhe) angegeben. Die Anzahl steht für die Menge der einzelnen annotierten Bilder. Die Annotation entspricht der Anzahl der Orientierungspunkte für die Fahrspur im Bild.

Nr.	**Name**	**Auflösung**	**Annotation**	**Beschreibung**	**Anzahl**
1	Sim 1	160 × 80	14	Nur Spur, ohne Fahrzeuge	9000
2	Sim 2	320 × 160	14	Nur Spur, ohne Fahrzeuge	9000
3	Sim 3	640 × 160	14	Nur Spur, ohne Fahrzeuge	9000
4	Sim 4	640 × 320	14	Nur Spur, ohne Fahrzeuge	9000
5	Sim 5	320 × 160	14	Spur mit Fahrzeugen	6000
6	Sim 6	320 × 160	14	Nur Spur, Spur mit Fahrzeugen	12000

Durch die automatische Erzeugung der Bilder gibt es in jedem Datensatz Farbbilder, Graustufenbilder und Binärbilder als Eingabe mit der dazugehörigen Annotation (Anzahl der Orientierungspunkte). So konnten im nächsten Schritt verschiedene Experimente durchgeführt werden. Diese sind ebenfalls in dem Abschnitt 4.6 genauer erklärt. Die Datensätze 1 bis 4 (Sim 1 - 4) beinhalten die Bilder mit der Fahrspur ohne zusätzliche Störfaktoren. Auf den Abbildungen 4.1a, 4.1b und 4.1c kann man somit nur den Spurverlauf inklusive der Leitplanken links und rechts erkennen. Diese Datensätze beinhalten jeweils 3000 Bilder von der ersten, 3000 Bilder von der zweiten und 3000 Bilder von der dritten Spur.

Der Datensatz 5 (Sim 5) enthält 6000 Bilder inklusive der Fahrspur und den verschiedenen simulierten Fahrzeugen auf der simulierten Autobahn. Dieser enthält ebenfalls 2000 Bilder von der ersten, 2000 Bilder von der zweiten und 2000 Bilder von der dritten Fahrspur. Einige Ausschnitte aus dem Datensatz 5 (Sim 5) sind auf den Abbildungen 4.1d, 4.1e und 4.1f zu erkennen. Der Datensatz 6 (Sim 6) wurde aus den Datensätzen 2 (Sim 2) und 5 (Sim 5) mit jeweils 6000 Bildern kombiniert. Die Abbildungen 4.1g, 4.1h und 4.1i veranschaulichen diesen Datensatz. Die Verteilung der Bilder in den Datensätzen ist gleichmäßig. Somit sind alle verwendeten Datensätze ausbalanciert. Die nachfolgende Abbildung 4.1 zeigt einige Ausschnitte aus den beschriebenen Datensätzen für die Spurerkennung mit den Datensätzen 1 bis 6 (Sim 1 bis 6) mit simulierter Autobahn und simulierten autonomen Fahrzeugen auf einen Blick.



(a) Datensätze 1 bis 4 Spur 3. (b) Datensätze 1 bis 4 Spur 2. (c) Datensätze 1 bis 4 Spur 1.

(d) Datensatz 5 Spur 3. (e) Datensatz 5 Spur 2. (f) Datensatz 5 Spur 1.

(g) Datensatz 6 Spur 3. (h) Datensatz 6 Spur 2. (i) Datensatz 6 Spur 1.

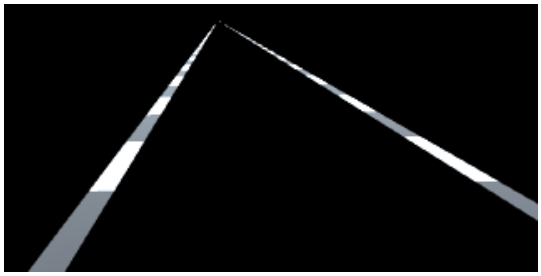
Abbildung 4.1: Annotierte Datensätze für die Spurerkennung mit Datensätzen 1 bis 6 (Sim 1 bis 6) mit simulierter Autobahn und simulierten autonomen Fahrzeugen.

4.3.1 Automatische Annotation

Wie bereits erwähnt, konnten durch den im vorherigen Kapitel 3 vorgestellten Simulator, tausende annotierte Bilddateien automatisch erzeugt werden. Durch die Annotation (Anzahl der Orientierungspunkte) konnten diese Daten ebenfalls als Trainingsdaten für das faltende neuronale Netzwerk und für die Berechnung des Fehlers und der Erkennungsrate bei dem gefilterten Canny-Algorithmus verwendet werden. Für die automatische Erzeugung der benötigten Datensätze wurden in das simulierte Fahrzeug in dem Simulator zwei Kameras an die gleiche Stelle an der Frontscheibe verbaut. Dies ist natürlich nur in der Simulation möglich. Die erste Kamera hatte die komplette Sicht auf die Umgebung inklusive der Fahrspuren, der Fahrbahnmarkierungen, Leitplanken und Fahrzeugen (Abb. 4.2a). Die zweite Kamera sah nur die Spurmarkierung links und rechts auf der eigenen Fahrbahn des Fahrzeugs (Abb. 4.2b). Diese beiden Bilder, die in Abbildung 4.2 zu sehen sind, konnten anschließend für die Erzeugung der Datensätze genutzt werden. Aus der Abbildung 4.2a konnte zusätzlich ein Graustufenbild erstellt werden (Abb. 4.3a). Um zu der automatischen Annotation der Daten zu kommen, wurde aus der Abbildung 4.2b zunächst ein Binärbild erstellt (Abb. 4.3b). Dazu musste der Schwellwert Parameter für die simulierten Bilder gefunden und angepasst werden [1]. Dieser Parameter ist ebenfalls abhängig von der Stärke der verschiedenen Lichteinflüsse im Bild. Aus diesem

Grund wurden die Lichteinflüsse in dem Simulator gleich gehalten. Die nachfolgenden Abbildungen 4.2 und 4.3 zeigen die erstellten Bilder.

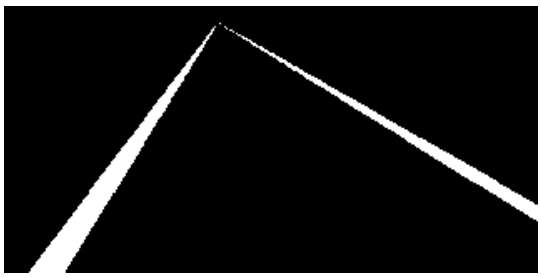
(a) Erste Kamera für die komplette Sicht auf die Fahrbahn (Farbeingabebild).



(b) Zweite Kamera für die Sicht der Fahrbahnmarkierungen.

Abbildung 4.2: Kamerabilder aus dem simulierten autonomen Fahrzeug für die Erzeugung der Datensätze. Bilder aus dem Datensatz 5 (Sim 5).

(a) Umgewandeltes Graustufenbild (Graustufeneingabebild).



(b) Umgewandeltes Binärbild für die Beschriftung und die Eingabe (Binäreingabebild).

Abbildung 4.3: Graustufen- und Binärbild für die Annotation. Bilder aus dem Datensatz 5 (Sim 5).

Um eine gekrümmte Straße zu erkennen, sollten für diese Straße mehrere Orientierungspunkte definiert werden. Aus diesem Grund wurden sieben verschiedene Höhen definiert, um den geraden oder gekrümmten Spurverlauf zu beschreiben. Die definierten Höhen 32, 40, 52, 66, 84, 104, 128 sind bei der Ausrichtung der Kamera wichtig und sind in dieser Arbeit für die simulierte Kamera immer gleich. Diese Höhen werden ebenfalls prozentual auf die verschiedenen Bildauflösungen umgerechnet, um die Ergebnisse zwischen den verschiedenen Größen der Bilder vergleichen zu können. Die Höhen der einzelnen Abschnitte sind unterschiedlich und werden, von unten nach oben betrachtet, kleiner. Dies liegt daran, dass sich die Spurmarkierungen in die „Tiefe“ des Bilds bewegen und die horizontalen Abstände zwischen den Spurmarkierungen kleiner werden. Nach der Festlegung der Höhen, konnten anschließend aus dem erstellten Binärbild die Informationen für den Spurverlauf links und rechts extrahiert werden (Abb. 4.4). Zuerst muss die Mitte der einzelnen Spurmarkierung auf einer bestimmten Höhe links und rechts gefunden werden. Da die Höhen vordefiniert sind (Y-Koordinaten), wird nur die X-Koordinate dieser

Pixel als Annotation für das Eingabebild verwendet. Das Eingabebild kann entweder ein Farbbild (Abb. 4.2a), ein Graustufenbild (Abb. 4.3a) oder ein Binärbild (Abb. 4.3b) sein. Somit ergeben sich in diesem Fall 2 bis 14 Klassen als Ausgabe des faltenden neuronalen Netzwerks. Mehr dazu beschreibt der Abschnitt 4.5. Falls der Spurverlauf links oder rechts auf einer bestimmten Höhe nicht verfügbar ist, wird dieser mit -1 als Annotation in dem Datensatz angegeben. Somit konnten die X-Koordinaten 46, 319, 63, 286, 77, 251, 90, 221, 99, 197, 107, 179, 112 und 166 in Abbildung 4.4 aus der Abbildung 4.3b von unten nach oben extrahiert werden. Diese X-Positionen dienen als eine Annotation für das Farbeingabebild (Abb. 4.2a), für das Graustufeneingabebild (Abb. 4.3a) und für das Binäreingabebild (Abb. 4.3b).

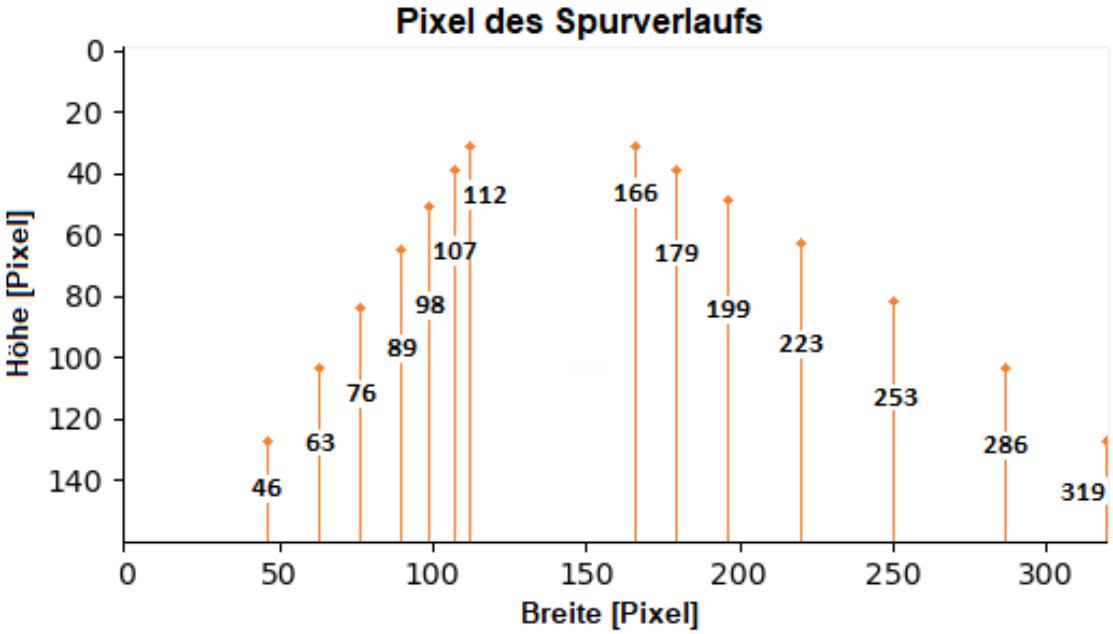
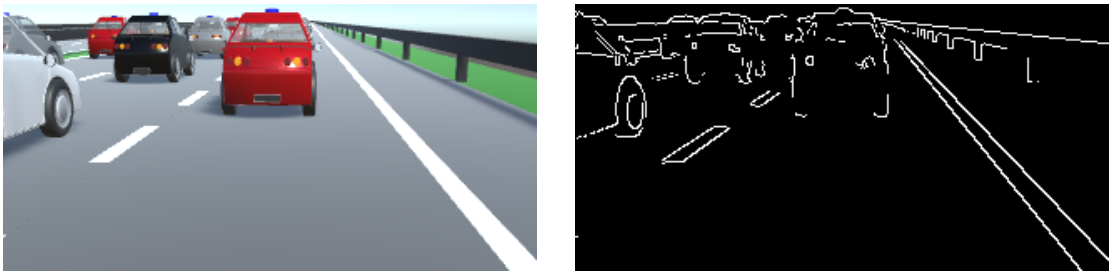


Abbildung 4.4: Diagramm des Spurverlaufs für die automatische Beschriftung (Annotation) der Daten. Orangefarbige Punkte: Mitte der einzelnen Spurmarkierung auf einer vordefinierten Höhe links und rechts inklusive der Annotation aus der Abbildung 4.3b.

4.4 Spurerkennung mit gefilterten Canny-Algorithmus

Der Canny-Algorithmus wird oft für die Kantenerkennung in einem zweidimensionalen Bild verwendet, da er schnell und präzise funktioniert. Mehr dazu wird in dem Abschnitt 2.3.2 beschrieben. In einem Bild in dem eine Fahrspur dargestellt ist, erkennt das menschliche Auge direkt den Spurverlauf. Doch um ein Fahrzeug eigenständig in der Fahrspur halten zu können, müssen diese Linien zuerst erkannt und anschließend gefiltert werden. Aus diesem Grund wird eine Erweiterung für diesen Algorithmus überlegt und ein

eigenes Vorgehen für das Filtern erforscht. Das zuvor publizierte Vorgehen für das Filtern der Linien wird in diesem Abschnitt erweitert und somit die Genauigkeit verbessert [55]. Um die Fahrspur mittels des Canny-Algorithmus ermitteln zu können, muss als erstes aus dem farbigen Eingabebild (Abb. 4.5a) ein Graustufenbild erstellt werden. Aus dem Graustufenbild wird im nächsten Schritt ein Kantenbild mit dem Canny-Algorithmus erstellt. Die nachfolgende Abbildung 4.5b zeigt das erstellte Kantenbild aus dem Farbbild (Abb. 4.5a).



(a) Farbeingabebild.

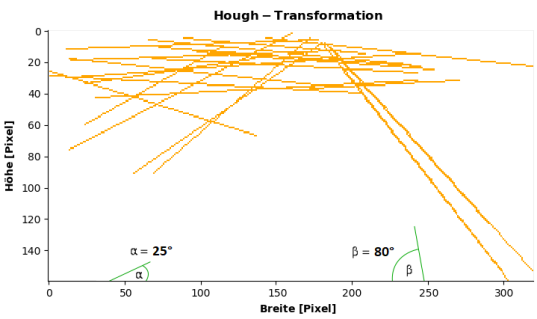
(b) Generiertes Kantenbild mit Canny.

Abbildung 4.5: Generierung des Kantenbilds mit dem Canny-Algorithmus aus dem Datensatz 5 (Sim 5) aus der Simulation in Kapitel 3.

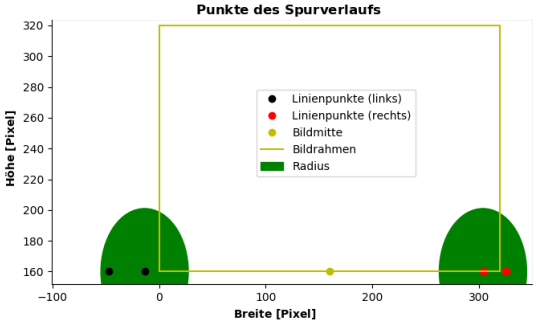
Aus dem erstellten Kantenbild können nun mit der probabilistischen Hough-Transformation die einzelnen Kanten als Linien extrahiert werden. Die Grundlagen zu der Hough-Transformation beschreibt der Abschnitt 2.3.3. Durch diese Transformation bekommt man die Koordinaten aller Linien im Bild als $P(x1, y1)$ und $Q(x2, y2)$ Koordinaten, anhand der für die Transformation gewählten Parameter. Die nachfolgende Abbildung 4.6a zeigt alle gefundenen Linien aus dem Canny-Kantenbild mit der probabilistischen Hough-Transformation dargestellt in einem Diagramm. Für das Filtern wird nun für jede einzelne Linie die Steigung m der Geraden berechnet. Aus dieser Steigung kann anschließend mit der Arkustangensfunktion $atan(m)$ der Winkel θ in Grad zur X-Achse berechnet werden. Nach der Berechnung der Winkel können alle Linien mit einem Winkel unter 25° und über 80° (α und β in Abb. 4.6a) ignoriert werden. Diese Parameter funktionieren bei der verwendeten Ausrichtung und Neigung der Kamera in der Simulation. Es wurde angenommen, dass zwei Meter genau einer Unity Einheit in der Simulation entsprechen. Somit ist die Neigung der Kamera 11° und die Höhe der Kamera von der Fahrbahn $0,67$ Unity Einheiten oder 134 Zentimeter (cm). Wenn die Position der Kamera sich ändert, können die Parameter für die Winkel in diesem Algorithmus angepasst werden.

Für alle restlichen vorgefilterten Linien kann nun der Schnittpunkt mit der X-Achse SX berechnet werden. Von der Bildmitte B können anschließend die linken und rechten Linien in getrennte Datenfelder einsortiert werden. Dabei müssen die linken Linien eine negative Steigung m und die rechten Linien eine positive Steigung m beinhalten. Anderenfalls können diese Linien ignoriert werden. Anschließend werden die Schnittpunkte SX, der Winkel θ, die Steigung m inklusive der P- und Q-Koordinaten der Linien für jeweils linke und rechte Linien gespeichert. Im nächsten Schritt werden die Schnittpunkte der linken und rechten Linien miteinander berechnet und in einem vordefinierten Bereich

zwischen dem Linienschnittpunkt 1 und Linienschnittpunkt 2 akzeptiert. Diese beiden Parameter sind ebenfalls von der Neigung der Kamera abhängig. Der Ursprung dieser gefundenen Linien muss sich ebenfalls in den unteren zwei Dritteln des Bilds befinden. Ansonsten können diese Linien ignoriert werden. Von der Bildmitte B muss im nächsten Schritt der kleinste Abstand $A = |SX - B|$ auf der X-Achse jeweils links und rechts gefunden werden. Anschließend wird in einem vordefinierten Radius R nach weiteren Schnittpunkten SX gesucht, um die Mitte der Spurmarkierung berechnen zu können. Die gefundenen Linien im vordefinierten Radius R müssen einen ähnlichen Winkel zur X-Achse haben. Dafür wird der Winkelschwellwert W mit $|\theta_1 - \theta_2| < W$ verwendet. Der Radius und der Winkelschwellwert sind von der Bildbreite (Pixel) abhängig und können ebenfalls angepasst werden. Die nachfolgende Abbildung 4.6b zeigt bildlich dieses Vorgehen. Aus den gefundenen Linien kann jeweils eine Durchschnittsfunktion links und rechts errechnet werden. Diese Durchschnittsfunktionen sind der Verlauf der Spurmarkierung links und rechts.



(a) Alle gefundenen und ungefilterten Linien inklusive der Fahrspur im Bild. Orange: Gezeichneten Linien nach der probabilistischen Hough-Transformation. Grün: Winkel $\alpha = 25°$ und $\beta = 80°$.

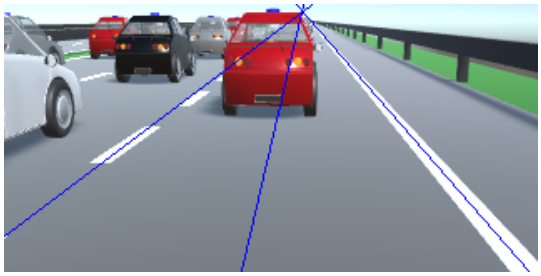


(b) Gefilterte Linien der Fahrspur (SX). Schwarz: Gefundene Punkte der Spur links. Rot: Gefundene Punkte der Spur rechts. Dunkelgrün: Vorbestimmter Radius (R). Hellgrün: Bildgröße und Bildmitte (B).

Abbildung 4.6: Diagramme der erkannten ungefilterten und gefilterten Fahrspur.

Um die Mitte der Fahrspur zu berechnen kann aus diesen beiden Durchschnittsfunktionen die Mitte errechnet werden. Die nächste Abbildung 4.7 zeigt die Fahrspur jeweils vor (Abb. 4.7a) und nach dem Filtern (Abb. 4.7b). Dabei wurde angenommen, dass der Spurverlauf gerade ist. Auf Autobahnen ist es meistens auch der Fall. Die Straße ist meistens nur leicht gekrümmt. Wenn die Straße gerade ist, reichen nur zwei Orientierungspunkte (links und rechts) an einer bestimmten Höhe im Bild, um den Spurverlauf zu ermitteln (Abb. 4.8a). Allerdings sind mehrere Orientierungspunkte auf verschiedenen Höhen notwendig, wenn die Straße gekrümmt ist. Dazu kann das Bild in mehrere Ausschnitte horizontal aufgeteilt werden [117]. Die Anzahl und die Höhe der Ausschnitte kann beliebig gewählt werden. Diese horizontale Aufteilung ist von der Auflösung des Bilds, dem sichtbaren Bildbereich der Kamera und der Ausrichtung der Kamera abhängig. Sobald die Aufteilung erfolgt ist, kann für jeden Ausschnitt die Spurmarkierung jeweils links und rechts als eine gerade Linie mit dem gefilterten Canny-Algorithmus erkannt werden.

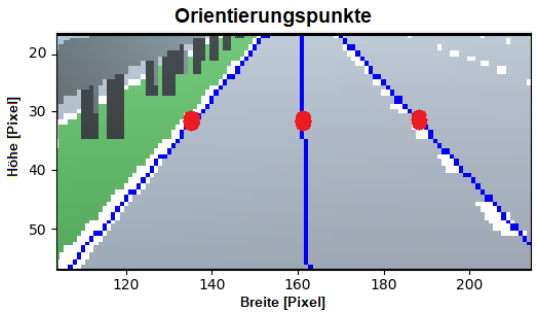
(a) Gezeichnete Linien nach der probabilistischen Hough-Transformation im farbigen Eingabebild.



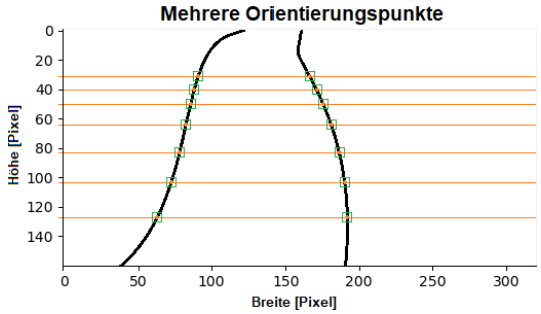
(b) Gefiltertes Farbeingabebild durch die gefilterte Fahrspurerkennung (gefilterter Canny-Algorithmus).

Abbildung 4.7: Ungefilterte und gefilterte Spurerkennung mit dem Canny-Algorithmus und der probabilistischen Hough-Transformation.

Die nachfolgende Abbildung 4.8b beschreibt dieses Vorgehen. Die einzelnen Ausschnitte sind durch die horizontalen orangefarbigen Linien getrennt. Nach der Erkennung der Fahrspur für jeden Ausschnitt erhält man die Spurmarkierung jeweils links und rechts im Bild als P- und Q-Koordinaten. Somit kann die Spurmitte im nächsten Schritt ermittelt werden und der Verlauf der Fahrspur für jeden Abschnitt als eine gerade Linie dargestellt werden.



(a) Gerade Fahrspur. Blau: Erkannte Fahrspur. Rot: Orientierungspunkte.



(b) Gekrümmte Fahrspur. Abschnitte (orange). Orientierungspunkte (grün).

Abbildung 4.8: Orientierungspunkte zur Berechnung des Mittelpunkts der Fahrspur zum Halten des Fahrzeugs in der Fahrspur (Lenkung des Fahrzeugs).

So sieht der gefilterte Canny-Algorithmus auf einen Blick aus:

0. Als Vorarbeit sollte die Kamera so ausgerichtet und kalibriert werden, dass nur die notwendigen Stellen im Bild zu sehen sind (nur die Fahrspur). Zum Beispiel sollten die Motorhaube des Fahrzeugs oder die Stromleitungen am Himmel nicht zu sehen sein. Wenn eine Ausrichtung der Kamera aus welchen Gründen auch immer nicht möglich ist, können die unnötigen Stellen im Bild oben oder unten ebenfalls abgeschnitten werden.

1. Aus dem Farbeingabebild ein Graustufenbild erstellen.
2. Canny-Kantenerkennungsbild aus dem Graustufenbild erstellen.
3. Probabilistische Hough-Transformation auf das Kantenbild anwenden (Ermittlung der 2D-Koordinaten für P und Q).
4. Für jede gerade Linie die Steigung m berechnen.
5. Aus der Steigung m den Winkel θ mit $\theta = atan(m)$ berechnen.
6. Alle Linien mit einem Winkel θ unter $\alpha = 25°$ und über $\beta = 80°$ ignorieren $(\alpha < \theta < \beta)$.
7. Für alle restlichen Linien, die Schnittpunkte mit der X-Achse SX berechnen.
8. Von der Bildmitte B die linken und rechten Linien in getrennte Datenfelder sortieren.
9. Die linken Linien müssen eine negative Steigung m und die rechten Linien eine positive Steigung m beinhalten.
10. Die Schnittpunkte SX, den Winkel θ, die Steigung m und die P- und Q-Koordinaten der Linien für jeweils linke und rechte Linien speichern.
11. Die Schnittpunkte der linken und rechten Linien miteinander berechnen und in einem vordefinierten Bereich zwischen Linienschnittpunkt 1 und Linienschnittpunkt 2 akzeptieren.
12. Der Ursprung dieser Linien muss sich ebenfalls in den unteren zwei Dritteln des Bilds befinden.
13. Von der Bildmitte B den kleinsten Abstand $A = |SX - B|$ jeweils links und rechts herausfinden.
14. In einem vordefinierten Radius R nach weiteren Schnittpunkten SX links und rechts suchen.
15. Die gefundenen Linien im vordefinierten Radius R müssen einen ähnlichen Winkel zur X-Achse enthalten. Dafür wird der Winkelschwellwert W mit $|\theta_1 - \theta_2| < W$ verwendet.
16. Aus den gefundenen Linien eine Durchschnittsfunktion (gerade Linie) links und rechts errechnen.
17. Aus diesen beiden Funktionen links und rechts die Mitte der Fahrspur errechnen, um eine Trajektorie für das Fahrzeug zu bestimmen.

Der 12. Schritt wird bei der Aufteilung des Bilds in sieben verschiedene Ausschnitte übersprungen, da der echte Ursprung dieser Linien in jedem einzelnen Ausschnitt nicht bekannt ist. Für den gefilterten Canny-Algorithmus und die dafür durchgeführte probabilistische Hough-Transformation können ebenfalls experimentell folgende Parameter gefunden werden:

- Canny-Algorithmus
  - Schwellwert 1 (min): 350
  - Schwellwert 2 (max): 400
- Probabilistische Hough-Transformation (Ausschnitte = 1)
  - Schwellwert: 15
  - Minimale Länge der Linien: 20
  - Maximaler zulässiger Abstand zwischen den Linien: 40
- Probabilistische Hough-Transformation (Ausschnitte = 7)
  - Schwellwert: 10
  - Minimale Länge der Linien: 3
  - Maximaler zulässiger Abstand zwischen den Linien: 3
- Filterung der Linien
  - Radius: 14
  - Winkel 1 (α): 25°
  - Winkel 2 (β): 80°
  - Linienschnittpunkt 1 (min): 0
  - Linienschnittpunkt 2 (max): 15
  - Winkelschwellwert: 10

4.5 Spurerkennung mit faltendem neuronalen Netzwerk

Dieser Abschnitt stellt den zweiten Ansatz, eine Spurerkennung mit einem faltenden neuronalen Netzwerk, vor. Um das faltende neuronale Netzwerk optimal trainieren zu können, werden für das überwachte Lernen sehr viele annotierte Datensätze (Trainingsdaten) benötigt. Solche Daten kann man natürlich auch manuell erstellen. Allerdings dauert das Annotieren bei so vielen Datensätzen sehr lange. Je nach Anwendungsfall werden auch unterschiedliche Datensätze benötigt. Die Menge der Trainingsdaten, die Eingabe- und Ausgabedaten sind ebenfalls unterschiedlich. Der CULane Datensatz bietet Daten für die Spurerkennung für die akademische Forschung an [83]. Diese Daten wurden bereits manuell annotiert. Allerdings für die weitere Forschung, um auf die autonomen Modellfahrzeuge umzusteigen, muss die Simulation an den realen Einsatz angepasst sein.

In diesem Fall muss die Kamera in der Simulation die gleiche Auflösung haben und exakt gleich ausgerichtet sein, wie die Kamera im Modellfahrzeug. Aus diesem Grund wurden die bereits vorgestellten Datensätze erstellt und automatisch annotiert. Falls der annotierte Wert für einen bestimmten Spurverlauf in dem Datensatz nicht vorhanden war, wurde dieser Wert durch eine lineare Regression für das faltende neuronale Netzwerk interpoliert. Nach einer langen Zeit des Trainings, der Evaluation und des Testens wurde eine optimale Konfiguration für das ConvNet für die vorgestellten Trainingsdaten gefunden. Das faltende neuronale Netzwerk hat die folgende Netzwerkarchitektur:

- Conv (8, kernel = (5, 5), strides = (2, 2), padding = 'same', activation = 'relu')
- Conv (8, kernel = (3, 3), strides = (1, 1), padding = 'same', activation = 'relu')
- Conv (8, kernel = (3, 3), strides = (1, 1), padding = 'same', activation = 'relu')
- Dropout (0.5)
- Conv (16, kernel = (5, 5), strides = (2, 2), padding = 'same', activation = 'relu')
- Conv (16, kernel = (3, 3), strides = (1, 1), padding = 'same', activation = 'relu')
- Conv (16, kernel = (3, 3), strides = (1, 1), padding = 'same', activation = 'relu')
- Dropout (0.5)
- Conv (32, kernel = (5, 5), strides = (2, 2), padding = 'same', activation = 'relu')
- Conv (32, kernel = (3, 3), strides = (1, 1), padding = 'same', activation = 'relu')
- Conv (32, kernel = (3, 3), strides = (1, 1), padding = 'same', activation = 'relu')
- Dropout (0.5)
- Conv (64, kernel = (5, 5), strides = (2, 2), padding = 'same', activation = 'relu')
- Conv (64, kernel = (3, 3), strides = (1, 1), padding = 'same', activation = 'relu')
- Conv (64, kernel = (3, 3), strides = (1, 1), padding = 'same', activation = 'relu')
- Dropout (0.5)
- Flatten
- Dense (2000, activation = 'relu')
- Dense (1000, activation = 'relu')
- Dense (200, activation = 'relu')
- Dense (14, activation = 'linear')

Die Faltungsschichten des faltenden neuronalen Netzwerks beinhalten die Parameter Output, Kernel, Stride, Padding und Activation. Mehr Informationen zu den faltenden neuronalen Netzwerken gibt der Abschnitt 2.2.2. An dieser Stelle wird die Rectified Linear Unit (ReLU) als Aktivierungsfunktion für alle Schichten verwendet. In der letzten vollständig verbundenen Schicht (engl. Dense) wurde eine lineare Funktion als Aktivierungsfunktion für die Regression verwendet. Auf der letzten Dense-Schicht erkennt man ebenfalls die Anzahl der Ausgabe des neuronalen Netzwerks. Dieser beinhaltet in diesem Beispiel 14 Klassen für die einzelnen X-Koordinaten. Mehr dazu wird in dem Abschnitt 4.3.1 erklärt. Für das Training wurde die Lernrate auf $0,001$ gesetzt. Um die Netzwerkgewichte während des Trainings anhand von Trainingsdaten anzupassen wird der Adam-Optimierer (engl. Adam Optimizer) verwendet. Damit die Modelle anschließend vergleichbar sind, wird als Fehlermaß der mittlere absolute Fehler verwendet. Mehr dazu findet man in dem Abschnitt 2.2.3. Die Vorstellung der Laufzeit und des Fehlers des faltenden neuronalen Netzwerks wird in dem Abschnitt 4.6.2 behandelt.

4.6 Experimente

Die folgenden Experimente wurden durchgeführt, um die Funktionalität mittels der Laufzeit und des Fehlers des gefilterten Canny-Algorithmus und des faltenden neuronalen Netzwerks für die Fahrspurerkennung zu vergleichen. Die Auflösung ist als Breite × Höhe angegeben. Alle Experimente, das Training und die Laufzeitmessungen des ConvNets, Laufzeit- und Fehlerberechnungen vom gefilterten Canny-Algorithmus, werden auf derselben Hardware durchgeführt, um die Ergebnisse hinterher vergleichen zu können. Dafür wird das Dell G3 15 3590 Notebook verwendet. Diese Hardware wird in Abschnitt 2.6.1 genauer vorgestellt. Die Laufzeitmessungen erfolgen auf dem Prozessor und nicht auf der Grafikkarte der verwendeten Hardware. Die Bilder für die Evaluation der jeweiligen vorgestellten Methoden wurden ebenfalls gleich gehalten. Für das Training der ConvNet-Modelle wurde *TensorFlow 2.1.0* und *Keras 2.2.4-tf* verwendet.

4.6.1 Laufzeit und Fehler vom gefilterten Canny-Algorithmus

Bei diesen Experimenten werden die Laufzeit, der Fehler und die Erkennungsrate des gefilterten Canny-Algorithmus auf Simulationsdaten untersucht. Für den gefilterten Canny-Algorithmus wurden die Numpy-Bibliothek verwendet, da diese besonders schnell mit den Datenfeldern funktionieren [61]. Die nachfolgende Gleichung 4.1 zeigt die Zusammensetzung der Laufzeiten des gefilterten Canny-Algorithmus. Die Anzahl der Ausschnitte (s) im Bild orientiert sich an den im Abschnitt 4.3.1 vorgestellten Höhen. Bei einem Bild mit einer Höhe von 160 Pixel entspricht es den Abschnittsgrenzen $0, 32, 40, 52, 66, 84, 104, 128$ und 160, wobei der oberste Ausschnitt im Bild nicht beachtet wird. Dies entspricht den Abschnittsgrenzen 0 bis 32 Pixel. Die Zeitmessung der Bilder aus den verschiedenen Datensätzen ist in Millisekunden (ms) angegeben. Da die Ermittlung der Linien (Spurerkennung) im Bild pro Ausschnitt (s) unterschiedlich ist, wurde die Durchschnittszeit bei der Zeitberechnung bei der Bildung der Ausschnitte gebildet. Für die Berechnung der Gesamtlaufzeit wird die Anzahl der Ausschnitte (s) mit der Zeit

pro Ausschnitt (st) multipliziert. Durch das einmalige Aufteilen (Schnitt) des Bilds in verschiedene Ausschnitte wird die Zeit für das Aufteilen (ct) nur einmal berechnet und anschließend aufaddiert. Somit ergibt sich in der Tabelle 4.2 die Gesamtlaufzeit (t) aus:

$$t = st \times s + ct \tag{4.1}$$

Auf jeder angegebenen Höhe sind zwei Orientierungspunkte, einer für die linke und einer für die rechte Spurmarkierung vorgesehen. Somit kann die Anzahl der Ausschnitte 1 und 7 mit den im nächsten Abschnitt angegebenen Klassen 2 und 14 für das faltende neuronale Netzwerk gleichgesetzt werden. Die verschiedenen Experimente mit der verschiedenen Auflösung der Bilder haben ebenfalls gezeigt: Je größer die Bildauflösung in Pixel ist, desto genauer ist auch die Erkennung der Fahrspur. Allerdings dauert die Findung der Fahrspur in diesem Bild auch länger (Vergleich zwischen Exp. Nr. 1 und 4 in Tab. 4.2). Verdoppelt sich die Auflösung des Bilds, so verdoppelt sich auch die Laufzeit für die Findung der Fahrspur im Bild (Vergleich zwischen Exp. Nr. 2 und 4 in Tab. 4.2).

Tabelle 4.2: Laufzeit, Fehler und Erkennungsrate des gefilterten Canny-Algorithmus für die Spurerkennung. Die erste Spalte enthält die Identifikationsnummer (ID) des Experiments (Exp. Nr.). Der Fehler ist als mittlerer absoluter Fehler prozentual der Bildbreite angegeben.

Exp. Nr.	**Daten-satz**	**Auflösung [Pixel]**	**Ausschnitte (s)**	**Fehler [%]**	**Erkennungs-rate [%]**	**Laufzeit (t) [ms]**
1	Sim 1	160 × 80	1	0,50	99,68	1,19
2	Sim 2	320 × 160		0,32	99,95	1,68
3	Sim 3	640 × 160		0,15	99,68	2,94
4	Sim 4	640 × 320		0,28	99,99	3,44
5	Sim 5	320 × 160		0,48	98,46	2,03
6	Sim 6	320 × 160		0,40	99,19	1,79
7	Sim 1	160 × 80	7	1,42	34,85	1,33
8	Sim 2	320 × 160		0,82	59,73	2,91
9	Sim 3	640 × 160		0,47	57,77	4,06
10	Sim 4	640 × 320		0,57	84,67	5,27
11	Sim 5	320 × 160		1,25	53,58	3,00
12	Sim 6	320 × 160		1,02	56,53	2,95

In der Tabelle 4.2 ist der Fehler inklusive der Erkennungsrate angegeben. Anders als bei der Erkennung der Fahrspur mit dem faltenden neuronalen Netzwerk, gibt es bei der Erkennung der Spur mit dem Canny-Algorithmus bei der Nichterkennung der einzelnen Linie keinen Wert für einen Vergleich. Somit wird die Erkennungsrate, berechnet aus der Anzahl der einzelnen erkannten Fahrspuren geteilt durch die Gesamtanzahl der Spuren und multipliziert mit 100, angegeben. Aus allen einzelnen erkannten Spuren kann anschließend der durchschnittliche Fehler in Prozent ausgerechnet werden. An dieser Stelle wird der mittlere absolute Fehler verwendet. Dieser Fehler wird prozentual auf

die verschiedenen Bildbreiten umgerechnet, um die Ergebnisse zwischen den einzelnen Auflösungen vergleichen zu können. Das erste Ausführen des Algorithmus auf einem System ist etwas langsamer, da die notwendigen Komponenten zuerst geladen werden müssen [55]. Durch das mehrmalige Ausführen der Algorithmen hintereinander, können somit bessere Laufzeiten erreicht werden. Das Betriebssystem verwendet einen schnellen Pufferspeicher, um die Zugriffe auf die notwendigen Bibliotheken zu optimieren [95]. Wie in der Tabelle 4.2 in Exp. Nr. 2 zusehen ist, erreicht der Algorithmus bei der Auflösung 320 × 160 Pixel ein sehr gutes Gleichgewicht zwischen der Genauigkeit und der Geschwindigkeit. Aus diesem Grund wurde die Auflösung 320 × 160 Pixel als die optimale Auflösung für die Spurerkennung für die vorgestellten Datensätze ausgewählt. Bei einem geraden Spurverlauf dauert die Auswertung für das einzelne Bild ca. 1,68 Millisekunden (Exp. Nr. 2 in Tab. 4.2). Dabei wird ebenfalls eine Erkennungsrate von 99,95 % und ein Fehler von 0,32 % erreicht. Bei einem ungeraden Spurverlauf hingegen dauert die Auswertung für das einzelne Bild ca. 2,91 Millisekunden (Exp. Nr. 8 in Tab. 4.2) auf der verwendeten Hardware. An dieser Stelle sinkt die Erkennungsrate auf 59,73 % und der Fehler steigt auf 0,82 %. Die Verschlechterung der Erkennungsrate und des Fehlers ist hierbei auf die Erhöhung der zu erkennenden Orientierungspunkte zurückzuführen. Zusätzlich kommt es vor, dass die Lücke der mittleren Spurmarkierungen genau in einen Ausschnitt im Bild fällt. Dadurch können die Orientierungspunkte in diesem Ausschnitt nicht bestimmt werden.

4.6.2 Laufzeit und Fehler vom faltenden neuronalen Netzwerk

Diese Experimente stellen die Laufzeit und den Fehler des faltenden neuronalen Netzwerks gegenüber der Auflösung des Eingabebilds vor. Als erstes muss für das ConvNet das optimale Eingabebild gefunden werden. Dazu stehen Farbbilder, Graustufenbilder und Binärbilder in dem generierten Datensatz zur Verfügung. Um sich für eins dieser Eingabebilder zu entscheiden, muss das faltende neuronale Netzwerk mit diesen Bildern trainiert werden und die Ergebnisse anschließend evaluiert und verglichen werden. Durch das Mischen (engl. Shuffle) der Datensätze und die verschiedenen Anfangsgewichte des faltenden neuronalen Netzwerks ist jeder Trainingsdurchlauf unterschiedlich. Hierbei wird aus jeweils zehn Trainingsdurchläufen das beste Modell ausgewertet. Die Verteilung der Trainings-, Evaluierungs- und Testdaten ist 70 %, 15 % und 15 %. Anders als bei der Spurerkennung mit dem gefilterten Canny-Algorithmus gibt es bei der Vorhersage der Fahrspur mit dem faltenden neuronalen Netzwerk keine Berechnung der Erkennungsrate. Durch die Regression des neuronalen Netzwerks gibt es immer eine Vorhersage zu den angegebenen 2 oder 14 Ausgabeknoten. Aus dieser Vorhersage kann anschließend der Fehler in Prozent jeweils pro Klasse angegeben werden. Um die Fehler auf verschiedenen Bildergrößen vergleichen zu können, wird dieser Fehler auf die Bildbreite prozentual umgerechnet. Wie bereits erwähnt wird als Fehlermaß der mittlere absolute Fehler errechnet. Die gemessenen Laufzeiten beinhalten nicht nur die Zeit für die Vorhersage des faltenden neuronalen Netzwerks [55]. Die gemessene Laufzeit in der Tabelle 4.3 enthält zusätzlich die Zeit für das notwendige Expandieren, Multiplizieren und das Runden der Werte der Vorhersage. Durch einige Vorexperimente konnte man erkennen, dass die mit

den Binärbildern trainierten Modelle bessere (kleinere) Fehlerraten liefern. Das ist sowohl für die Modelle mit 2 Klassen als auch für die Modelle mit 14 Klassen der Fall. Allerdings sind diese Modelle nur in der Simulation einsetzbar. Die automatische Erstellung eines Binärbilds aus den realen Bildern, zum Beispiel aus dem Modellbaubereich, ist nicht so einfach möglich. Durch die verschiedenen Lichteinflüsse im Bild muss der Schwellwert für die Erstellung der verschiedenen Binärbilder immer wieder gefunden und angepasst werden. Das ist bei den simulierten Bildern nicht der Fall. Somit werden bei diesen Experimenten nur die Farbbild- und Graustufenbild-Modelle betrachtet.

Tabelle 4.3: Laufzeit und Fehler des faltenden neuronalen Netzwerks (ConvNet) für die Spurerkennung. Die erste Spalte enthält die Identifikationsnummer (ID) des Experiments (Exp. Nr.). Der Fehler ist als mittlerer absoluter Fehler prozentual der Bildbreite angegeben.

Exp. Nr.	**Datensatz**	**Auflösung [Pixel]**	**Klassen**	**Modell**	**Fehler [%]**	**Laufzeit (t) [ms]**
1	Sim 1	160 × 80	2	Grau	0,56	20,94
2				Farbe	0,81	20,86
3	Sim 2	320 × 160		Grau	1,58	28,33
4				Farbe	1,63	28,95
5	Sim 3	640 × 160		Grau	0,81	37,47
6				Farbe	1,02	37,65
7	Sim 4	640 × 320		Grau	2,67	50,13
8				Farbe	1,15	50,63
9	Sim 5	320 × 160		Grau	2,06	27,98
10				Farbe	1,30	28,14
11	Sim 6	320 × 160		Grau	0,93	27,88
12				Farbe	1,07	29,02
13	Sim 1	160 × 80	14	Grau	0,95	20,96
14				Farbe	1,02	20,47
15	Sim 2	320 × 160		Grau	0,94	28,22
16				Farbe	0,80	28,89
17	Sim 3	640 × 160		Grau	1,26	37,53
18				Farbe	1,41	37,61
19	Sim 4	640 × 320		Grau	1,66	50,04
20				Farbe	2,03	50,70
21	Sim 5	320 × 160		Grau	2,18	27,64
22				Farbe	2,14	28,10
23	Sim 6	320 × 160		Grau	1,41	27,87
24				Farbe	1,47	28,69

Ebenfalls, haben die verschiedenen Experimente gezeigt, dass die Modelle mit dem Graustufenbild als Eingabe meistens bessere Trainingsergebnisse erzielen als mit dem Farbbild. Je nach Anwendungsfall, kann dieses Ergebnis bestätigt werden [10]. Dieses

Ergebnis hängt ebenfalls von der Art der Bilder, die klassifiziert werden sollen, ab [121]. Sobald mehr Informationen aus den Bildern benötigt werden, kann ein Farbbild teilweise bessere Ergebnisse erzielen als ein Graustufenbild. Die durchgeführten Experimente zeigen, dass das Graustufenbild-Modell mit 2 Klassen und der 160 × 80 Pixelauflösung den kleinsten Fehler liefert und somit auch genauer ist gegenüber anderen Modellen mit 2 Klassen (Exp. Nr. 1 in Tab. 4.3). Bei 14 Klassen wird das beste Ergebnis mit dem Farbbild-Modell und der 320 × 160 Pixelauflösung erreicht (Exp. Nr. 16 in Tab. 4.3). Nach den verschiedenen Trainingsvorgängen der Modelle, kann nun die Laufzeit für die Erkennung der Fahrspur gemessen werden. Die Laufzeitmessungen der Modelle mit verschiedenen Auflösungen sind ebenfalls in der Tabelle 4.3 angegeben. Die Klassen 2 und 14 in Tabelle 4.3 entsprechen genau den Ausschnitten 1 und 7 in der Tabelle 4.2 aus den Experimenten mit dem gefilterten Canny-Algorithmus für die Spurerkennung (Abschn. 4.6.1). Bei diesen Experimenten ist ebenfalls zu sehen, dass die Anzahl der Klassen keine Auswirkung auf die Laufzeit hat (Vergleich zwischen Exp. Nr. 1 und 13 in Tab. 4.3). Die Modelle mit den Farbbildern als Eingabe benötigen gegenüber den Graustufenbildern als Eingabe mehr Laufzeit, um eine gewünschte Ausgabe zu erzielen (Vergleich zwischen Exp. Nr. 3 und 4 in Tab. 4.3). Das ist darauf zu führen, dass die Graustufenbilder nur einen Farbkanal haben und die Farbbilder drei Farbkanäle (rot, grün und blau - RGB) besitzen. Die Laufzeit erhöht sich ebenfalls bei größeren Bildern gegenüber kleineren Bildern als Eingabe (Vergleich zwischen Exp. Nr. 1 und 7 in Tab. 4.3). Es liegt daran, dass der Kernel der Faltungsschichten des faltenden neuronalen Netzwerks mehr Pixel in dem Eingabebild durchläuft. Ebenfalls hat der Fehler des trainierten Modells keine Auswirkung auf die Laufzeit (Vergleich zwischen Exp. Nr. 7 und 8 in Tab. 4.3).

4.6.3 Auswertung der Ergebnisse

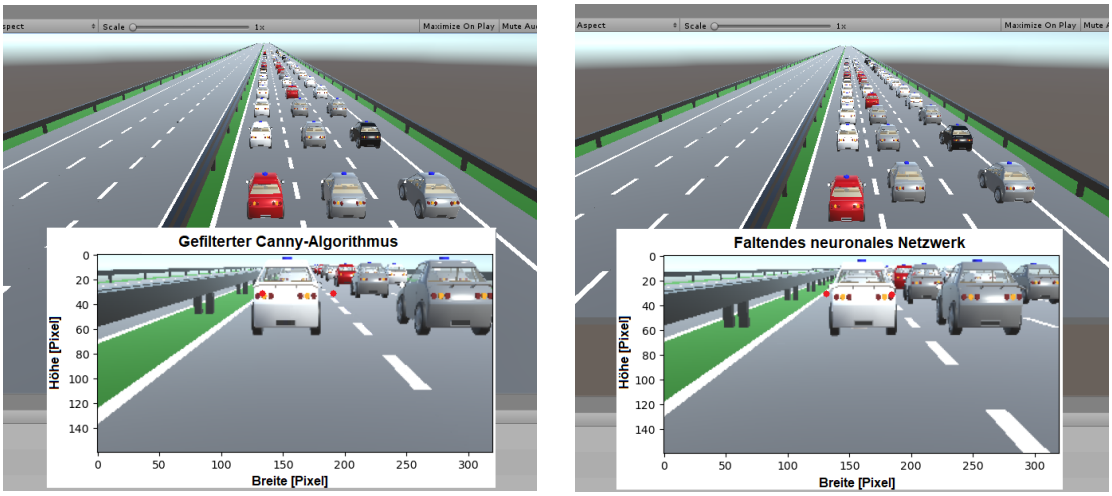
Nachdem die Tests der Performance für die jeweiligen Ansätze, gefilterter Canny-Algorithmus und faltendes neuronales Netzwerk für die Spurerkennung, abgeschlossen sind, können diese beiden Vorgehensweisen verglichen und ausgewertet werden. Der gefilterte Canny-Algorithmus benötigt ca. $1,68$ Millisekunden (ms) in einem 320×160 Pixel Eingabebild, um die gerade Fahrspur auf der in diesem Kapitel verwendeten Hardware zu erkennen. Dieser Algorithmus erreicht ebenfalls eine sehr gute Erkennungsrate von $99,95$ % und enthält einen sehr geringen Fehler von $0,32$ % auf dem Datensatz 2 (Sim 2) (Exp. Nr. 2 in Tab. 4.2). Auf dem Datensatz 5 (Sim 5) steigt der Fehler auf $0,48$ % und die Erkennungsrate sinkt auf $98,46$ %. Die Laufzeit steigt zusätzlich auf $2,03$ Millisekunden (Exp. Nr. 5 in Tab. 4.2). Dieses Ergebnis ist ebenfalls zu erwarten, da der Datensatz 5 die Bilder inklusive verschiedener Fahrzeuge auf der Autobahn enthält. Diese Fahrzeuge verdecken teilweise die Fahrspur und verursachen mehr Linien im Bild, die zusätzlich erkannt und gefiltert werden müssen. Bei einem Eingabebild mit einer Breite von 320 Pixel entspricht der mittlere absolute Fehler von $0,48$ % ca. 2 Pixel. Dieser ist für den Zweck dieser Forschungsarbeit völlig akzeptabel.

Einige der Frontkameras, die in den selbstfahrenden Autos in der Automobilindustrie verbaut werden, können 1080p Videos mit bis zu 60 Bilder pro Sekunde (engl. Frames Per Second - FPS) aufnehmen [107]. Bei einer Videoaufnahme von 30 FPS und der

Ausführung des gefilterten Canny-Algorithmus in einem Ausführungsprozess sinkt die tatsächliche Bildfrequenz, da pro Bild zusätzlich $2,03$ Millisekunden für die Bestimmung der Fahrspur notwendig sind (1,73 ms $\times$ 30 = 60,9 ms). Um 30 Bilder zu verarbeiten, benötigt das System ca. $1,061$ Sekunden. Zurückgerechnet auf die Bildrate, werden um die $28,3$ FPS erreicht. Sobald bei der Ausführung mit Multithreading gearbeitet wird, können zum Beispiel zwei verschiedene Ausführungsprozesse die Laufzeit erhöhen. Dafür kann sich eine Warteschlange als Datenstruktur eignen, die von einem Ausführungsprozess mit Bildern gefüllt wird. Der zweite Ausführungsprozess führt parallel den gefilterten Canny-Algorithmus auf diesen Bildern aus und leert somit die Warteschlange. Dabei ist bei einer Videoaufnahme von 30 Bilder pro Sekunde die Laufzeit der Algorithmen unter $33,\overline{3}$ ms synchron (Echtzeit). So können 30 FPS beibehalten werden. Das ConvNet benötigt $28,33$ ms, um die Fahrspur in einem 320×160 Pixel Graustufenbild zu bestimmen. In einem Ausführungsprozess sind ca. 850 ms zusätzlich pro 30 Bilder notwendig (28,33 ms $\times$ 30). Dadurch sinkt die tatsächliche Bildfrequenz auf ca. 16 FPS. Bei der Ausführung mit Multithreading können die Modelle bis zu einer Eingabegröße von 320×160 Pixel in Echtzeit auf der vorgestellten Hardware ausgeführt werden. Durch die Auswertung der Ergebnisse ergeben sich folgende Vorteile für den Ansatz mit dem gefilterten Canny-Algorithmus:

- Um ca. Faktor 16 schnellere Erkennung der Fahrspur auf der in diesem Kapitel verwendeten Hardware.
- Kein Training des künstlichen neuronalen Netzwerks.
- Keine Trainingsdaten für das trainieren des Modells notwendig.
- Nach Anpassung der Parameter einsatzbereit für verschiedene Ausrichtungen der Kamera.
- Wenige Daten für die Anpassungen der Parameter ausreichend.

Somit eignen sich die beiden vorgestellten und getesteten Vorgehensweisen für die Spurerkennung in einem zweidimensionalen Bild. Bei einer Geschwindigkeit von 100 km/h legt das Fahrzeug ca. 28 Meter in der Sekunde zurück. Bei 28 FPS könnte die Fahrspur jeden Meter bestimmt werden. Beide Ansätze können ebenfalls erfolgreich in dem entwickelten Simulator aus Kapitel 3 getestet werden. Dieses zeigt die nachfolgende Abbildung 4.9. Die Diagramme der Fahrspur in Abbildung 4.9a und 4.9b sind jeweils aus der Frontkamera des ersten roten simulierten autonomen Fahrzeugs im Bild. Die roten Punkte in den beiden Diagrammen zeigen die erkannten Orientierungspunkte der Fahrspur. Die Erkennung der Fahrspur mit dem gefilterten Canny-Algorithmus erfolgt auf dem gesamten Eingabebild (1 Ausschnitt). Die Fahrspurerkennung mit dem faltenden neuronalen Netzwerk wird mit dem Graustufenbild-Modell mit zwei Klassen durchgeführt. Die Auflösung der Eingabebilder beträgt 320×160 Pixel. Somit gibt es jeweils zwei Orientierungspunkte für die Lenkung des Fahrzeugs an der vordefinierten Höhe 32 für die beiden Methoden.



(a) Gefilterter Canny-Algorithmus.

(b) Faltendes neuronales Netzwerk (ConvNet).

Abbildung 4.9: Ausführung der vorgestellten Ansätze für die Spurerkennung im vorgestellten Simulator aus Kapitel 3. Die roten Punkte sind die erkannten Orientierungspunkte der Fahrspur.

4.7 Schlussfolgerungen

Für die Erkennung der Fahrspur wurden zwei Methoden vorgestellt, evaluiert und getestet. Bei der ersten Methode handelt es sich um den Canny-Algorithmus, der in diesem Kapitel um einen Filter erweitert wurde. Dieser wurde der gefilterte Canny-Algorithmus benannt. Bei diesem Ansatz wurden die Linien aus einem Canny-Kantenerkennungsbild durch die probabilistische Hough-Transformation extrahiert und unter bestimmten Parametern in einer bestimmten Reihenfolge gefiltert. Wie die Experimente zeigen, ist dieses Vorgehen sowohl für gerade als auch für gekrümmte Spurverläufe einsetzbar. Bei gekrümmten Straßen kann das Eingabebild in mehrere Ausschnitte horizontal aufgeteilt werden. Auf diese einzelnen Ausschnitte kann jeweils der gefilterte Canny-Algorithmus angewendet werden, um die Fahrspur auf einer Fahrbahn zu finden. Die zweite Methode wurde mit einem faltenden neuronalen Netzwerk realisiert, welches trainiert wurde um den Spurverlauf erkennen zu können. Dabei wurden sieben Höhen (vertikal) an verschiedenen Stellen in einem zweidimensionalen Eingabebild definiert. An diesen Höhen wurden die X-Koordinaten für die Fahrspur links und rechts abgegriffen und für das Training des ConvNets verwendet. Somit hat das faltende neuronale Netzwerk 14 Ausgabewerte, die die Fahrspur auf sieben verschiedenen Höhen beschreiben. Die gefundene Netzstruktur wurde ebenfalls vorgestellt. Auch die Beschaffung von automatisch annotierten Trainingsdaten aus der Simulation wurde präsentiert. Beide vorgestellten Methoden eignen sich sowohl von der Qualität, als auch von der Performance für die Erkennung der Fahrspur auf der vorgestellten Hardware in einem zweidimensionalen Bild. Für die beiden vorgestellten Vorgehensweisen wurden ebenfalls einige Experimente durchgeführt, um Laufzeit- und Genauigkeitsvergleiche zwischen diesen Ansätzen aufstellen zu können. Dadurch kann ein

optimales Gleichgewicht zwischen der Laufzeit und der Genauigkeit gefunden werden. Somit kann diese Spurerkennung zum Beispiel auch auf IoT-Hardware mit beschränkten Ressourcen eingesetzt werden. Wird ein schnelles System benötigt, so muss vielleicht etwas auf die Genauigkeit verzichtet werden. Abschließend lässt sich sagen, dass die zuvor gestellte Forschungsfrage: *Bieten die traditionellen Methoden eine ausreichende Funktionalität angewendet auf eine bildliche Spurerkennung oder muss hierbei auf die künstlichen neuronalen Netzwerke ausgewichen werden?* positiv beantwortet werden kann.

Diese beiden vorgestellten Methoden für die Spurerkennung können nicht nur auf die Bilder aus der Simulation angewendet werden. Nach einigen Anpassungen der Parameter sind diese Methoden auch für den Echtwelteinsatz geeignet. Wie bereits angekündigt, ist das Ziel dieser Forschungsarbeit erfolgreich die erforschten Algorithmen von der Simulation aus Kapitel 3 auf die echten Modellfahrzeuge zu übertragen. Das heißt, die simulierten Algorithmen werden komplett auf einem echten Modellfahrzeug angewendet. Dadurch kann das Verhalten der Fahrzeuge in der Simulation mit dem Verhalten der Modellfahrzeuge in der Realität verglichen werden. Ein wichtiger Aspekt auf den Autobahnen ist die Bildung einer Rettungsgasse für die Rettungsdienste bei einem Unfall. Spannend ist zu sehen, ob die autonomen Modellfahrzeuge anschließend nach der Übertragung der Algorithmen in der Lage sind eine Rettungsgasse für die Polizei- und Rettungsfahrzeuge, bei ähnlichen Szenarien wie zuvor in der Simulation, zu bilden. So können ebenfalls die in Abschnitt 3.4 entwickelten Algorithmen für die Bildung der Rettungsgasse auf Robustheit überprüft werden. Im nachfolgenden Kapitel 5 wird die beschriebene Übertragung von der Simulation auf die Realität durchgeführt.

Die schwierigste Brücke ist die zwischen Menschen.

Hermann Lahm
1948 –

Kapitel 5

Übertragung von Simulation auf Realität

5.1 Einführung

Um die Übertragung von der Simulation auf die Realität durchzuführen und die entwickelten Algorithmen für die Rettungsgassenbildung aus Kapitel 3 in der Praxis zu testen, werden in diesem Kapitel mehrere Modellfahrzeuge inklusive der Konstruktion und der Hardware vorgestellt. Diese Modellfahrzeuge wurden ebenfalls für den Zweck dieser Forschungsarbeit angepasst und erweitert. Bekanntlich unterscheidet sich die Theorie teilweise von der Praxis. Somit werden die in der Simulation entwickelten und getesteten Algorithmen ebenfalls in dem Modellbaubereich getestet. Durch die beschränkte und stromsparende Hardware der Modellfahrzeuge gibt es ebenfalls bei der Ausführung der Algorithmen Einschränkungen in der Geschwindigkeit. Der Ansatz dieser Arbeit ist es, ein Gleichgewicht zwischen der Genauigkeit und der Geschwindigkeit der entwickelten Vorgehensweisen zu finden. Somit steht die Entwicklung von Software auf Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte im Vordergrund.

Zusätzlich gibt es in der Simulation weitere Hilfsmittel, die das Fahrzeug ohne bildliche Auswertung in der Fahrspur halten. In der Realität gibt es diese Hilfsmittel nicht. So wird in dem Abschnitt 5.4 ebenfalls die in Kapitel 4 entwickelte Spurerkennung für den Simulator aus Kapitel 3 auf die echten Modellfahrzeuge übertragen. Bei einem erfolgreichen Übertrag der entwickelten Algorithmen für die Spurerkennung und für die Bildung der Rettungsgasse, verhält sich das Modellfahrzeug in der echten Umgebung ähnlich wie zu vor in der simulierten Umgebung.

5.2 Zusammenbau der Modellfahrzeuge

Für den Zusammenbau der Modellfahrzeuge aus dem Modellbaubereich wurde der Bausatz *SunFounder PiCar-S Smart Car* für den Raspberry Pi verwendet [96]. Dieser enthält bereits mehrere Motoren, Räder, Kontrollplatten und Sensoren für die Entwicklung (Abb. 5.1a). Die Kosten für jeden einzelnen Bausatz belaufen sich zum Zeitpunkt der Entwicklung auf ca. 90 Euro. Der abgebildete Raspberry Pi ist in dem Bausatz nicht enthalten. Eigene Messungen zeigen, dass das Modellfahrzeug eine Geschwindigkeit von $0,6154$ Meter pro Sekunde (m/s) beziehungsweise $2,22$ Kilometer pro Stunde (km/h) erreicht. Der *SunFounder PiCar-S Smart Car* Bausatz wurde anschließend für den Einsatz in dieser Forschungsarbeit angepasst und erweitert. Die nachfolgende Abbildung 5.1b zeigt zum Vergleich das angepasste und erweiterte Modellfahrzeug. Mehr dazu wird in dem Abschnitt 5.2.2 beschrieben.

(a) Original [96]. (b) Umbau.

Abbildung 5.1: Originale und umgebaute *SunFounder PiCar-S Smart Car* Bausatz für Raspberry Pi.

5.2.1 Konstruktion

Die Karosserie dieser Modellfahrzeuge besteht aus mehreren gestanzten Acrylplatten, die anschließend mit einigen Schrauben zusammengehalten werden. Dadurch ergeben sich leichte Ungenauigkeiten bei der Lenkung der Modellfahrzeuge. Für die Lenkung der Modellfahrzeuge ist ein Schrittmotor zuständig. Die Lenkung kann statt in 1°-Schritten nur in minimal 2°-Schritten durchgeführt werden. Wenn der Schrittmotor in 1°-Schritt bewegt wird, bewegen sich die Vorderräder nicht. Dies ist auf die Ungenauigkeiten bei der Konstruktion zurückzuführen. Aus diesem Grund erkennt man beim autonomen Fortbewegen der Modellfahrzeuge leichtes Schwanken auf der Fahrbahn. Ebenfalls sind

die Räder nicht orthogonal zur Fahrbahn ausgerichtet. Dies ist auf die kostengünstige Konstruktion der Karosserie zurückzuführen. Für den Zweck dieser Forschungsarbeit sind diese Modellfahrzeuge völlig ausreichend. Es konnten ebenfalls alle notwendigen Experimente mit den Modellfahrzeugen durchgeführt und vorgestellt werden. Diese Experimente sind in Abschnitt 5.6 beschrieben.

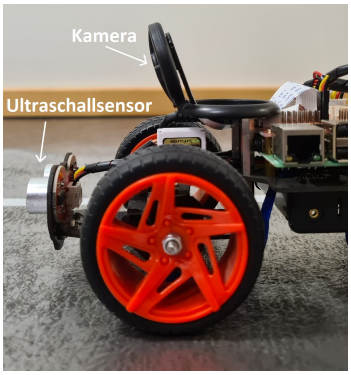
5.2.2 Hardware, Anpassungen und Erweiterungen

Wie bereits erwähnt, wurde für die Modellfahrzeuge der *SunFounder PiCar-S Smart Car* Bausatz für Raspberry Pi verwendet. Dieser enthält bereits den größten notwendigen Teil der Hardware, um ein autonomes Modellfahrzeug zu entwickeln. Der verbaute Ultraschallsensor ist ziemlich genau und schwankt, laut der Dokumentation von SunFounder, nur um 3 Millimeter (mm) [97]. Dies kann ebenfalls bei der alleinigen Ausführung des Prozesses der Abstandsmessung bestätigt werden. Sobald andere Prozesse, wie die Spurerkennung durch eine Kamera, auf dem Raspberry Pi gleichzeitig ausgeführt werden, wird dieser Sensor ziemlich ungenau. Dies liegt teilweise an der mitgelieferten Software für das HC-SR04 Modul. Für die Entfernungsmessung muss zwischen dem Senden und Empfangen einer Schallwelle eine ganz genaue Zeit abgewartet werden. Somit darf dieser Prozess nicht in den Zustand des Schlafens versetzt werden, da sonst während dieser Zeit ein anderer Prozess von dem Betriebssystem ausgeführt werden kann. Somit wurde für den Prozess der Entfernungsmessung ein aktives Warten (engl. Busy Wait) implementiert und die mitgelieferte Software angepasst, um die Messfehler bei der Entfernungsmessung geringer zu halten. Dieses Vorgehen hat das Schwanken bei der Entfernungsmessung etwas verbessert.

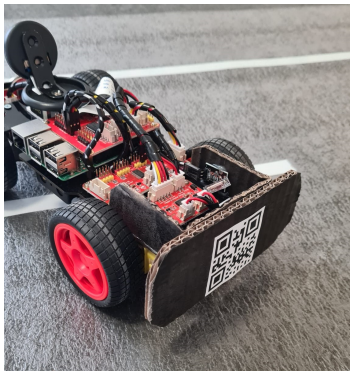
Zusätzlich ist der horizontale Erfassungsbereich dieses Ultraschallsensors nicht wie angegeben. Laut der Dokumentation von SunFounder ist der horizontale Erfassungsbereich des Sensors etwa 30° bis 40° [97]. Laut der durchgeführten Überprüfungen ergibt sich ein horizontaler Erfassungsbereich von etwa 13°. Dieser Erfassungsbereich wird ebenfalls in der Dokumentation des HC-SR04 Sensormoduls mit 15° bestätigt [24]. Für die Bildung der Rettungsgasse beim stockenden Verkehr ist dieser Erfassungsbereich ausreichend, da die Fahrzeuge genügend Sicherheitsabstand halten und somit die Rettungsgasse, ohne stark zu lenken, gebildet werden kann. Die Rettungsgassenbildung bei einem stehenden Verkehr kann mit diesem horizontalen Erfassungsbereich ebenfalls durchgeführt werden, da die autonomen Modellfahrzeuge im ersten Schritt nach hinten fahren, um die Entfernung für die Bildung der Rettungsgasse zu vergrößern. Viel wichtiger ist an dieser Stelle die Präzision des Sensors. Dieser Ultraschallsensor hat teilweise falsche Eingangsimpulse in der Umgebung wahrgenommen, die zu einer falschen Entfernungsmessung führten. Dies kann ebenfalls bestätigt werden [25].

Aus diesem Grund wurde ein zweiter Ultraschallsensor bei dem Hersteller SunFounder nachbestellt. Dieser nachgelieferte Sensor hatte überraschenderweise einen horizontalen Erfassungsbereich von den in der Dokumentation angegebenen 30°. Doch die Präzision der Sensoren wurde damit nicht verbessert. Es wurden weiterhin teilweise falsche Eingangsimpulse in der Umgebung wahrgenommen. Somit wurden die Experimente in Abschnitt 5.6 mehrfach wiederholt und das beste Ergebnis festgehalten. Im nächsten Schritt wurde

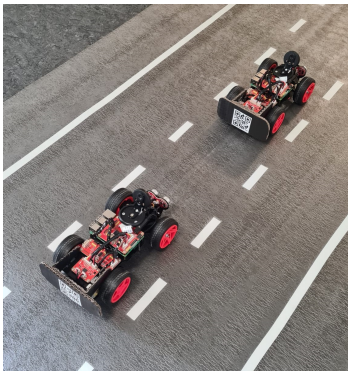
dieser Bausatz um einen Raspberry Pi 3 B und um ein Raspberry Pi V2 Kameramodul erweitert. So konnte die Fahrspur anschließend mit dem Modellfahrzeug bildlich erkannt werden. Die Kamera wurde im ersten Schritt in ein passendes Kameragehäuse verbaut. Anschließend wurde dieses Gehäuse mit zwei Abstandshaltern befestigt und auf einer bestimmten Höhe von der Fahrbahn in einem bestimmten Winkel zur Fahrbahn fixiert (Abb. 5.2a).



(a) Ultraschallsensor nach unten verlegt und Kamera verbaut.



(b) Schwarze Stoßstange aus Pappe mit QR-Code angebracht.



(c) Messung des Abstands zum vorderen Modellfahrzeug durch Ultraschall.

Abbildung 5.2: Anpassungen und Erweiterungen des *SunFounder PiCar-S Smart Car* Bausatzes für Raspberry Pi.

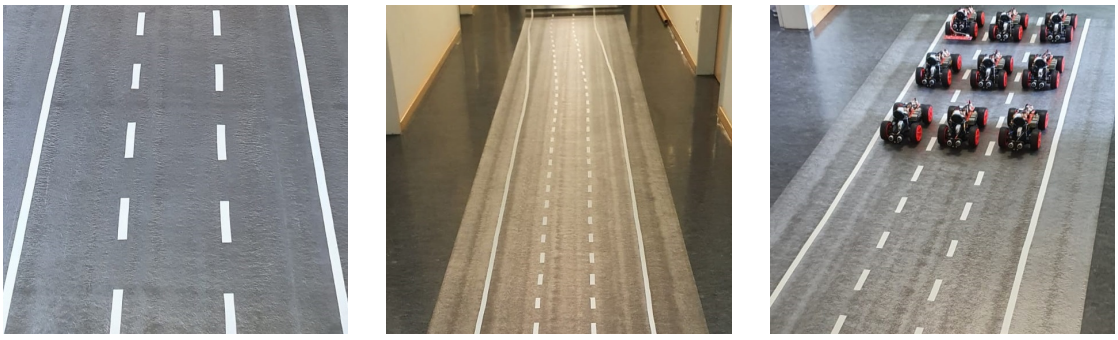
Der Ultraschallsensor für die Abstandsmessung wurde nach unten verlegt, um die Sicht der Kamera nicht einzuschränken. Damit die Ultraschallsensoren eine gerade Fläche zum Messen des Abstands zum vorderen Modellfahrzeug haben (Abb. 5.2c), wurde bei jedem Modellfahrzeug eine schwarze Stoßstange aus Pappe angebracht. Die Abbildung 5.2b zeigt diese Stoßstange aus Pappe. Der aufgeklebte QR-Code enthält die Informationen zu der Identifikationsnummer (ID) des Fahrzeugs. Ebenfalls wurde in den Modellfahrzeugen ein Stoßsensormodul verbaut. Dadurch können unvorhersehbare Kollisionen und Drehungen erkannt werden. Somit ergibt sich folgende, verbaute Hardware in jedem Modellfahrzeug:

- Mehrere Acrylplatten, Leitungen, Schrauben, Muttern und Abstandshalter für die Konstruktion
- Hauptsteuerungsplatine für den Raspberry Pi (Robot HAT)
- 4 Räder
- 2 Gleichspannungsmotoren für den hinteren Antrieb
- Motoransteuerungsplatine (TRA9118A) für die zwei Gleichspannungsmotoren
- Servomotor (Schrittmotor) für die Lenkung
- Modul für Pulsweitenmodulation (PCA9685) für die Ansteuerung des Servomotors

- Raspberry Pi 3 B mit ARM Cortex-A53 1,2 GHz CPU, 1 GB RAM, USB 2.0, 8 GB SD mit Raspbian.
- Raspberry Pi V2 Kameramodul inklusive Gehäuse und Halterung
- Ultraschallsensormodul (HC-SR04) mit einem horizontalen Erfassungsbereich von ca. 13° für die Abstandsmessung (Radarsensor), Ansteuerung von der Hauptsteuerungsplatine.
- 2 Akkus (18650) inklusive Akkuhalterung
- Stoßsensormodul (KY-031)

5.3 Aufbau der Teststrecke

Für die Durchführung ähnlicher Experimente aus der Simulation, wurde in der Realität eine Teststrecke aufgebaut. Dadurch konnten die zuvor entwickelten Algorithmen in Kapitel 3 überprüft werden. Dabei kann ebenfalls das Verhalten der Fahrzeuge auf der Fahrbahn aus der Realität mit dem Verhalten der Fahrzeuge aus der Simulation verglichen werden. An dieser Stelle steht die Erstellung eines Notfallkorridors für die Polizei- und Rettungsfahrzeuge in der echten Umgebung im Vordergrund. Zusätzlich konnte diese Teststrecke dafür verwendet werden, um die zuvor in Kapitel 4 entwickelte Spurerkennung auf Modellfahrzeugen zu überprüfen und für diesen Zweck anzupassen. Zum Ende der Strecke ist diese ebenfalls gekrümmt, um die Spurerkennung in den Kurven zu überprüfen (Abb. 5.3b). Die nachfolgende Abbildung 5.3 zeigt die erstellte Teststrecke im Einsatz.



(a) Ohne Modellfahrzeuge, Spurverlauf gerade.

(b) Ohne Modellfahrzeuge, Spurverlauf gekrümmt.

(c) Mit Modellfahrzeugen, Spurverlauf gerade.

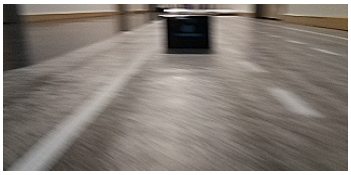
Abbildung 5.3: Aufbau der Teststrecke (dreispurige Autobahn).

Die Teststrecke wurde zusätzlich an die Größe der verwendeten Modellfahrzeuge angepasst. Die erstellte Teststrecke ähnelt einer dreispurigen Autobahn (Abb. 5.3a). Um die geplanten Experimente auf einer zweispurigen Autobahn durchzuführen, kann zum Beispiel eine der Spuren „gesperrt“ werden. Für die Teststrecke wurde eine acht Meter lange Trittschalldämmung für Laminat verwendet. Diese ähnelt von der Farbe der echten

Fahrbahn. Um die Fahrbahntrennungen darzustellen, wurde weißes Isolierband verwendet. Die vorherige Abbildung 5.3c zeigt die Teststrecke inklusive der neun umgebauten Modellfahrzeuge.

5.4 Lenkverhalten, Spur-, Entfernungs- und Rotationserkennung

Durch die verschiedenen Versuche ist bei der Entwicklung des Lenkverhaltens der Modellfahrzeuge aufgefallen, dass die tatsächliche Pixeldifferenz bei der Spurerkennung für die Lenkung der Modellfahrzeuge verwendet werden kann. Die bei der Spurerkennung ermittelte Pixeldifferenz, von der Fahrspurmitte bis zur Mitte des Modellfahrzeugs, kann exakt als Lenkschritt in Grad übertragen werden. Dabei muss lediglich der maximale Lenkschwellwert gesetzt werden. Dieser maximale Lenkschwellwert beträgt 10°. Somit konnte an dieser Stelle das Lenkverhalten ohne eine Funktion für die Lenkung bestimmt werden. Bei dem Modellfahrzeug sind die Entfernungsmessungen zum vorderen Modellfahrzeug durch einen Ultraschallsensor realisiert. Durch den verbauten Stoßsensor konnten ebenfalls die unvorhersehbaren Kollisionen und Rotationen des Modellfahrzeugs erkannt werden. Zusätzlich, wie die Abbildung 5.4 zeigt, ist die Einstellung der Belichtungszeit der Kamera sehr wichtig.



(a) Nach links lenken. Belichtungszeit = automatisch.

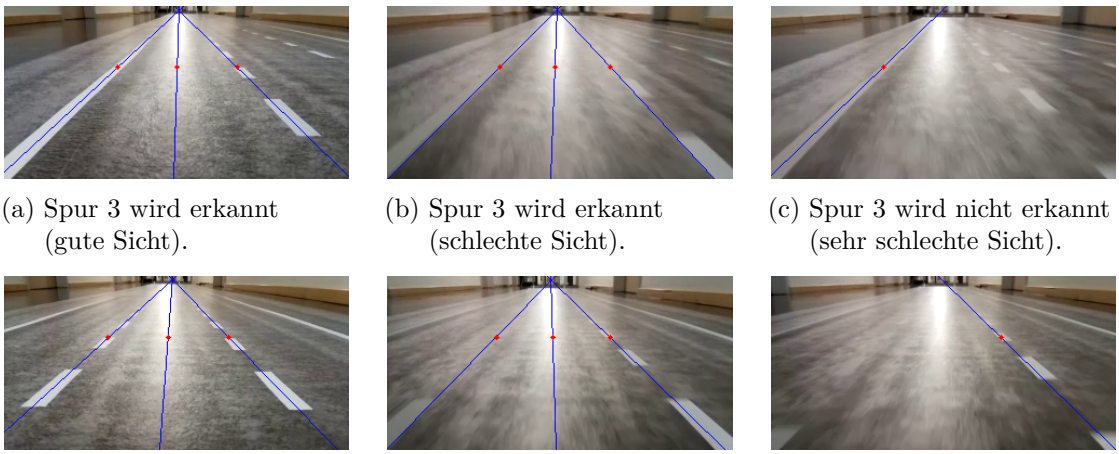
(b) Nach links lenken. Belichtungszeit = 5 ms.

(c) Nach rechts lenken. Belichtungszeit = 10 ms.

Abbildung 5.4: Darstellung der verschiedenen Lenkversuche mit verschiedener Einstellung der Belichtungszeit des Raspberry Pi V2 Kameramoduls.

Wenn die Belichtungszeit der Kamera zu hoch ist, wird ein verschwommenes Bild mit der Kamera aufgenommen (Abb. 5.4a). Dies war bei der automatischen Belichtungszeit der Fall. Außerdem kann man in Abbildung 5.4b erkennen, dass bei einer kleinen Belichtungszeit das Bild zwar schärfer, doch auch dunkler wird. Aus diesem Grund sollte ein fester Wert für die Belichtungszeit definiert werden. Dieser Beträgt in dieser Entwicklung bei dem Raspberry Pi V2 Kameramodul 10 Millisekunden (Abb. 5.4c). Wie bereits angesprochen, kann durch die verbaute Kamera die Fahrspur des Modellfahrzeugs erkannt werden. Dazu wurde die in Kapitel 4 vorgestellte Spurerkennung mit dem gefilterten Canny-Algorithmus verwendet, da diese sehr schnell und präzise auf den Datensätzen aus dem Simulator funktioniert. Wie die nachfolgende Abbildung 5.5 zeigt, ist diese durch die Anpassung der Parameter auch in der Praxis einsetzbar. Sobald die Fahrspur auf dem Bild klar dargestellt ist, wird diese Fahrspur exakt erkannt. Die Abbildungen 5.5a und 5.5d bestätigen dies. Wenn das Licht von der Fahrbahn reflektiert wird, wodurch das

Herausstechen der Fahrbahnmarkierungen beeinträchtigt wird, wird der Spurverlauf nicht mehr ganz genau erkannt (Abb. 5.5b und 5.5e). Sobald die Reflektion auf der Teststrecke zu hoch ist, kann die Fahrspur mit dem Modellfahrzeug nicht mehr erkannt werden. Die Abbildungen 5.5c und 5.5f zeigen dieses Extrembeispiel der Reflektion. Teilweise ist es auf diesen Bildern für das menschliche Auge ebenfalls schwer, die Spurmarkierung zu sehen. Im Normalfall tritt diese Reflektion nicht so häufig auf, wodurch das Modellfahrzeug optimal in der Fahrbahn gehalten werden kann. Diese extreme Reflektion ist auf die Folienbeschichtung der Teststrecke zurückzuführen. Außerdem beinhaltet die Experimentenumgebung mehrere Deckenlampen, die eine variable Anpassung der Schwellwerte bei der Spurerkennung erfordern. Aus diesem Grund wurden die beiden Schwellwerte für den gefilterten Canny-Algorithmus automatisch anhand der Lichtverhältnisse im Bild angepasst [90].



(a) Spur 3 wird erkannt (gute Sicht).
(b) Spur 3 wird erkannt (schlechte Sicht).
(c) Spur 3 wird nicht erkannt (sehr schlechte Sicht).
(d) Spur 2 wird erkannt (gute Sicht).
(e) Spur 2 wird erkannt (schlechte Sicht).
(f) Spur 2 wird nicht erkannt (sehr schlechte Sicht).

Abbildung 5.5: Gefilterter Canny-Algorithmus in dem Modellfahrzeug auf einer Teststrecke nach Übertragung des Algorithmus und Anpassung der Parameter aus der Simulation.

Zusätzlich konnten durch die Implementierung und Ausführung des gefilterten Canny-Algorithmus mit Multithreading auf der in diesem Kapitel verwendeten Hardware mit beschränkten Ressourcen bis zu ca. 30 FPS erreicht werden (Nr. 1 in Tab. 5.1). Die Tabelle 5.1 zeigt diese Laufzeitmessung der Spurerkennung inklusive der Laufzeit für die Entfernungsmessung mit einem Ultraschallsensor auf einen Blick. Dabei werden bei der Entfernungsmessung bis zu ca. 28 FPS erreicht (Nr. 7 in Tab. 5.1). Bei alleiniger Ausführung des Prozesses für die Entfernungsmessung können bei einem Abstand von 15 cm ca. 37 FPS, bei 30 cm ca. 34 FPS, bei 60 cm ca. 31 FPS und bei 90 cm ca. 28 FPS gemessen werden. Diese Messgeschwindigkeit des Ultraschallsensors und die Laufzeit für die Erkennung der Spur ist für den Zweck dieser Forschungsarbeit völlig ausreichend. Die Nummerierung der Spuren erfolgt, wie in der echten Umgebung, in Fahrtrichtung von

rechts nach links. Somit befindet sich ganz links die dritte Fahrspur. Die zurückgelegte Entfernung beträgt bei diesem Vorexperiment fünf Meter.

Tabelle 5.1: Laufzeit der Entfernungsmessung und der Spurerkennung auf dem Raspberry Pi 3 B. Die zweite Spalte enthält die Identifikationsnummer (ID) des Fahrzeugs. Die Laufzeit ist als Bilder pro Sekunde (FPS) angegeben.

Nr.	**Fahrzeug ID**	**Spur Nr.**	**Ultraschallsensor [FPS]**	**Spurerkennung [FPS]**
1	2	3	26,21	29,69
2	3	2	26,52	29,33
3	4	1	25,60	29,24
4	5	3	25,83	29,46
5	6	2	25,85	28,88
6	7	1	24,94	28,00
7	8	3	27,76	29,44
8	9	2	25,22	29,14
9	10	1	22,24	24,50

Die Vorversuche in Tabelle 5.1 zeigen, dass die verbauten Ultraschallsensoren unterschiedliche Laufzeiten bei der Entfernungsmessung erreichen. Das Auslesen der Sensoren ist unterschiedlich, da die verwendete Hardware, durch die kostengünstige Herstellung, leichte Unterschiede aufweist. Der Laufzeitenunterschied beträgt hierbei ca. 6 FPS (Vergleich zwischen Nr. 7 und 9 in Tab. 5.1). Durch die leistungsschwache Hardware wirkt sich diese Verlangsamung auf die Erkennung der Fahrspur aus (Nr. 9 in Tab. 5.1). Wie bereits erwähnt, wurden die Parameter für die Spurerkennung in dem Modellbaubereich für die Ausführung in der echten Umgebung angepasst. Bei der Anpassung der Parameter wurden lediglich die beiden Schwellwerte für den Canny-Algorithmus verändert, da die Lichtverhältnisse sich in der Praxis gegenüber der Simulation unterscheiden. Die Lichtverhältnisse sind ebenfalls nicht immer gleich, welche eine automatische Anpassung der Schwellwerte anhand der Lichtverhältnisse im Bild für den gefilterten Canny-Algorithmus erfordern. Die gefundenen und angepassten Parameter für den gefilterten Canny-Algorithmus und die probabilistische Hough-Transformation sehen wie folgt aus:

- Canny-Algorithmus
  - Schwellwert 1 (min) [90]: $max(0, (1,0 - 0,33) \cdot median(Graustufenbild))$
  - Schwellwert 2 (max) [90]: $min(255, (1,0 + 0,33) \cdot median(Graustufenbild))$
- Probabilistische Hough-Transformation (Ausschnitte = 1)
  - Schwellwert: 15
  - Minimale Länge der Linien: 20
  - Maximaler zulässiger Abstand zwischen den Linien: 40

- Filterung der Linien
  - Radius: 14
  - Winkel 1 (α): 25°
  - Winkel 2 (β): 80°
  - Linienschnittpunkt 1 (min): 0
  - Linienschnittpunkt 2 (max): 15
  - Winkelschwellwert: 10

5.5 Kommunikation zwischen Modellfahrzeugen

Die Kommunikation zwischen den Modellfahrzeugen erfolgt über WLAN. An dieser Stelle ist es der einfachste Weg eine Kommunikation zu realisieren, da die Modellfahrzeuge ebenfalls über WLAN angesteuert werden. Dieses vereinfacht die Ausführung der verschiedenen Skripte auf dem Raspberry Pi und erleichtert durch das Secure Shell (SSH) Netzwerkprotokoll zusätzlich die Fehlersuche. Anschließend kann so auf die Ausgaben der ausgeführten Skripte zugegriffen werden. Der Zugriff auf das Modellfahrzeug über WLAN ermöglicht eine leichte Kalibrierung der Schrittmotoren und Konfiguration der Modellfahrzeuge über SSH. Diese Verbindung ist durch den Sicherheitsstandard WPA2 (engl. Wi-Fi Protected Access 2) gesichert. Bei der Übertragung werden zwei verschiedene Protokolle für den Austausch der Nachrichten mit UDP (engl. User Datagram Protocol) und TCP (engl. Transmission Control Protocol) implementiert und getestet. Beide Vorgehensweisen eignen sich für die Übertragung der Nachrichten in dem privaten Testnetzwerk der Modellfahrzeuge. Als eine Erweiterung kann die Kommunikation zwischen den Modellfahrzeugen ebenfalls auf eine 433 MHz Funkübertragung oder auf Bluetooth Low Energy (BLE) umgestellt werden. Weitere Informationen dazu gibt der Abschnitt 8.2.

Durch die Kommunikation zwischen den Modellfahrzeugen kann auf verschiedene Nachrichten eigenständig reagiert werden. Sobald zum Beispiel ein Unfall beziehungsweise eine Kollision des Modellfahrzeugs erkannt wird, führt das Fahrzeug eine Gefahrenbremsung durch und benachrichtigt die Fahrzeuge in der Umgebung. Dabei enthält die Nachricht einen Namen und Informationen über die eigene Identifikationsnummer des Unfallfahrzeugs auf der Fahrbahn. Anders als in der Simulation in Abschnitt 3.3.3 wird in der echten Umgebung über die Luft kommuniziert. In den versendeten Verbindungspaketen sind bereits der Empfänger beziehungsweise der Absender angegeben. Somit ändert sich das Format und die Datenstruktur für den Nachrichtenaustausch zu einer zusammengesetzten Zeichenkette. Das Format ändert sich wie folgt: `Nachrichtenname:SpurNummer:VorderID` oder `Nachrichtenname:Nachricht`. Zum Beispiel `LOCALMAP:1:4`, um eine lokale Karte aufzubauen oder `FORWARD:COLLISION`, um die Nachricht für einen Unfall weiterzuleiten.

Bei den Nachrichten kann ebenfalls zwischen Unfall beziehungsweise Kollision `COLLISION`, Weiterleitung `FORWARD`, Rettung `RESCUE`, Fahrspur `LANE`, Gesamtspuren `TOTALLANES`, lokale Karte `LOCALMAP`, Position `POSITION`, Anhalten `STOP` und Zustandsänderung `CHANGESTATE`

unterschieden werden. Die drei Nachrichtentypen Gesamtspuren `TOTALLANES`, lokale Karte `LOCALMAP` und Position `POSITION` werden hierbei sowohl für die Rettungsgassenbildung beim stehenden Verkehr als auch für die Ermittlung der Position des Fahrzeugs bei der Fahrt verwendet. Anders als in der Simulation, gibt es bei den Modellfahrzeugen keinen separaten Nachrichtennamen für die Drehung des Fahrzeugs, da die dafür notwendigen Sensoren in dem Modellfahrzeug nicht verbaut sind. Somit werden bei den Modellfahrzeugen die Kollision und die Drehung des Fahrzeugs zu einem Nachrichtentyp zusammengefasst. Die Tabelle 5.2 zeigt diese Nachrichten im Überblick.

Tabelle 5.2: Überblick der Nachrichtennamen inklusive des Typs und der Beschreibung bei der Kommunikation zwischen autonomen Modellfahrzeugen.

Name	**Typ**	**Beschreibung**
`COLLISION`	Kollision/Drehung	Zusammenstoß/Drehung des Fahrzeugs
`FORWARD`	Weiterleitung	Weiterleitung der ankommenden Nachricht
`RESCUE`	Rettung	Benachrichtigung der Rettungsfahrzeuge über einen Unfall
`LANE`	Fahrspur	Eigene Fahrspur des Fahrzeugs
`TOTALLANES`	Gesamtspuren	Gesamtspuren der Autobahn
`LOCALMAP`	Lokale Karte	Aufbau der lokalen Karte
`POSITION`	Position	Eigene Spur des Fahrzeugs und ID des vorderen Fahrzeugs
`STOP`	Anhalten	Eigenes Fahrzeug anhalten
`CHANGESTATE`	Zustandsänderung	Änderung des Zustands (Rettungsgassenbildung stehender Verkehr)

Die Fahrzeuge, die eine Nachricht `COLLISION` von einem Unfallfahrzeug bekommen, reagieren eigenständig auf diese Nachricht. Dabei kann durch den Aufbau der lokalen Karte die Fahrspur und die Position auf dieser Fahrspur dieser Fahrzeuge verglichen werden. Die Fahrzeuge, die unmittelbar hinter dem Unfallfahrzeug sind, führen eine Gefahrenbremsung durch und leiten diese Nachricht mit dem Namen `FORWARD` an die anderen Modellfahrzeuge in der Umgebung. Die anderen Modellfahrzeuge, für die keine Gefahr besteht, nehmen die Nachricht an und verständigen die Einsatzkräfte mit der Nachricht `RESCUE`. Wenn das autonome Modellfahrzeug die Gesamtspuren der Autobahn durch die Kommunikation bestimmen muss, kann dieses eine Nachricht `TOTALLANES` versenden. Die Modellfahrzeuge in der Umgebung antworten mit der Angabe ihrer eigenen Fahrspur in der Nachricht `LANE`. Wie bereits erwähnt, kann für die Bildung der Rettungsgasse bei einem stehenden Verkehr durch die Kommunikation zwischen den Modellfahrzeugen die notwendige lokale Karte der Modellfahrzeuge in der echten Umgebung aufgebaut werden. Um die Informationen jedes einzelnen Fahrzeugs anzufragen, wird die Nachricht `LOCALMAP` versendet. Die Fahrzeuge, die diese Nachricht bekommen, antworten mit dem Nachrichtennamen `POSITION`. Diese Nachricht enthält die Informationen über die eigene Fahrspur des Fahrzeugs und die Identifikationsnummer des vorderen Modellfahrzeugs auf derselben Fahrspur. Dadurch kann anschließend jedes einzelne Modellfahrzeug eine

eigene lokale Karte mit den Fahrzeugen in der echten Umgebung erstellen. Ein Beispiel der lokalen Karte zeigt die nachfolgende Tabelle 5.3.

Tabelle 5.3: Beispiel für eine lokale Karte mit neun autonomen Modellfahrzeugen und drei Unfallfahrzeugen auf einer dreispurigen Autobahn. Die Werte in den einzelnen Reihen sind als Identifikationsnummer (ID) der autonomen Modellfahrzeuge angegeben. Die Unfallfahrzeuge sind mit der ID 0 gekennzeichnet.

Spur	**Reihe 0**	**Reihe 1**	**Reihe 2**	**Reihe 3**
1	0	7	4	1
2	0	8	5	2
3	0	9	6	3

Für das Ändern des Zustands für die Ausführung der Bildung der Rettungsgasse bei einem stehenden Verkehr, wird die Nachricht `CHANGESTATE` versendet. Die Modellfahrzeuge entscheiden eigenständig, ob durch Zurücksetzen, der Abstand nach vorne für die Bildung der Rettungsgasse vergrößert werden muss. Da die autonomen Modellfahrzeuge keine Rücksensoren für die Entfernungsmessung beim Zurücksetzen haben, werden diese Fahrzeuge durch die unmittelbar hinteren Fahrzeuge benachrichtigt. Durch die Nachricht mit dem Namen `STOP` werden die ausführenden Modellfahrzeuge über zu nahes Auffahren benachrichtigt.

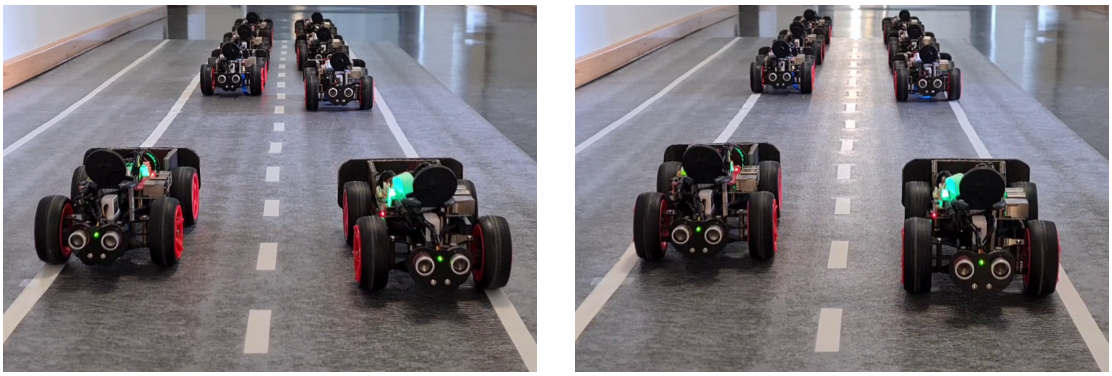
5.6 Experimente

Dieser Abschnitt präsentiert die durchgeführten Experimente aus der Praxis. Es werden analoge Experimente mit den autonomen Modellfahrzeugen durchgeführt, wie zuvor in dem Simulator. Dadurch kann das Verhalten der Algorithmen für die Bildung der Rettungsgasse in Simulation und Realität verglichen werden. In dem Abschnitt 3.4 konnte man sehen, dass die entwickelten Algorithmen für die Bildung der Rettungsgasse bei einem stockenden und stehenden Verkehr in dem entwickelten Simulator erfolgreich funktionieren. Hierbei wird ein ähnliches Verhalten der autonomen Modellfahrzeuge, wie zuvor mit den simulierten autonomen Fahrzeugen, erwartet.

5.6.1 Bildung der Rettungsgasse: Stockender Verkehr

Dieses Experiment zeigt den ersten Algorithmus aus dem Simulator in Abschnitt 3.4.1 für die Bildung der Rettungsgasse bei einem stockenden Verkehr mit den autonomen Modellfahrzeugen. Die Rettungsgasse wird zwischen der letzten und der vorletzten Fahrspur gebildet. Die Zählung beginnt in Fahrtrichtung von rechts nach links, wie auf einer echten Autobahn ebenfalls. Bei der Rettungsgassenbildung muss die Position des linken oder rechten Vorderrads des Modellfahrzeugs mit den zuvor erkannten Fahrbahnmarkierungen verglichen werden. An dieser Stelle wird auf die bereits vorgestellte Spurerkennung in Kapitel 4 zurückgegriffen. Somit weiß das Modellfahrzeug in welche Richtung (links oder rechts) es sich bewegen muss, um eine Rettungsgasse abhängig von eigener Fahrspur zu

bilden. Wie die nachfolgende Abbildung 5.6a zeigt, kann die Rettungsgasse mit dem autonomen Modellfahrzeug erfolgreich gebildet werden. Bei einem stockenden Verkehr fahren die Modellfahrzeuge automatisch nach links beziehungsweise rechts, um einen Korridor für die Rettungsfahrzeuge zu erstellen. Wenn die Geschwindigkeit der Modellfahrzeuge sich erhöht, wird die geöffnete Rettungsgasse automatisch geschlossen (Abb. 5.6b). So kann sichergestellt werden, dass die Fahrzeuge zu jedem Zeitpunkt nach dem Bremsen eine Rettungsgasse für die Polizei- und Rettungsfahrzeuge gebildet haben.



(a) Öffnen einer Rettungsgasse.

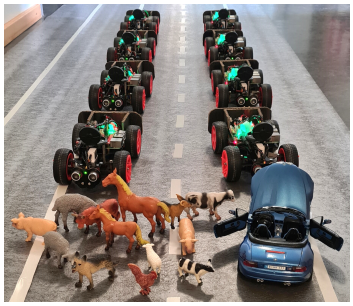
(b) Schließen einer Rettungsgasse.

Abbildung 5.6: Bildung einer Rettungsgasse mit autonomen Modellfahrzeugen bei einem stockenden Verkehr auf einer zweispurigen Autobahn.

5.6.2 Bildung der Rettungsgasse: Hindernisse auf der Fahrbahn

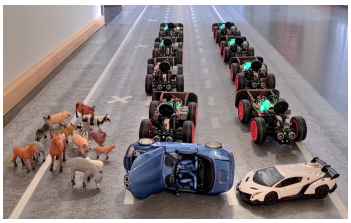
In diesem Abschnitt wird die Bildung der Rettungsgasse bei verschiedenen Hindernissen auf einer Autobahn überprüft. Wenn die Geschwindigkeit sich verkleinert, ist es theoretisch für die Modellfahrzeuge irrelevant, aus welchem Grund dies geschieht. Dabei ist die Position des Objekts gegenüber der Art des Objekts wichtiger. Sobald ein Objekt auf der eigenen Fahrbahn erkannt wird, welches sich dem Fahrzeug nähert, wird die Geschwindigkeit im Modellfahrzeug verringert. Anschließend wird die Rettungsgasse automatisch geöffnet, wenn die Geschwindigkeit unter einem bestimmten Wert fällt. Wie die nachfolgende Abbildung 5.7 zeigt, kann die Rettungsgasse erfolgreich auf einer zweispurigen Autobahn bei einem Unfall (Abb. 5.7a), auf einer zweispurigen Autobahn bei einem Wildwechsel (Abb. 5.7b und 5.7c), auf einer dreispurigen Autobahn bei einem Unfall mit gesperrter Spur (Abb. 5.7d) und auf einer dreispurigen Autobahn bei einem Unfall (Abb. 5.7e und 5.7f) gebildet werden. Die Bildung der Rettungsgasse mit den autonomen Modellfahrzeugen auf einer dreispurigen Autobahn musste nacheinander Fahrspur für Fahrspur durchgeführt werden, damit die verbauten Ultraschallsensoren sich während der Fahrt nicht gegenseitig stören (Abb. 5.7e und 5.7f). Dabei wird vermutet, dass durch die unterschiedlichen Schallwellen der Ultraschallsensoren teilweise falsche Messwerte aus der Umgebung wahrgenommen werden. Auf dieses Thema wird in dem Abschnitt 5.6.5 mehr eingegangen.

(a) Auf einer zweispurigen Autobahn bei einem Unfall (Frontansicht).



(b) Auf einer zweispurigen Autobahn bei einem Wildwechsel (Frontansicht).

(c) Auf einer zweispurigen Autobahn bei einem Wildwechsel (Rückansicht).



(d) Auf einer dreispurigen Autobahn mit gesperrter Fahrspur (Frontansicht).

(e) Auf einer dreispurigen Autobahn bei einem Unfall (Frontansicht).

(f) Auf einer dreispurigen Autobahn bei einem Unfall (Rückansicht).

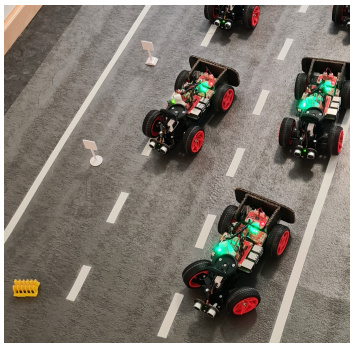
Abbildung 5.7: Bildung einer Rettungsgasse mit autonomen Modellfahrzeugen bei Hindernissen auf Autobahnen mit verschiedenen Anzahl der Fahrspuren.

5.6.3 Bildung der Rettungsgasse: Platzen des Vorderreifens

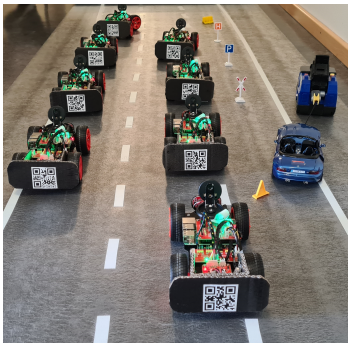
Um zu überprüfen, ob die Rettungsgasse mit den autonomen Modellfahrzeugen zu jedem zufälligen Zeitpunkt gebildet werden kann, wurde ein Platzen des Vorderreifens des Modellfahrzeugs vorgespielt. Bei diesem Experiment kam der Stoßsensor zum Einsatz, um eine unvorhersehbare Bewegung des Modellfahrzeugs zu erkennen. Die für das Modellfahrzeug unvorhersehbare Bewegung wurde händisch durchgeführt. Dabei wurde das Modellfahrzeug bei der Fahrt zufällig angestoßen. Dieses Unfallfahrzeug hat somit einen Unfall erkannt und eine Notbremsung durchgeführt. Wie die Abbildung 5.8a zeigt, befindet sich dieses Modellfahrzeug auf der dritten Fahrspur. Nachdem der Unfall beziehungsweise eine unvorhersehbare Drehung erkannt wurde, wurden die Fahrzeuge in der Umgebung benachrichtigt, um einen Aufprallunfall zu vermeiden. Diese Modellfahrzeuge konnten anschließend auf die ankommende Nachricht eigenständig reagieren. Wie die Abbildung 5.8b zeigt, hat das erste Modellfahrzeug auf der zweiten Fahrspur ebenfalls eine Gefahrenbremsung durchgeführt, da dieses Modellfahrzeug dem Unfallfahrzeug am nächsten ist. Dabei hat das erste Modellfahrzeug auf der zweiten Fahrspur bei der Notbremsung trotzdem die Bildung der Rettungsgasse durchgeführt. Alle anderen Fahrzeuge haben den Sicherheitsabstand eingehalten und waren somit etwas weiter entfernt. Da-

durch konnten diese Modellfahrzeuge eine normale Bremsung durchführen und dabei den Rettungskorridor für die Polizei- und Rettungsfahrzeuge öffnen (Abb. 5.8c).

(a) Gefahrenbremsung des Unfallfahrzeugs.



(b) Gefahrenbremsung der Modellfahrzeuge in der Nähe.

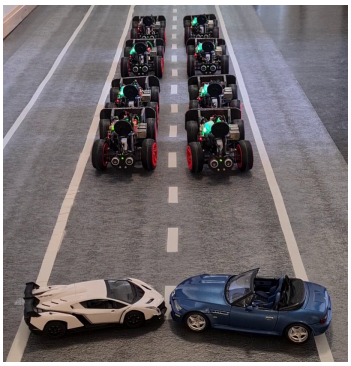


(c) Rettungsgassenbildung der hinteren Modellfahrzeuge.

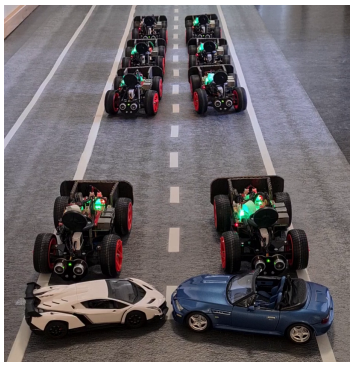
Abbildung 5.8: Geplante Durchführung eines geplatzten Vorderreifens während der Fahrt eines autonomen Modellfahrzeugs auf einer dreispurigen Autobahn mit gesperrter erster Fahrspur.

5.6.4 Bildung der Rettungsgasse: Stehender Verkehr

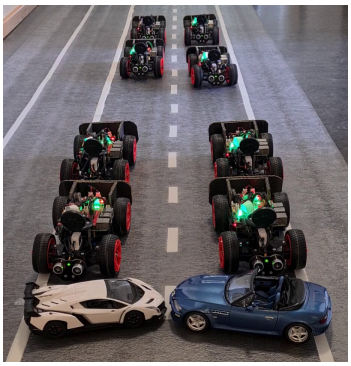
Dieser Abschnitt zeigt die Experimente des zweiten Algorithmus aus dem Simulator in Abschnitt 3.4.2 für die Bildung der Rettungsgasse bei einem stehenden Verkehr in der echten Umgebung. Die nachfolgende Abbildung 5.9 zeigt die systematische Ausführung dieses Algorithmus mit den autonomen Modellfahrzeugen.



(a) Erste Reihe.



(b) Zweite Reihe.



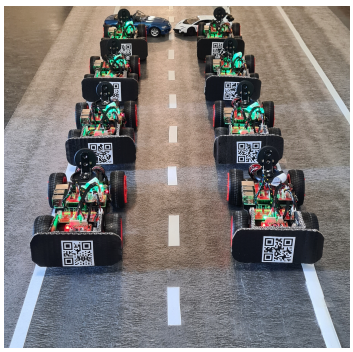
(c) Dritte Reihe.

Abbildung 5.9: Bildung einer Rettungsgasse mit autonomen Modellfahrzeugen auf einer zweispurigen Autobahn beim stehenden Verkehr bei einem Unfall.

An dieser Stelle ist es ebenfalls für jedes einzelne Modellfahrzeug erforderlich, eine lokale Karte durch die Kommunikation und bildliche Auswertung des vorderen Modellfahrzeugs

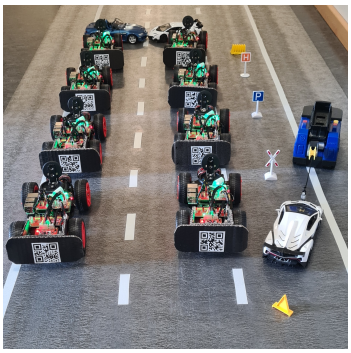
zu erzeugen. Durch das Austauschen eigener Identifikationsnummer, eigener Spurposition und der Identifikationsnummer des vorderen Fahrzeugs, kann anschließend die Reihenfolge der Modellfahrzeuge auf den verschiedenen Fahrspuren in der lokalen Karte erstellt werden. Somit entscheiden die Modellfahrzeuge selbst, wer mit der Ausführung der Rettungsgassenbildung beginnt. Dabei bilden die Modellfahrzeuge die Rettungsgasse für die Polizei- und Rettungsfahrzeuge systematisch Reihe für Reihe. Auch hier erfolgt die Ausführung der Rettungsgassenbildung bei einem stehenden Verkehr über verschiedene Zustände. Die Modellfahrzeuge, die den Unfallfahrzeugen am nächsten sind, beginnen mit der Bildung der Rettungsgasse (Abb. 5.9a). Wenn die Entfernung für die Rettungsgassenbildung zu gering ist, fahren die Modellfahrzeuge automatisch nach hinten. Da die autonomen Modellfahrzeuge keine Rücksensoren für die Entfernungsmessung haben, benachrichtigen die hinteren Fahrzeuge die vorderen Fahrzeuge über zu nahes Auffahren.

(a) Zweispurige Autobahn ohne gesperrte Spur (Frontansicht).



(b) Zweispurige Autobahn ohne gesperrte Spur (Rückansicht).

(c) Dreispurige Autobahn mit gesperrter Spur (Frontansicht).



(d) Dreispurige Autobahn mit gesperrter Spur (Rückansicht).

(e) Dreispurige Autobahn ohne gesperrte Spur (Frontansicht).

(f) Dreispurige Autobahn ohne gesperrte Spur (Rückansicht).

Abbildung 5.10: Bildung einer Rettungsgasse mit autonomen Modellfahrzeugen auf Autobahnen mit verschiedenen Anzahl der Fahrspuren beim stehenden Verkehr.

Sobald das ausführende Modellfahrzeug die Bildung der Rettungsgasse abgeschlossen hat, benachrichtigt dieses das Modellfahrzeug hinter sich auf der gleichen Fahrspur (Abb. 5.9b). Ab der dritten Reihe ist die Entfernung für eine Rettungsgasse ausreichend, sodass diese Modellfahrzeuge nicht mehr nach hinten fahren müssen (Abb. 5.9c). Wie die vorherige Abbildung 5.10 zeigt, kann die Rettungsgasse beim stehenden Verkehr erfolgreich auf einer zweispurigen Autobahn (Abb. 5.10a und 5.10b), auf einer dreispurigen Autobahn mit gesperrter Spur (Abb. 5.10c und 5.10d) und auf einer dreispurigen Autobahn (Abb. 5.10e und 5.10f) automatisch durchgeführt werden.

5.6.5 Auswertung der Ergebnisse

Dieser Abschnitt präsentiert die Auswertung der durchgeführten Experimente mit den autonomen Modellfahrzeugen in der echten Umgebung. Dabei kann als erstes an dieser Stelle erwähnt werden, dass das Hauptziel dieser Forschungsarbeit in diesem Kapitel erfüllt wurde. Die entwickelten Algorithmen für die Rettungsgassenbildung konnten erfolgreich aus der Simulation auf die Realität übertragen werden. Die volle Funktionalität der Algorithmen für die Rettungsgassenbildung beim stockenden und stehenden Verkehr konnte, sowohl in der Simulation als auch in der Realität, experimentell bewiesen werden. Die Rettungsgasse konnte mit den autonomen Modellfahrzeugen zu jedem Zeitpunkt erfolgreich gebildet werden. Dabei ist der Grund für die Bildung der Rettungsgasse nicht relevant. Dieser kann zum Beispiel stockender Verkehr, Wildwechsel, Platzen des Vorderreifens, ein Unfall oder ein anderes unvorhersehbares Ereignis sein. Die Durchführung der Experimente mit den Modellfahrzeugen gestaltet sich schwieriger als zuvor in der Simulation. In der echten Umgebung sind ebenfalls die Qualität und die Messtoleranz der verwendeten Sensoren wichtig. Dabei spielt die Konstruktion der getesteten Modellfahrzeuge zusätzlich eine wichtige Rolle.

Wie bereits angesprochen, wurde die Bildung der Rettungsgasse mit den Modellfahrzeugen bei einem Unfall auf einer dreispurigen Autobahn nacheinander Fahrspur für Fahrspur durchgeführt. So konnten die autonomen Modellfahrzeuge sich nicht gegenseitig während der Fahrt stören. Durch das Schwanken der Fahrzeuge auf der Fahrbahn wurden teilweise die vorderen Fahrzeuge in der Nachbarspur links und rechts erkannt. Das Modellfahrzeug konnte an dieser Stelle die Position des erkannten Objekts nicht eindeutig zuordnen. Dadurch wird die Geschwindigkeit verkleinert und die Rettungsgasse gebildet, da diese seitliche Erkennung gleichgestellt ist mit der Erkennung in der Front. Um dieses Problem zu beseitigen, könnte eventuell eine Teststrecke mit breiteren Fahrspuren helfen. Zusätzlich könnten hierbei mehrere Ultraschallsensoren mit einem geringen horizontalen Erfassungsbereich im autonomen Modellfahrzeug von Nöten sein. So könnte die Entfernungsmessung exakt einem Sensor und einem Objekt zugeordnet werden. Zusätzlich entsteht hierbei die Vermutung, dass die Ultraschallsensoren sich gegenseitig stören könnten, da diese Sensoren die gleichen Impulse für die Entfernungsmessung versenden und unterschiedliche Schallwellen von anderen Ultraschallsensoren empfangen.

Durch die bildliche Auswertung der Fahrspur dürfen die Fahrspurmarkierungen nicht komplett durch andere Objekte im Bild verdeckt werden. Bei der Rettungsgassenbildung beim stockenden Verkehr ist es auch der Fall, da die Modellfahrzeuge genügend

Sicherheitsabstand einhalten, um rechtzeitig bremsen zu können. Bei der Rettungsgassenbildung beim stehenden Verkehr hingegen, stehen die Modellfahrzeuge bereits und haben einen kürzeren Abstand zum vorderen Fahrzeug, wodurch das vordere Fahrzeug im Bild größer wirkt und teilweise die Fahrbahnmarkierung verdeckt. Wenn die Fahrspur im Bild nicht mehr erkannt werden kann, weil das Fahrzeug zu nah ist, kann die Rettungsgasse beim stehenden Verkehr an dieser Stelle nicht gebildet werden. Aus diesem Grund wurde der Ausgangsabstand zwischen den autonomen Modellfahrzeugen bei der Ausführung dieses Algorithmus auf ca. 26 cm gesetzt. Diese Angabe entspricht genau einer Modellfahrzeuglänge und ähnelt dem Vorgehen wie zuvor in der Simulation aus Kapitel 3.

5.7 Schlussfolgerungen

Die Experimente zeigen, dass die autonomen Modellfahrzeuge sich ähnlich, wie die autonomen Fahrzeuge in der Simulation, verhalten. Die Rettungsgasse konnte erfolgreich beim stockenden Verkehr, bei Hindernissen auf der Fahrbahn, beim Platzen des Vorderreifens und beim stehenden Verkehr gebildet werden. Trotz der Schwächen der Modellfahrzeuge bei der Konstruktion und verbauter Sensorik, konnten die verschiedenen Experimente erfolgreich durchgeführt werden. Damit kann man sagen, dass die verwendeten Modellfahrzeuge für den Zweck dieser Forschungsarbeit völlig ausreichend sind.

Wie man ebenfalls in diesem Kapitel sehen konnte, konnte die in dem vorherigen Kapitel 4 entwickelte Spurerkennung durch den gefilterten Canny-Algorithmus ebenfalls erfolgreich auf die Modellfahrzeuge übertragen werden. Wie bereits erwähnt, ist dieses Vorgehen durch die Anpassung einiger Parameter auch für den Echtwelteinsatz geeignet. Dies bestätigt ebenfalls dieses Kapitel. Die vorgestellte Spurerkennung konnte zusätzlich auf der verwendeten Hardware, dem Raspberry Pi 3 B, durch die Multithreading-Funktionalität auf eine Echtzeitauswertung von bis zu ca. 30 FPS gebracht werden.

Zusätzlich ist es für das autonome Modellfahrzeug interessant zu wissen, welche Objekte sich ebenfalls auf der gleichen Fahrbahn befinden. Dafür ist eine Objekterkennung auf Hardware mit beschränkten Ressourcen von Nöten. Diese wird im nachfolgenden Kapitel 6 demonstriert. Zusätzlich, nach der Erkennung der Objekte auf der Fahrbahn, kann die Implementierung einer Abstandsmessung in einem zweidimensionalen Bild vorgenommen werden. Dadurch kann, zum Beispiel, ein Radarkontrollsystem entwickelt werden. Dieses Vorgehen wird in dem Kapitel 7 vorgestellt. Das Radarkontrollsystem ist realisiert durch die Messungen der Pixel im Bild und anschließender Umwandlung dieser Pixel auf die tatsächliche gemessene Distanz. Ebenfalls wird in den nächsten Kapiteln 6 und 7 das Thema der Entwicklung der Software für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte eingeführt. Dadurch können bewusst die Kosten für den Stromverbrauch und die Hardware gesenkt werden.

In der Informatik geht es genau so wenig um Computer, wie in der Astronomie um Teleskope.

Edsger Wybe Dijkstra
1930 – 2002

Kapitel 6

Objekterkennung auf Hardware mit beschränkten Ressourcen

6.1 Einführung

Heutzutage ist es möglich, Fahrzeuge autonom ohne Eingriffe des Menschen fahren zulassen. Diese Fahrzeuge besitzen eine Vielzahl an Sensoren und Kameras, um ihre Umwelt zu interpretieren und mit ihr entsprechend interagieren zu können. Aus diesem Grund ist das Thema der Computer Vision in den letzten Jahren sehr beliebt geworden. Zum Beispiel kann mithilfe von Radarsensoren die Entfernung zum vorderen Fahrzeug ermittelt werden. Doch wie verhält sich das Fahrzeug, wenn dieses System ausfällt oder falsche Entfernungsmessungen liefert? Hierbei könnte ein Kontrollsystem vom Vorteil sein, welches die verschiedenen Messwerte überprüft und miteinander abgleicht. Dieses könnte mit einer optischen Entfernungsmessung in einem zweidimensionalen Bild realisiert werden. Doch dafür muss als erstes die Position und Größe dieser Objekte bekannt sein. Somit müssen die verschiedenen Objekte erkannt und klassifiziert werden.

Dieses Kapitel präsentiert mehrere Modelle zur Erkennung von individuellen Objekten mit TensorFlow 1 und TensorFlow 2 in einem zweidimensionalen Bild mit faltenden neuronalen Netzwerken (ConvNets). Dabei steht die Ausführung der Objekterkennung auf Hardware mit beschränkten Ressourcen im Vordergrund. Zu Beginn wird auf die verwandten Arbeiten dieser Thematik eingegangen. Zusätzlich werden die in diesem Kapitel vorgestellten ConvNet-Modelle sowohl mit Bilddaten aus der Simulation als auch mit echten Daten aus dem Modellbaubeireich trainiert und ausgewertet. Für das überwachte Lernen der ConvNet-Modelle sind bekanntlich viele annotierte Trainingsdaten notwendig. Diese annotierten Trainingsdaten werden mit dem in Kapitel 3 vorgestellten Simulator automatisch erzeugt. Das Vorgehen für das Erzeugen annotierter Trainingsdaten mit dem Simulator wird ebenfalls in diesem Kapitel vorgestellt. Außerdem werden verschiedene Modelle für die Objekterkennung trainiert und als erstes auf einer leistungsstarken Hardware

mit einer Grafikkarte ausgewertet, um erste Laufzeitunterschiede der Modelle zu erhalten. Die Idee hierbei ist es ein geeignetes System für die in dem Kapitel 5 vorgestellten autonomen Modellfahrzeuge mit einer nicht leistungsstarken Hardware ohne Grafikkarte zu finden. Die verschiedenen Modelle der Objekterkennung werden zusätzlich verglichen, um das bessere und schnellere System für die Objekterkennung auf einer Hardware mit beschränkten Ressourcen zu finden. Ebenfalls wird in diesem Kapitel der Ansatz eines Transferlernens (engl. Transfer Learning) bei der Objekterkennung ausprobiert. Es werden einige Modelle für die Simulation-zu-Realität Übertragung (engl. Sim-to-Real Transfer) trainiert und getestet. Durch die in diesem Kapitel beschriebenen Experimente wird der Vergleich der Laufzeit präsentiert. Ebenfalls wird in diesem Kapitel der Einsatz einer Hardwareerweiterung demonstriert. Dazu wird der Intel Neural Compute Stick 2 (NCS2) verwendet, um auf einer Hardware mit beschränkten Ressourcen eine Echtzeitobjekterkennung zu entwickeln. Dabei wird hauptsächlich die Netzwerkarchitektur der Single Shot Detector (SSD) Modelle verwendet.

Die individuelle Objekterkennung auf Hardware mit beschränkten Ressourcen ist nicht nur für den Modellbaubereich interessant. Zusätzlich kann diese Erkennung der Objekte in Echtzeit auf Hardware ohne Internetanbindung vom Vorteil sein. Zum Beispiel bei Post-Drohnen, die das Paket in dem Garten abstellen sollen oder in Überwachungskameras aus dem Land- und Forstbetrieb, um verschiedene Tiere im Wald zu erkennen und zu klassifizieren. Zusätzlich kann durch eine Echtzeitauswertung zum Beispiel die Route eines Objekts verfolgt werden. Die Idee hinter dieser Objekterkennung ist, die Voraussetzung für das in Kapitel 7 vorgestellte Kontrollsystem für den Radarsensor, zu erfüllen. Ebenfalls ist es interessant zu sehen: Wenn keine traditionellen Methoden verwendet werden können, wie können dann die neuronalen Netzwerke trotzdem auf der Hardware mit beschränkten Ressourcen verwendet werden, um eine Auswertungen in Echtzeit zu erreichen? Zusätzlich wird in diesem Kapitel das Thema der Entwicklung der Software für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte eingeführt und steht bei der Entwicklung im Vordergrund. Dadurch können bewusst die Kosten für den Stromverbrauch und die Hardware eingespart werden.

6.2 Verwandte Arbeiten

Zahlreiche wissenschaftliche Arbeiten befassen sich mit der Objekterkennung. Zum Beispiel gibt es eine Deep-Learning-basierte Multiskalen-Multi-Objekterkennung und Klassifizierung von Objekten für das autonome Fahren [29] oder eine Verbesserung der Hinderniserkennung in autonomen Fahrzeugen unter Verwendung des YOLO Non-Maximum Suppression Fuzzy-Algorithmus [123]. In einigen wissenschaftlichen Arbeiten wird ein Rahmen für die Formerkennung zur Erkennung von Objekten in unübersichtlichen Bildern eingeführt [124]. Ein weiterer Ansatz zur Objekterkennung ist die Unterscheidung der Objekte anhand von 3D-Informationen. Zu diesem Zweck können Lidarsensoren [7] oder die bereits zuvor beschriebene Stereokamera verwendet werden [60]. Allerdings gibt es nur wenige wissenschaftliche Arbeiten, die sich mit der Erkennung der Objekte in Echtzeit auf Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte

beschäftigen. An dieser Stelle gibt es ein Echtzeitobjekterkennungssystem auf mobilen Geräten. Dieses System erreicht 23, 6 Bilder pro Sekunde (engl. Frames Per Second - FPS) auf einem iPhone 8 [116]. In ähnlicher Weise gibt es ebenfalls eine Echtzeitobjekterkennung auf eingebetteten Plattformen mit geringem Stromverbrauch. Dieses System arbeitet mit 22 FPS auf einem stromsparenden TDA2PX System on Chip (SoC) von Texas Instruments [46]. Darüber hinaus gibt es wissenschaftliche Arbeiten, die sich mit der YOLO Echtzeitobjekterkennung für Hardware mit beschränkten Ressourcen beschäftigen, wie zum Beispiel ein Objekterkennungsalgorithmus in Echtzeit, der für Nicht-GPU-Computer optimiert ist [39]. Zusätzlich wurde ein eingebettetes Fußgängererkennungssystem in Echtzeit unter Verwendung von YOLO, optimiert durch LNN, implementiert [45]. Darüber hinaus befassen sich auch einige wissenschaftliche Arbeiten mit dem Intel Neural Compute Stick 2. Zum Beispiel gibt es ein System, welches eine Vorhersage der Verkehrsdichte mittels der YOLO Objekterkennung mit Raspberry Pi 3 B+ und Intel NCS2 ausgibt [2]. Zusätzlich gibt es verwandte wissenschaftliche Arbeiten, die sich mit dem Ansatz der Simulation-zu-Realität Übertragung beschäftigen. Meistens sind es die Ansätze des tiefen bestärkendes Lernens (engl. Deep Reinforcement Learning). Zum Beispiel ein Roboter, der in einer simulierten Umgebung das Laufen lernt und es dann in der Realität anwendet [101]. Ebenfalls im Bereich der autonomen Navigationssysteme gibt es Roboter, die lernen auf Bürgersteigen zu laufen [48].

Der Ansatz dieser Forschungsarbeit in diesem Kapitel ist es, eine individuelle Echtzeitobjekterkennung für eigene Objekte für stromsparende IoT-Geräte angepasst an eine reale Umgebung aus dem Modellbaubereich zu entwickeln. Durch diese Echtzeitobjekterkennung kann anschließend in Kapitel 7 ein Radarkontrollsystem entwickelt werden. Zusätzlich ermöglicht dieser Ansatz, eine kostengünstige Echtzeitobjekterkennung zum Beispiel für Drohnen, Überwachungskameras oder Wildkameras in kurzer Zeit zu entwickeln. Für die IoT-Hardware mit beschränkten Ressourcen wird der Raspberry Pi 3 B, der Raspberry Pi 4 B und der Intel Neural Compute Stick 2 (NCS2) als Hardwareerweiterung verwendet. Diese werden in Abschnitt 2.6 genauer vorgestellt.

6.3 Datensätze

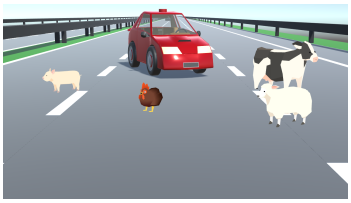
Vor dem Training der faltenden neuronalen Netzwerke müssen für jeden spezifischen Anwendungsfall annotierte Trainingsdaten gewonnen werden. Diese Trainingsdaten bilden die Grundlage für eine erfolgreiche Objekterkennung. Die nachfolgende Tabelle 6.1 stellt die erstellten Datensätze für die Objekterkennung auf einen Blick inklusive der Namen, Anzahl der Klassen, Bezeichnung der Klassen und Menge der einzelnen annotierten Daten auf einen Blick dar. Einige kleine Vorversuche haben gezeigt: Es genügt das Objekt in einer 360°-Ansicht zu fotografieren. Die Farbe der Objekte spielt bei der Objekterkennung ebenfalls keine Rolle. Die Objekte werden durch ihre unterschiedlichen Formen unterschieden. Somit werden, anders als bekannt, nicht viele Bilder pro Objekt benötigt. Es reicht nicht aus, das ConvNet-Modell zum Beispiel mit nur einem Modellauto auf einem weißen oder schwarzen Hintergrund zu trainieren. Das ConvNet-Modell erlernt die Unterschiede zwischen den verschiedenen Konturen der Objekte. Jedes weitere unbekannte Objekt

würde an dieser Stelle als das erlernte Objekt der ersten Klasse interpretiert werden. Somit werden die zu erlernenden Objekte mit verschiedenen unbekannten Objekten als Vergleich aufgenommen.

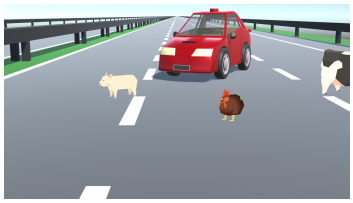
Tabelle 6.1: Erstellten Datensätze für die Objekterkennung mit TensorFlow. Die Auflösung der Bilder beträgt 1280 × 720 (Breite × Höhe) Pixel. Die Anzahl steht für die Menge der einzelnen annotierten Bilder. Die Annotation steht für die Bezeichnung der Klassen.

Nr.	**Name**	**Anzahl der Klassen**	**Annotation**	**Anzahl**
1	Sim 1	1	SimCar	300
2	Sim 2	2	SimCar, SimAnimal	1000
3	Sim 3	1	SimPig	300
4	Mod 1	1	PiCar	111
5	Mod 2	4	PiCar, ModCar, ModAnimal, ModPerson	200

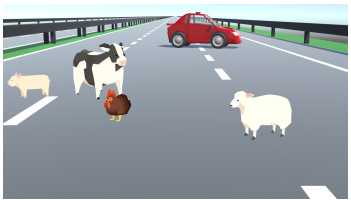
Die Datensätze 1 (Sim 1) und 2 (Sim 2) wurden mit dem bereits in Kapitel 3 vorgestellten Simulator erstellt und automatisch annotiert. Dieses Verfahren beschreibt der nächste Abschnitt 6.3.1. Der Datensatz 1 (Sim 1) enthält Bilder eines Simulationsautos *SimCar* mit verschiedenen unbekannten Objekten. Dieser Datensatz wird ebenfalls für die Sim-to-Real Übertragung in Abschnitt 6.5.1 verwendet. In dem Datensatz 1 (Sim 1) wird nur das Simulationsfahrzeug mit 300 Bildern annotiert (eine Klasse) und nicht die anderen im Bild sichtbaren Objekte. So kann das ConvNet-Modell den Unterschied zu den anderen Objekten erlernen. Der Datensatz 2 (Sim 2) wurde erweitert und enthält zwei Klassen mit den Bezeichnungen *SimCar* und *SimAnimal*. Die Menge ist hierbei 1000 Bilder. Der Datensatz 3 (Sim 3) wurde zusätzlich für die Sim-to-Real Übertragung aus dem Abschnitt 6.5.1 vorgesehen. Dieser Datensatz umfasst 300 Bilder mit einer Klasse mit der Bezeichnung *SimPig*. Die weiteren Datensätze 4 (Mod 1) und 5 (Mod 2) wurden mit einigen Objekten aus dem Modellbaubereich erstellt und manuell annotiert. Dieses Vorgehen beschreibt der Abschnitt 6.3.2. Der Datensatz 4 (Mod 1) enthält 111 Bilder mit einer Klasse von dem Modellfahrzeug *PiCar*. Der Datensatz 5 (Mod 2) wurde zusätzlich um drei weitere Klassen mit den Bezeichnungen *ModCar*, *ModAnimal* und *ModPerson* aus der realen Welt erweitert. Somit umfasst der Datensatz 5 (Mod 2) 200 Bilder mit vier Klassen mit den Bezeichnungen *PiCar*, *ModCar*, *ModAnimal* und *ModPerson*. Diese Bezeichnungen der Klassen stehen für das in Kapitel 5 vorgestellte Modellfahrzeug *PiCar*, ein Modellauto *ModCar*, ein Modelltier *ModAnimal* und eine Modellperson *ModPerson* aus dem Modellbaubereich. Die nachfolgende Abbildung 6.1 zeigt einige Ausschnitte der erstellten Trainingsdaten aus den vorgestellten Datensätzen. Die Aufteilung von Trainings- und Testdaten erfolgt mit 80 % für die Trainingsdaten und 20 % für die Testdaten. Darüber hinaus ist in Abbildung 6.1 folgendes zu sehen: Der Datensatz 1 (Abb. 6.1a, 6.1b und 6.1c) und Datensatz 4 (Abb. 6.1j, 6.1k und 6.1l) enthalten jeweils nur eine Klasse für das simulierte autonome Fahrzeug *SimCar* und das autonome Modellfahrzeug *PiCar*.



(a) Datensatz 1 (Sim 1) 1



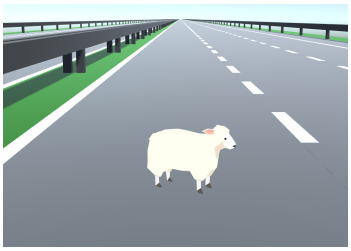
(b) Datensatz 1 (Sim 1) 2



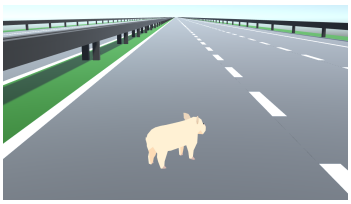
(c) Datensatz 1 (Sim 1) 3

(d) Datensatz 2 (Sim 2) 1

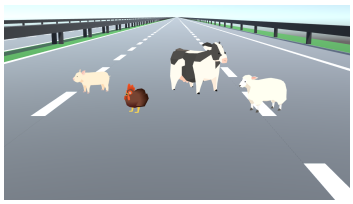
(e) Datensatz 2 (Sim 2) 2



(f) Datensatz 2 (Sim 2) 3



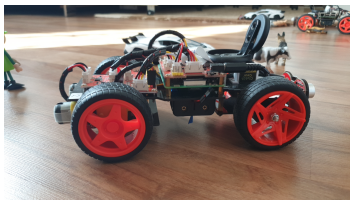
(g) Datensatz 3 (Sim 3) 1



(h) Datensatz 3 (Sim 3) 2

(i) Datensatz 3 (Sim 3) 3

(j) Datensatz 4 (Mod 1) 1



(k) Datensatz 4 (Mod 1) 2

(l) Datensatz 4 (Mod 1) 3

(m) Datensatz 5 (Mod 2) 1

(n) Datensatz 5 (Mod 2) 2

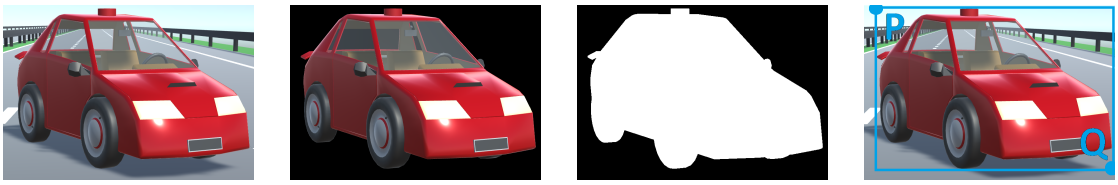
(o) Datensatz 5 (Mod 2) 3

Abbildung 6.1: Annotierten Datensätze für die Objekterkennung mit TensorFlow 1 und 2. Die Datensätze 1 bis 3 (Sim 1 - 3) sind aus dem Simulator mit simuliertem Fahrzeug und simulierten Modelltieren. Die Datensätze 4 (Mod 1) und 5 (Mod 2) sind aus dem Modellbaubereich aus der realen Umgebung mit zwei Modellfahrzeugen, einigen Modelltieren und Modellpersonen.

Anschließend können die Ergebnisse aus der Simulation und dem Modellbaubereich hinterher verglichen werden, um ein geeignetes Objekterkennungssystem für Hardware mit beschränkten Ressourcen zu finden. Diese Datensätze enthalten jeweils nur ein annotiertes Objekt zusammen mit verschiedenen unbekannten Objekten. So kann das ConvNet-Modell die Unterschiede zu den anderen Objekten erlernen. Wenn die Objekterkennung auf realen echten Daten implementiert werden soll, können die bereits veröffentlichten Datensätze wie Microsoft Common Objects in Context (MS COCO) [62] oder PASCAL Visual Object Classes (PASCAL VOC) [28] verwendet werden. Diese enthalten bereits Tausende annotierte reale Objekte. In dem in Kapitel 5 vorgestellte Modellfahrzeug *PiCar* wurde zusätzlich hinten eine schwarze Stoßstange verbaut, um eine gerade Fläche zum Messen für den Ultraschallsensor von dem hinteren Modellfahrzeug zu erreichen. Für die Entwicklung und Experimente in diesem und im nachfolgenden Kapitel 7, wurde die hintere Stoßstange entfernt und somit der ursprüngliche Zustand des Modellfahrzeugs wiederhergestellt. Dadurch bekommt das Modellfahrzeug mehr Merkmale bei der Objekterkennung aus der hinteren Sicht.

6.3.1 Automatische Annotation

Mit dem bereits in Kapitel 3 vorgestellten Simulator konnten Tausende von annotierten Trainingsdaten (Eingabe- und Ausgabedaten) automatisch erzeugt werden. Für die automatische Generierung der benötigten Datensätze wurden zwei virtuelle Kameras an der gleichen Stelle in einem simulierten Fahrzeug im Simulator installiert. Die erste Kamera konnte die virtuellen Umgebung komplett sehen (Abb. 6.2a). Die zweite Kamera sah nur das zu erkennende Objekt auf einem schwarzen Hintergrund (Abb. 6.2b). Diese beiden Bilder konnten für die weitere Generierung der Trainingsdaten verwendet werden. Das Bild der ersten virtuellen Kamera ist das Eingabebild für das faltende neuronale Netzwerk. Das Bild der zweiten virtuellen Kamera wurde zunächst in ein Graustufenbild umgewandelt. Um die Annotation der Daten (Ausgabedaten) zu erhalten, wurde anschließend aus dem Graustufenbild ein Binärbild erzeugt (Abb. 6.2c). Aus diesem Binärbild (schwarzer Hintergrund, weißes Objekt) konnten die Informationen über die Position des Objekts (oben links und unten rechts) extrahiert werden. So erhält man die Position des Objekts (Abb. 6.2d) für das Eingabebild als Koordinaten für $P(xMin, yMin)$ und $Q(xMax, yMax)$.



(a) Bild von der ersten Kamera.

(b) Bild von der zweiten Kamera.

(c) Umgewandeltes Binärbild.

(d) Position des Objekts (P und Q).

Abbildung 6.2: Erzeugung der automatisch annotierten Trainingsdaten in dem Simulator aus Kapitel 3 für die Objekterkennung.

Diese Koordinaten inklusive der Beschriftung des Objekts werden anschließend in einer XML-Datei gespeichert. Dazu gehören zum Beispiel ein simuliertes Auto in verschiedenen Farben und simulierte Tiere wie Kuh, Schwein, Hund, Schaf und Hahn. Somit ergeben sich die Beschriftungen *SimCar* für das simulierte Fahrzeug und *SimAnimal* für die simulierten Tiere. Die Annotation des Datensatzes 3 (Sim 3) funktionierte nach dem gleichen Prinzip. An dieser Stelle wurde das simulierte Schwein *SimPig* beschriftet. Der Vorteil dieses Ansatzes: Viele annotierte Trainingsdaten mit verschiedenen Objekten können in kurzer Zeit erstellt werden. Es können auch mehrere Objekte in einem Bild erstellt und automatisch annotiert werden. Da die Position der Objekte bekannt ist, können diese Objekte verschoben oder ebenfalls untereinander ausgetauscht werden. Um viele verschiedene Trainingsdaten zu erzeugen, kann zusätzlich die Position oder Größe der Objekte verändert werden. Auch der Austausch des Hintergrundbilds ist an dieser Stelle denkbar. Die Auflösung der automatisch erzeugten Bilder aus der Simulation beträgt 1280 × 720 (Breite × Höhe) Pixel. Diese Auflösung ist größer als die Auflösung der Eingabebilder der ConvNet-Modelle. Somit können diese Bilder anschließend auf die passende Eingabegröße komprimiert werden.

6.3.2 Manuelle Annotation

Für die Datensätze aus dem Modellbaubereich wurden verschiedene Bilder der Modellobjekte aus der echten Umgebung aufgenommen. Dazu gehören zum Beispiel zwei bereits vorgestellte Modellfahrzeuge (PiCar), zwei Modellautos (weißer Lamborghini), sechs Modelltiere (Kuh, Schwein, Hund, Ziegenbock, Stier, Schaf) und drei Modellpersonen (zwei Pirate und ein Ritter). Diese Modellobjekte aus dem Modellbaubereich sind in der Abbildung 6.3 zu sehen.

(a) Zwei PiCars, zwei Modellautos, sechs Modelltiere und drei Modellpersonen.

(b) Ein Modellauto, sechs Modelltiere und drei Modellpersonen.

Abbildung 6.3: Modellobjekte für die manuelle Annotation aus dem Modellbaubereich für die Objekterkennung (Ausschnitt aus dem Datensatz 5).

Anschließend wurden diese Bilder manuell mit der Software *LabelImg* beschriftet [110]. Somit ergeben sich die Klassenbeschriftungen *PiCar*, *ModCar*, *ModAnimal* und *ModPerson*. Dieses Werkzeug vereinfacht das Zeichnen des Rechtecks um ein Objekt und bestimmt automatisch die Koordinaten für $P(xMin, yMin)$ und $Q(xMax, yMax)$. Diese Koordinaten können anschließend in einer XML-Datei mit demselben Namen wie die Eingabedatei für das Training mit TensorFlow als Annotation gespeichert werden. Mit

diesem Werkzeug ist es ebenfalls möglich, die Annotationen in verschiedenen Formaten, zum Beispiel in dem YOLO-Format, zu erzeugen [115]. Die Auflösung der manuell erzeugten Bilder aus dem Modellbaubereich beträgt 1280 × 720 (Breite × Höhe) Pixel. Diese Auflösung ist an dieser Stelle ebenfalls größer als die Auflösung der Eingabebilder der zu trainierten ConvNet-Modelle. Somit können diese Bilder anschließend auf die passende Eingabegröße komprimiert werden.

6.4 Funktionsweise

Nachdem die notwendige Datenbeschaffung für die individuelle Objekterkennung auf den Bildern aus der Simulation und Bildern aus dem Modellbaubereich abgeschlossen ist, kann mit der eigentlichen Objekterkennung in diesen Datensätzen begonnen werden. Somit präsentieren die nachfolgenden Abschnitte dieses Kapitels die Funktionsweise der Objekterkennung auf einer Hardware mit einer leistungsstarken Grafikkarte, auf einer Hardware mit beschränkten Ressourcen ohne Hardwareerweiterung und auf einer Hardware mit beschränkten Ressourcen mit der Intel Neural Compute Stick 2 (NCS2) Hardwareerweiterung. Dabei wird der Ansatz der Objekterkennung mit TensorFlow und der Programmierschnittstelle für die Objekterkennung mit TensorFlow verfolgt. Mehr dazu beschreiben die Abschnitte 2.5.5 und 2.5.6.

6.4.1 Erkennung der Objekte ohne Hardwareerweiterung

Um die Objekterkennung mit TensorFlow zuerst auf einer leistungsstarken Hardware mit Grafikkarte und anschließend auf Hardware mit beschränkten Ressourcen durchführen zu können, wurde als erstes nach bereits vortrainierten Modellen gesucht, die sich bereits bei der Objekterkennung mit echten Daten etabliert haben. Wie bereits erwähnt, stehen in dem TensorFlow 2 Erkennungsmodellzoo verschiedene ConvNet-Modelle für TensorFlow 2 zur Verfügung. Mehr dazu beschreibt der Abschnitt 2.5.6. Diese wurden bereits auf dem Microsoft Common Objects in Context (MS COCO 17) Datensatz vortrainiert und können 90 Objekte erkennen und klassifizieren. Ebenfalls sind die Geschwindigkeit und die Genauigkeit dieser Modelle angegeben. Dadurch kann bereits eine erste Vorauswahl anhand der Geschwindigkeit und der Genauigkeit der ConvNet-Modelle getroffen werden. Um eine erste Vorauswahl zu treffen und diese Ergebnisse anhand der Laufzeitmessungen zu überprüfen, wurden einige Vorexperimente zur Bestimmung der schnellen TensorFlow 2 ConvNet-Modelle durchgeführt. Die Laufzeitmessungen der vortrainierten TensorFlow Modelle wurden auf der NVIDIA GeForce GTX 1660 Ti Grafikkarte (GPU) des Dell G3 15 3590 Notebooks durchgeführt. Weitere Informationen zu dieser Hardware gibt der Abschnitt 2.6.1. Diese Vorexperimente wurden auf dem vorgestellten Datensatz 5 (Mod 2) mit 200 Bildern erledigt. Die nachfolgende Tabelle 6.2 zeigt die Unterschiede der Geschwindigkeit der ausgewählten ConvNet-Modelle. Die Erkennungsgeschwindigkeit ist in der Tabelle 6.2 als Bilder pro Sekunde (FPS) angegeben. Die Genauigkeit wird als COCO mittlere durchschnittliche Genauigkeit angegeben [40]. Diese Metrik wird in dem Abschnitt 2.3.4 genauer erklärt. Wie erwartet, bestätigt dieser kleine Vorexperiment: Eine kleinere Auflösung des Modells führt zu einer schnelleren Verarbeitung der Bilder

(Vergleich zwischen Nr. 2 und 6 in Tab. 6.2). Nach einem Vergleich der Genauigkeit (mAP) und der Geschwindigkeit (FPS) zeigt sich, dass eine höhere Auflösung der Eingabebilder der ConvNet-Modelle eine genauere Objekterkennung ermöglicht (Vergleich zwischen Nr. 3 und 4 in Tab. 6.2).

Tabelle 6.2: Vorexperiment zur Bestimmung der schnellen TensorFlow 2 ConvNet-Modelle. Die Laufzeit ist als Bilder pro Sekunde (FPS) angegeben. Die Genauigkeit ist als COCO mittlere durchschnittliche Genauigkeit angegeben.

Nr.	**Name**	**Auflösung [Pixel]**	**Genauigkeit [mAP]**	**Laufzeit [FPS]**
1	SSD MobileNet V2	320 × 320	20,2	16,4
2	EfficientDet D0	512 × 512	33,6	5,7
3	SSD ResNet50 V1 FPN	640 × 640	34,3	4,7
4	SSD ResNet50 V1 FPN	1024 × 1024	38,3	2,3
5	CenterNet HourGlass104	1024 × 1024	44,5	1,7
6	EfficientDet D4	1024 × 1024	48,5	1,5

Aus diesem Grund musste ein Gleichgewicht zwischen der Genauigkeit und der Laufzeit gefunden werden, um die Objekterkennung auf Hardware mit beschränkten Ressourcen nutzen zu können. Weitere Experimente der Laufzeiten beschreibt der Abschnitt 6.5. Durch dieses Vorexperiment der Geschwindigkeit der ConvNet-Modelle ist die Wahl auf die Modelle *SSD MobileNet V2* und *EfficientDet D0* gefallen. Um die ausgewählten TensorFlow 2 ConvNet-Modelle auf anderen Datensätzen zu vergleichen, wurden die *SSD MobileNet V2* und *EfficientDet D0* Modelle auf allen bereits in diesem Kapitel vorgestellten Datensätzen trainiert und evaluiert. Die Auflösungen der einzelnen Modelle wurden aus dem TensorFlow 2 Modellzoo übernommen. Somit wurde die Auflösung 512 × 512 Pixel für das *EfficientDet D0* ConvNet-Modell und 320 × 320 Pixel für das *SSD MobileNet V2* ConvNet-Modell beibehalten. Zusätzlich wurde die Auflösung der *SSD MobileNet V2* ConvNet-Modelle auf 224 × 224 Pixel verkleinert, da dieses Modell bereits gute Geschwindigkeit bei der Auswertung im Vorexperiment erzielt hat.

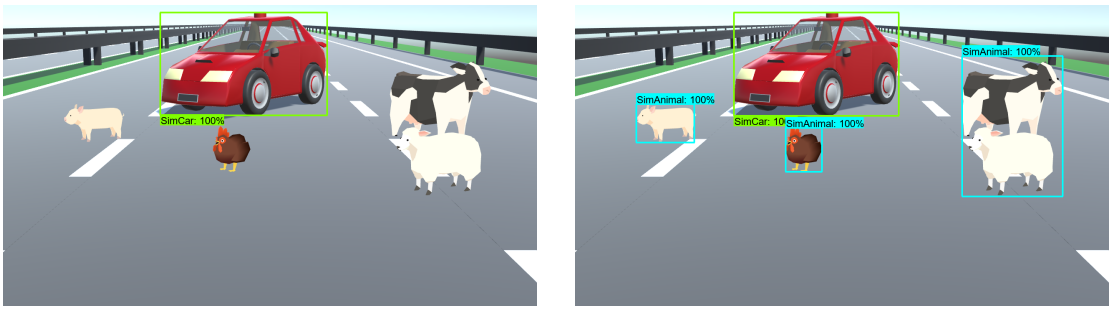
Die Basis-Lernrate (engl. Base Learning Rate) wurde auf $0,008$ gesetzt. Die Aufwärm-Lernrate (engl. Warmup Learning Rate) beträgt $0,0001$. Die Stapelgröße (engl. Batch Size) ist 4 an dieser Stelle. Standardmäßig sind diese angegebenen Werte der Parameter höher. Der MS COCO-Datensatz enthält viel mehr Trainingsdaten. Daher werden in diesem Szenario diese Parameterwerte reduziert. Die Schritte (engl. Steps) geben die Gesamttrainingsschritte der ausgewählten Modelle an. Dieser Parameter ist wichtig, um eine Überanpassung und Unteranpassung des Modells zu verhindert. Mehr dazu wird in dem Abschnitt 2.2.1 erklärt. Alle Trainingsversuche wurden mit den gleichen Einstellungen der Parameter und den vorgestellten Trainingsdaten durchgeführt. Das Training der Modelle wurde beendet, sobald sich die Verlustfunktion nicht mehr verändert hat. Die Genauigkeit der Objekterkennung ist von der Netzwerkarchitektur der ConvNet-Modelle abhängig. Diese ausgewählten und trainierten Modelle zeigt die nachfolgende Tabelle 6.3 im Überblick. Zusätzlich zeigt ein Vergleich der Genauigkeiten, dass die

Modelle mit höherer Eingabebildauflösung genauer sind (Vergleich zwischen Nr. 4, Nr. 8 und Nr. 10 in Tab. 6.3).

Tabelle 6.3: Überblick der trainierten *EfficientDet D0* und *SSD MobileNet V2* TensorFlow 2 ConvNet-Modelle mit verschiedener Auflösung der Eingabebilder. Die Genauigkeit ist als COCO mittlere durchschnittliche Genauigkeit (mAP) angegeben.

Nr.	**Modell**	**Auflösung [Pixel]**	**Schritte**	**Datensatz**	**Genauigkeit [mAP]**
1	EfficientDet D0	512 × 512	150 k	Sim 1	95,8
2	EfficientDet D0	512 × 512	100 k	Sim 2	87,1
3	EfficientDet D0	512 × 512	150 k	Mod 1	91,7
4	EfficientDet D0	512 × 512	100 k	Mod 2	76,4
5	SSD MobileNet V2	320 × 320	150 k	Sim 1	93,4
6	SSD MobileNet V2	320 × 320	100 k	Sim 2	79,6
7	SSD MobileNet V2	320 × 320	150 k	Mod 1	92,1
8	SSD MobileNet V2	320 × 320	100 k	Mod 2	69,3
9	SSD MobileNet V2	224 × 224	150 k	Mod 1	90,9
10	SSD MobileNet V2	224 × 224	100 k	Mod 2	60,6

Zuerst wurden die mit den Simulationsdaten trainierten Modelle ausgewertet. Die nachfolgende Abbildung 6.4 stellt die Objekterkennung mit dem *EfficientDet D0 512 × 512* Modell (Nr. 1 und 2 in Tab. 6.3) auf dem Datensatz mit den Simulationsdaten grafisch dar. Die Position und Größe des vom Modell erkannten Objekts wird durch ein Rechteck dargestellt. In grün wird die Klasse *SimCar* und in blau die Klasse *SimAnimal* dargestellt.



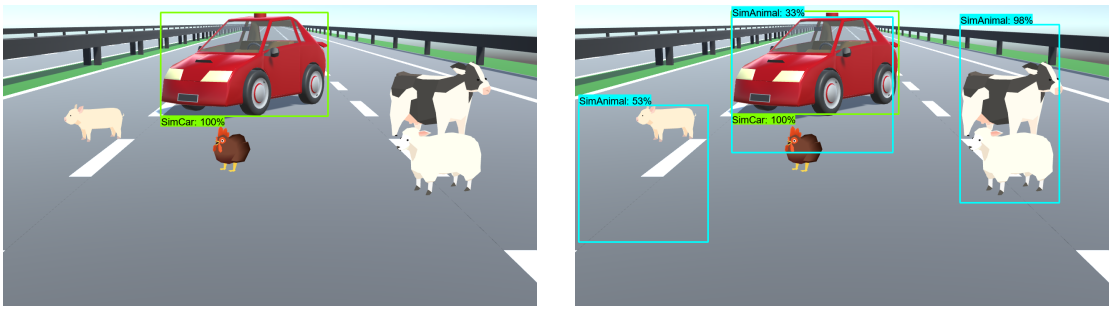
(a) Datensatz: Sim 1. Klasse: *SimCar*.

(b) Datensatz: Sim 2. Klassen: *SimCar*, *SimAnimal*.

Abbildung 6.4: Objekterkennung mit dem *EfficientDet D0 512 × 512* Modell (Nr. 1 und 2 in Tab. 6.3).

Wie man erkennen kann, wird das simulierte Fahrzeug *SimCar* sehr gut erkannt (Abb. 6.4a). Die Position und Größe des Objekts stimmt ebenfalls überein. Die simulierten Tiere werden wie gewollt nicht erkannt, da der Datensatz 1 (Sim 1) nur die Klasse

SimCar enthält. Das Modell (Nr. 1 in Tab. 6.3) hat somit die Unterschiede zwischen den verschiedenen Konturen der Objekte erlernt. Anschließend kann das nächste Modell untersucht werden. Die vorherige Abbildung 6.4b zeigt eine Objekterkennung mit dem *EfficientDet D0 512 × 512* Modell (Nr. 2 in Tab. 6.3) auf dem Datensatz 2 (Sim 2). Aus dieser Grafik erkennt man, dass die verschiedenen Objekte richtig erkannt und klassifiziert werden. Die Position und Größe dieser Objekte stimmt ebenfalls überein. Zusätzlich ist zu sehen, dass die simulierte Kuh und das simulierte Schaf als ein Objekt der Klasse *SimAnimal* erkannt werden. An dieser Stelle sind die Objekte zu nah bei einander und der gleiche weiße Farbverlauf erschwert diese Erkennung. Ebenso hat die Fahrbahnmarkierung eine ähnliche Farbe. Daraus sind die Konturen dieser Objekte nur schwer zu erkennen. Die Objekterkennung mit dem *SSD MobileNet V2 320 × 320* Modell auf dem Datensatz 1 (Sim 1) mit einer Klasse *SimCar* zeigt ebenfalls ein sehr gutes Ergebnis (Abb. 6.5a). Die Position, die Größe und die Klassifizierung des Objekts stimmt ebenfalls überein. In der Abbildung 6.5b kann man erkennen, dass durch eine andere Netzwerkarchitektur und Auflösung der Eingabebilder des Modells die Erkennung der Objekte auf dem Datensatz 2 (Sim 2) mit zwei Klassen *SimCar* und *SimAnimal* schlechter ist. Dies bestätigt auch der Vergleich der Genauigkeiten (mAP) dieser Modelle (Vergleich zwischen Nr. 2 und 6 in Tab. 6.3).



(a) Datensatz: Sim 1.
Klasse: *SimCar*.

(b) Datensatz: Sim 2.
Klassen: *SimCar, SimAnimal*.

Abbildung 6.5: Objekterkennung mit dem *SSD MobileNet V2 320 × 320* Modell (Nr. 5 und 6 in Tab. 6.3).

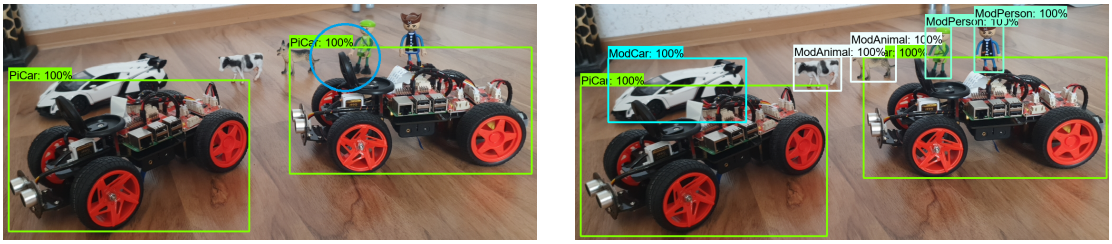
Das *EfficientDet D0 512 × 512* Modell trainiert auf dem Datensatz 2 (Sim 2) erreicht eine Genauigkeit von $87,1$ % auf den Testbildern. Das *SSD MobileNet V2 320 × 320* Modell hingegen, erreicht eine Genauigkeit von $79,6$ % auf demselben Datensatz 2. An dieser Stelle stimmt teilweise die Klassifikation der Objekte nicht. Zusätzlich stimmt die Position und Größe der simulierten Tiere nicht überein. Der Datensatz 2 (Sim 2) enthält pro Eingabebild jeweils nur ein annotiertes Objekt. Sobald die Objekte zusammen auf einem Eingabebild zu sehen sind, wird die Objekterkennung erschwert. Hingegen einzelne Objekte auf den Bildern werden sehr gut erkannt. Die Abbildung 6.6 veranschaulicht dieses Beispiel. Jedes dieser simulierten Objekte (Schaf, Schwein, Kuh, Hahn) wird richtig klassifiziert. Ebenfalls stimmt die Position und Größe des Objekts. Damit die verschiedenen Objekte auch zusammen erkannt werden können, sollte der Datensatz

um weitere verschiedene Eingabebilder mit mehreren verschiedenen Objekten erweitert werden. Diese Erkenntnis wird auf die Datensätzen 4 (Mod 1) und 5 (Mod 2) aus dem Modellbaubereich übernommen, um zu sehen ob diese Vermutung übereinstimmt.

(a) Schaf links (b) Schaf rechts (c) Schwein links (d) Schwein rechts (e) Kuh links (f) Kuh rechts (g) Hahn links (h) Hahn rechts

Abbildung 6.6: Erkennung der einzelnen Objekte der Klasse *SimAnimal* pro Eingabebild mit dem *SSD MobileNet V2 320 × 320* Modell (Nr. 6 in Tab. 6.3).

Weitere Tests zeigen ebenfalls, dass diese Modelle teilweise Objekte an Stellen erkennen, an denen keine Objekte zu sehen sind. Diese werden dann als *SimCar* klassifiziert, da das die erste Ausgabeklasse ist. Um dieses Problem zu lösen, werden mehr Trainingsdaten mit den zu erkennenden Objekten und weiteren Objekten mit verschiedenen Formen benötigt. Dadurch kann das Modell den Unterschied der Konturen dieser Objekte erlernen. Um ein geeignetes Modell für die Objekterkennung aus dem Modellbaubereich zu erhalten, wurde anschließend mit der Objekterkennung auf den Datensätzen 4 (Mod 1) und 5 (Mod 2) und den *EfficientDet D0 512 × 512* und *SSD MobileNet V2 320 × 320* Modellen fortgesetzt. An dieser Stelle sollte ein ConvNet-Modell gefunden werden, welches möglichst gut die Objekte erkennt und auf der anderen Seite ziemlich schnell in dieser Erkennung ist. Die Abbildung 6.7a zeigt die Erkennung der Objekte mit dem *EfficientDet D0 512 × 512* Modell (Nr. 3 in Tab. 6.3) aus dem Modellbaubereich mit einer Klasse *PiCar*. Die Rechtecke zeigen die Erkennung und Klassifizierung des Objekts durch das Modell. An dieser Stelle wird nur das Modellfahrzeug *PiCar* und keine anderen Objekte erkannt und klassifiziert. Ebenfalls stimmt die Position und Größe des erkannten Objekts überein.



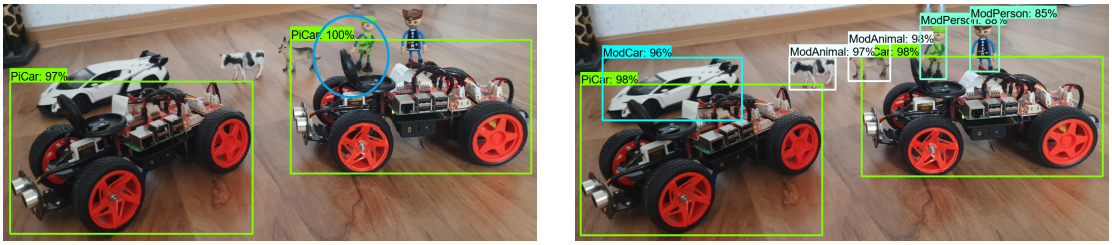
(a) Datensatz: Mod 1. Klasse: *PiCar*.

(b) Datensatz: Mod 2. Klassen: *PiCar, ModCar, ModAnimal, ModPerson*.

Abbildung 6.7: Objekterkennung mit dem *EfficientDet D0 512 × 512* Modell (Nr. 3 und 4 in Tab. 6.3).

Wie in Abbildung 6.7b zusehen ist, werden die Objekte aus dem Datensatz 5 (Mod 2) mit dem *EfficientDet D0 512 × 512* Modell (Nr. 4 in Tab. 6.3) ebenfalls richtig erkannt und klassifiziert. Die Position und Größe der verschiedenen Objekte stimmt ebenfalls überein. Die verschiedenen Farben der Rechtecke symbolisieren die verschiedenen

vier Klassen *PiCar*, *ModCar*, *ModAnimal* und *ModPerson*. An dieser Stelle kann eine Genauigkeit von 76,4 % erreicht werden (Nr. 4 in Tab. 6.3). Zum Vergleich wurde das *SSD MobileNet V2 320 × 320* Modell mit der gleichen Ausgabeklasse *PiCar* ausgewertet (Abb. 6.8a). Wie man erkennen kann, ist dieses Modell weniger genau bei der Erkennung des Modellfahrzeugs *PiCar*. Dieses ist in den Abbildungen 6.7a und 6.8a hellblau markiert. Dies spiegelt sich jedoch nicht immer in der Genauigkeit (mAP) wider (Vergleich zwischen Nr. 3 und 7 in Tab. 6.3). Das Modell *SSD MobileNet V2 320 × 320* erreicht bei diesem Datensatz 4 (Mod 1) sogar eine bessere durchschnittliche Genauigkeit gegenüber dem *EfficientDet D0 512 × 512* Modell. Aus diesem Grund sollten die trainierten Modelle mit verschiedenen Testdaten für jeden bestimmten Zweck evaluiert und untersucht werden.



(a) Datensatz: Mod 1. Klasse: *PiCar*.

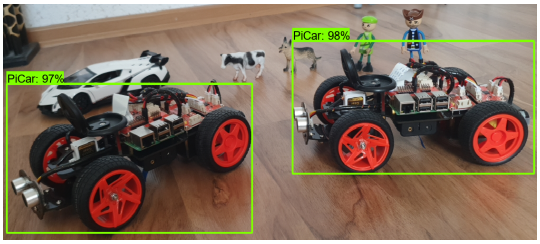
(b) Datensatz: Mod 2. Klassen: *PiCar, ModCar, ModAnimal, ModPerson*.

Abbildung 6.8: Objekterkennung mit dem *SSD MobileNet V2 320 × 320* Modell (Nr. 7 und 8 in Tab. 6.3).

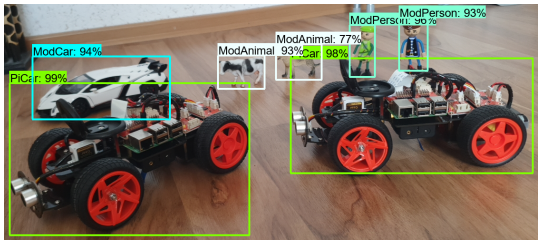
Die weitere Auswertung erfolgt auf dem Datensatz 5 (Mod 2) und dem *SSD MobileNet V2 320 × 320* Modell mit vier Ausgabeklassen *PiCar*, *ModCar*, *ModAnimal* und *ModPerson*. Die prozentuale Erkennung der einzelnen Klassen ist geringer gegenüber dem *EfficientDet D0 512 × 512* Modell. Doch die Erkennung der Position und Größe der Objekte stimmt exakt überein. Auch die Klassifikation der einzelnen Objekte stimmt überein (Abb. 6.8b). Hier kann eine Genauigkeit von 69,3 % erreicht werden (Nr. 8 in Tab. 6.3). Diese Erkennung ist für den Zweck der Objekterkennung auf Hardware mit beschränkten Ressourcen völlig akzeptabel und für die Anwendung dieser Forschungsarbeit ausreichend.

Zusätzlich wurde das *SSD MobileNet V2 320 × 320* Modell in ein *SSD MobileNet V2 224 × 224* Modell umgewandelt. Die Eingabeauflösung des ConvNet-Modells wurde an dieser Stelle verkleinert. Dieses Modell war im TensorFlow 2 Modellzoo zum Zeitpunkt der Entwicklung dieser Forschungsarbeit nicht vorhanden. Daher ist es interessant zu sehen, wie diese verkleinerten Modelle in Bezug auf die Genauigkeit und die Laufzeit abschneiden. Die Nummern 9 und 10 in Tabelle 6.3 zeigen einen Überblick über die Genauigkeit dieser Modelle. Mit dem *SSD MobileNet V2 224 × 224* Modell kann eine Genauigkeit von 90,9 % auf dem Datensatz 4 (Mod 1) und eine Genauigkeit von 60,6 % auf dem Datensatz 5 (Mod 2) erreicht werden. Zum Vergleich, erreicht das *SSD MobileNet V2 320 × 320* Modell 92,1 % und 69,3 % auf den gleichen Datensätzen. Die nachfolgende Abbildung 6.9 zeigt die Objekterkennung des *SSD MobileNet V2 224 × 224* Modells. Man kann erkennen, dass die Erkennung der einzelnen Objekte ziemlich genau ist. Die Position und Größe der Objekte stimmt ebenfalls überein. Wie man ebenfalls sieht, ist die

prozentuale Klassifikation geringer gegenüber der bereits ausgewerteten Modellen, doch diese Erkennung ist ebenfalls für den Modellbaubereich ausreichend. Durch die geringere Eingabeauflösung der Modelle, steigt die Laufzeit für die Erkennung der Objekte. Mehr zu den Laufzeiten beschreibt der Abschnitt 6.5.



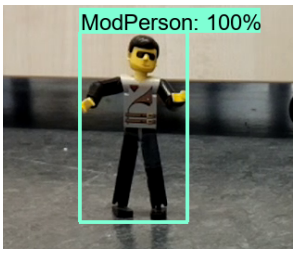
(a) Datensatz: Mod 1. Klasse: *PiCar*.



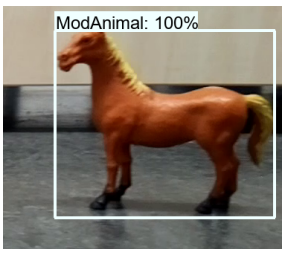
(b) Datensatz: Mod 2. Klasse: *PiCar, ModCar, ModAnimal, ModPerson*.

Abbildung 6.9: Objekterkennung mit dem *SSD MobileNet V2 224 × 224* Modell (Nr. 9 und 10 in Tab. 6.3).

Als nächstes werden die vorgestellten Modelle auf noch nie gesehenen Objekten aus dem Modellbaubereich ausgewertet. Auch die noch nie gesehenen Objekte aus dem Modellbaubereich konnten von den vorgestellten Modellen *EfficientDet D0 512 × 512* (Abb. 6.10), *SSD MobileNet V2 320 × 320* (Abb. 6.11) und *SSD MobileNet V2 224 × 224* (Abb. 6.12) erkannt werden. Zum Beispiel werden eine Lego-Person, ein Modellpferd und ein blauer BMW von diesen drei Modellen erkannt. Die Erkennung des *EfficientDet D0 512 × 512* Modells bei noch nie gesehenen Objekten aus dem Modellbaubereich ist sehr akkurat (Abb. 6.10). Ebenfalls stimmt die erkannte Position und Größe dieser Objekte überein. Die prozentuale Klassifikation liegt hier jeweils bei 100 % pro erkannte Klasse *ModPerson*, *ModAnimal* und *ModCar*.



(a) Modellperson.



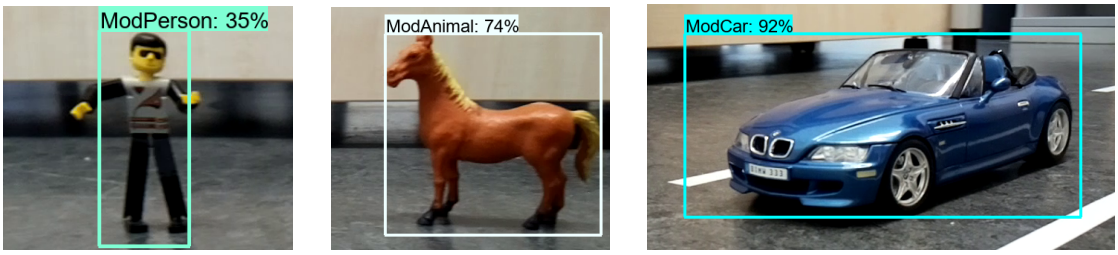
(b) Modellpferd.

(c) Modellauto.

Abbildung 6.10: Objekterkennung mit dem *EfficientDet D0 512 × 512* Modell (Nr. 4 in Tab. 6.3) bei nie gesehenen Objekten.

Beim Training haben die ConvNet-Modelle bereits verschiedene Figuren von Modellautos, Modelltieren und Modellpersonen gesehen. Diese Modelle lernten die ähnlichen Formen der Objekte und nicht nur die Eingabebilder aus dem Trainingsdatensatz. Außerdem wird jedes andere Objekt mit einer unbekannten Form als *PiCar* erkannt. Die Modelle

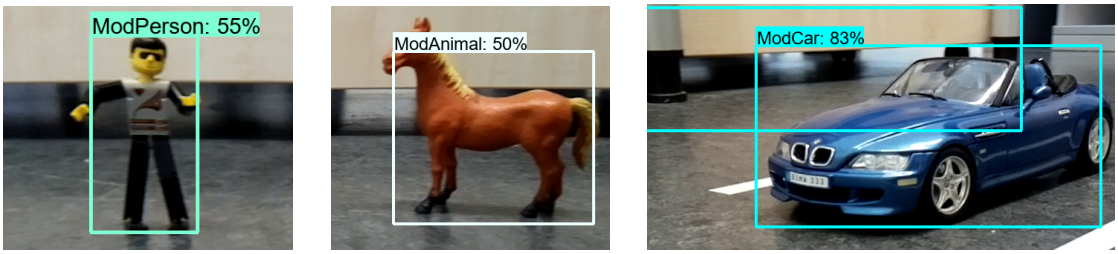
haben beim Training noch nie ein Objekt mit einer ähnlichen Form (Kontur) gesehen. Daher kann das Objekt nicht eindeutig klassifiziert werden. Infolgedessen wird diesem unbekannten Objekt die erste Ausgabeklasse *PiCar* zugewiesen. Dieses Problem kann durch eine Vergrößerung des Datensatzes gelöst werden. Es werden mehr Trainingsdaten mit zum Beispiel dem *PiCar* und vielen verschiedenen Objekten (verschiedenen Formen) benötigt. So kann der Unterschied zu anderen Konturen erlernt werden. Wie zu erwarten war, ist die Erkennung des *SSD MobileNet V2 320 × 320* Modells etwas schlechter (Abb. 6.11). Die erkannte Größe des Objekts stimmt teilweise nicht überein. Ebenfalls ist die prozentuale Klassifikation geringer.



(a) Modellperson. (b) Modellpferd. (c) Modellauto.

Abbildung 6.11: Objekterkennung mit dem *SSD MobileNet V2 320 × 320* Modell (Nr. 8 in Tab. 6.3) bei nie gesehenen Objekten.

In der Abbildung 6.12 kann man erkennen, dass die Erkennung des Modells *SSD MobileNet V2 224 × 224* noch ungenauer gegenüber der anderen beiden Modellen ist. Teilweise ist die prozentuale Klassifikation der Klassen geringer und die Größe der erkannten Objekte stimmt nicht exakt überein. Um diese Ergebnisse zu verbessern, können die Datensätze um mehrere verschiedene Objekte erweitert werden.



(a) Modellperson. (b) Modellpferd. (c) Modellauto.

Abbildung 6.12: Objekterkennung mit dem *SSD MobileNet V2 224 × 224* Modell (Nr. 10 in Tab. 6.3) bei nie gesehenen Objekten.

Die Abschnitte 6.5.2 und 6.5.3 stellen die Experimente der Laufzeitmessungen der vorgestellten Modelle. An dieser Stelle wird auf die Laufzeitmessungen aus diesen Experimenten vorgegriffen. Mit dem *SSD MobileNet V2 224 × 224* Modell können auf einer leistungsstarken Hardware mit einer Grafikkarte bis zu 40 Bilder pro Sekunde erreicht werden (Exp. Nr. 5 in Tab. 6.7). Das gleiche *SSD MobileNet V2 224 × 224 Q* Modell erreicht

eine Geschwindigkeit von ca. 3 Bilder pro Sekunde auf einer Hardware mit beschränkten Ressourcen (Exp. Nr. 7 in Tab. 6.8).

6.4.2 Erkennung der Objekte mit Hardwareerweiterung

Nach der Erkennung der Objekte ohne Hardwareerweiterung ist die Idee der Objekterkennung mit Hardwareerweiterung aufgekommen. Dafür wurde der Intel Neural Compute Stick 2 (NCS2) als Hardwareerweiterung für Hardware mit beschränkten Ressourcen verwendet. Mehr dazu beschreibt der Abschnitt 2.6.4. Zusätzlich stellte sich die folgende Forschungsfrage: *Kann die bereits vorgestellte Objekterkennung für Hardware mit beschränkten Ressourcen von ca. 3 FPS auf eine Echtzeitobjekterkennung von ca. 30 FPS auf Hardware mit beschränkten Ressourcen durch die Intel NCS2 Hardwareerweiterung beschleunigt werden?*

Um diese Frage zu beantworten, wurde ein neuer Ansatz für die Objekterkennung mit Hardwareerweiterung ausprobiert. Der Intel NCS2 Stick kann durch seine verbaute Bildverarbeitungseinheit und implementierte Software die OpenVINO-Modelle verarbeiten. Mehr dazu beschreibt der Abschnitt 2.6.4. Um die in Abschnitt 6.4.1 trainierten TensorFlow 2 Modelle auf dieser Hardwareerweiterung laufen zu lassen, mussten diese Modelle nach OpenVINO konvertiert werden. Doch zum Zeitpunkt der Entwicklung war dieses Vorgehen leider nicht möglich. Die Konvertierung nach OpenVINO konnte nur mit den TensorFlow 1 Modellen durchgeführt werden. Somit war der erste Schritt neue schnelle TensorFlow 1 Modelle zu finden, die für eine Objekterkennung auf Hardware mit beschränkten Ressourcen mit Hardwareerweiterung geeignet sind. Für TensorFlow 1 sind ebenfalls einige bereits vortrainierte ConvNet-Modelle verfügbar. Diese Modelle haben sich bereits in der Objekterkennung mit realen Daten bewährt. In dem TensorFlow 1 Modellzoo sind mehrere dieser ConvNet-Modelle für TensorFlow 1 verfügbar. Diese Modelle wurden bereits mit dem Microsoft Common Objects in Context (MS COCO 17) Datensatz vortrainiert und können 90 verschiedene Objekte erfolgreich erkennen und klassifizieren. Die Genauigkeit und die Laufzeit dieser Modelle ist ebenfalls angegeben. Diese Angaben ermöglichen einen ersten Vergleich der verfügbaren ConvNet-Modelle.

An dieser Stelle wurden die Modelle *SSD MobileNet V2* und *SSD Lite MobileNet V2* ausgewählt, da diese Modelle schnell und präzise sind. Dies bestätigen ebenfalls die zuvor durchgeführten Laufzeitmessungen ohne Hardwareerweiterung mit den *SSD MobileNet V2* Modellen in Abschnitt 6.5.3. Darüber hinaus können diese Modelle im nächsten Schritt mit den eigenen in Abschnitt 6.3 vorgestellten Datensätzen durch die Programmierschnittstelle für die Objekterkennung mit TensorFlow trainiert werden. Die Eingabeauflösung dieser faltenden neuronalen Netzwerke kann ebenfalls angepasst werden. Bei diesem Ansatz werden nur die Datensätze 4 (Mod 1) und 5 (Mod 2) aus dem Modellbaubereich betrachtet, da die Echtzeitobjekterkennung an dieser Stelle nur für die Auswertung realer Modellobjekte aus dem Modellbaubereich auf Hardware mit beschränkten Ressourcen relevant ist. Um diese Modelle auf dem Raspberry Pi mit der Intel NCS2 Hardwareerweiterung für eigene Objekte zu evaluieren, werden die Modelle zunächst mit *TensorFlow 1.15.5* und den vorgestellten Trainingsdaten aus dem Modellbaubereich trainiert. Anschließend müssen die trainierten ConvNet-Modelle als

ein eingefrorener Inferenzgraph exportiert werden. Nachfolgend können diese Modelle mit der *OpenVINO 2021.3* Bibliothek in OpenVINO-Modelle umgewandelt werden, um sie auf dem Intel Neural Compute Stick 2 auszuführen [82]. Die Skriptsprache *Python 3.7.3* wurde verwendet, um die konvertierten OpenVINO-Modelle auf der Hardware mit beschränkten Ressourcen mit der Intel NCS2 Hardwareerweiterung auszuführen.

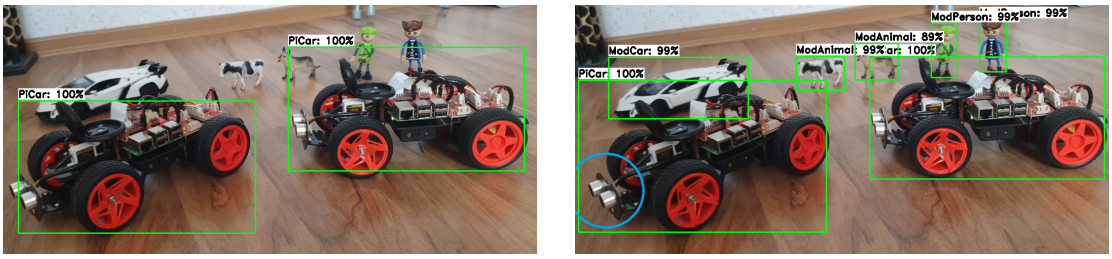
Interessanterweise sind die *SSD MobileNet V2* und *SSD Lite MobileNet V2* Modelle nicht als unterstützte Netzwerkarchitektur auf der offiziellen Intel Movidius-Webseite aufgeführt [71]. Zum Zeitpunkt der Experimente und Implementierung sind nur die *SSD MobileNet V1* und die *Inception* Modelle als unterstützte Netzwerkarchitektur auf der offiziellen Intel Movidius-Webseite aufgeführt. Das bedeutet, dass die Version 2 (*V2*) der *SSD MobileNet* Modelle nicht auf der Intel NCS2 Hardwareerweiterung lauffähig ist. Dennoch wurde an dieser Stelle der Ansatz für die Entwicklung der Echtzeitobjekterkennung mit den *SSD MobileNet V2* und *SSD Lite MobileNet V2* Modellen verfolgt. Laut dem TensorFlow 1 Modellzoo sind die *SSD MobileNet V2* Modelle teilweise schneller und genauer gegenüber den *SSD MobileNet V1* Modellen. Somit war es zusätzlich das Ziel, diese Modelle mit OpenVINO auf dem Intel NCS2 Stick laufen zu lassen. Die Tabelle 6.4 zeigt einen Überblick über die Genauigkeit der trainierten *SSD MobileNet V2* und *SSD Lite MobileNet V2* Modelle mit verschiedener Auflösung.

Tabelle 6.4: Überblick der trainierten *SSD MobileNet V2* und *SSD Lite MobileNet V2* TensorFlow 1 ConvNet-Modelle mit verschiedener Auflösung der Eingabebilder. Die Genauigkeit ist als COCO mittlere durchschnittliche Genauigkeit (mAP) angegeben.

Nr.	**Modell**	**Auflösung [Pixel]**	**Schritte**	**Datensatz**	**Genauigkeit [mAP]**
1	SSD MobileNet V2	224 × 224	100 k	Mod 1	93,3
2	SSD MobileNet V2	224 × 224	150 k	Mod 2	80,8
3	SSD MobileNet V2	320 × 320	80 k	Mod 1	96,7
4	SSD MobileNet V2	320 × 320	150 k	Mod 2	88,5
5	SSD Lite MobileNet V2	224 × 224	150 k	Mod 1	99,4
6	SSD Lite MobileNet V2	224 × 224	150 k	Mod 2	83,3
7	SSD Lite MobileNet V2	320 × 320	50 k	Mod 1	99,8
8	SSD Lite MobileNet V2	320 × 320	150 k	Mod 2	93,6

Wie man erkennen kann, können die *SSD Lite MobileNet V2* Modelle mit einer Klasse *PiCar* als Ausgabe schnell übertrainieren (Nr. 5 und 7 in Tab. 6.4). Ebenfalls erkennt man, dass eine höhere Auflösung die Objekte schneller erlernt und somit schnell übertainieren kann. Aus diesem Grund werden weniger Trainingsschritte bei höherer Auflösung durchgeführt (Vergleich zwischen Nr. 1 und 3 in Tab. 6.4). Die *SSD Lite MobileNet V2* Modelle lernen ebenfalls schneller gegenüber den *SSD MobileNet V2* Modellen. Somit wird an dieser Stelle mit weniger Trainingsschritten trainiert (Vergleich zwischen Nr. 3 und 7 in Tab. 6.4). Zusätzlich ist zu erwähnen, dass der Datensatz 4 (Mod 1) nur ein

annotiertes Objekt mit 111 verschiedenen Daten enthält. Das bedeutet: Es werden bei diesen Modellen weniger Trainingsschritte als bei den Modellen mit dem Datensatz 5 (Mod 2) durchgeführt (Vergleich Nr. 1 und 2 oder 7 und 8 in Tab. 6.4). Als nächstes wird die Auswertung der Testdaten der schlechtesten *SSD MobileNet V2 224 × 224* Modelle betrachtet (Abb. 6.13).

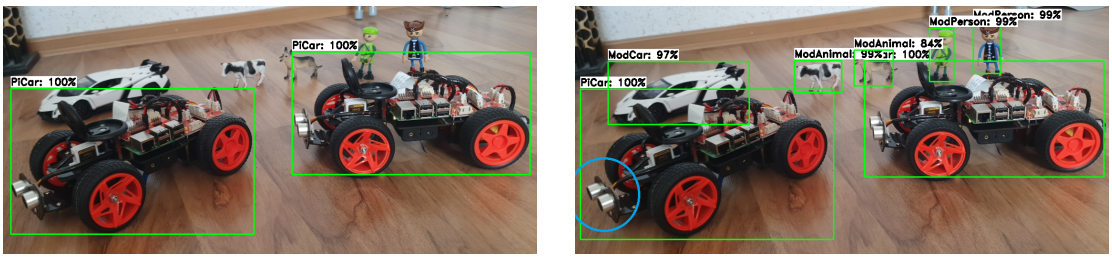


(a) Datensatz: Mod 1. Klasse: *PiCar*.

(b) Datensatz: Mod 2. Klasse: *PiCar, ModCar, ModAnimal, ModPerson*.

Abbildung 6.13: Objekterkennung mit dem *SSD MobileNet V2 224 × 224* Modell (Nr. 1 und 2 in Tab. 6.4).

Die Erkennung der Objekte ist durch grüne Rechtecke dargestellt. Diese Modelle erreichen eine Genauigkeit von 93, 3 % auf dem Datensatz 4 (Mod 1) und eine Genauigkeit von 80, 8 % auf dem Datensatz 5 (Mod 2). In der vorherigen Abbildung 6.13a ist zu erkennen, dass die Erkennung des Objekts mit einer Klasse *PiCar* ziemlich genau funktioniert. Die Klassifikation ist eindeutig und es stimmt die Position und Größe des Objekts überein. Ebenfalls liefert das *SSD MobileNet V2 224 × 224* Modell eine exakte Objekterkennung mit vier Klassen *PiCar*, *ModCar*, *ModAnimal* und *ModPerson* (Abb. 6.13b). Es stimmen die Klassifizierung und die Position der erkannten Objekte ziemlich genau überein. Es gibt nur minimale Abweichungen bei der Größe des Objekts. Das heißt, die vorhergesagte Objektgrenze überschneiden sich nicht immer mit der tatsächlichen Objektgrenze. Dies macht sich bei der prozentualen Genauigkeit (mAP) bemerkbar.



(a) Datensatz: Mod 1. Klasse: *PiCar*.

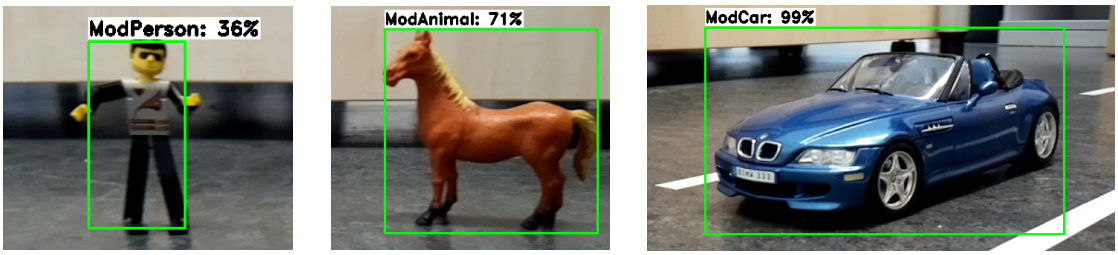
(b) Datensatz: Mod 2. Klasse: *PiCar, ModCar, ModAnimal, ModPerson*.

Abbildung 6.14: Objekterkennung mit dem *SSD Lite MobileNet V2 224 × 224* Modell (Nr. 5 und 6 in Tab. 6.4).

Wie die vorherige Abbildung 6.14 zeigt, funktioniert die Objekterkennung ebenfalls mit

dem *SSD Lite MobileNet V2 224 × 224* Modell sowohl mit einer Klasse (Abb. 6.14a) als auch mit vier Klassen *PiCar*, *ModCar*, *ModAnimal* und *ModPerson* (Abb. 6.14b). Die erkannte Klassifikation, die Position und Größe des Objekts stimmt ziemlich exakt überein. Bei einem Vergleich dieser beiden Modelle *SSD MobileNet V2 224 × 224* und *SSD Lite MobileNet V2 224 × 224* erkennt man, dass das *SSD Lite MobileNet V2 224 × 224* Modell etwas genauer ist. Diesen Vergleich verdeutlicht die hellblaue Markierung in Abbildung 6.13b und 6.14b. Dies macht sich ebenfalls bei der prozentualen Genauigkeit der Modelle bemerkbar (Vergleich zwischen Nr. 2 und 6 in Tab. 6.4). An dieser Stelle kann man sagen, dass die beiden Modelle der Objekterkennung für den Zweck dieser Forschungsarbeit völlig ausreichend sind.

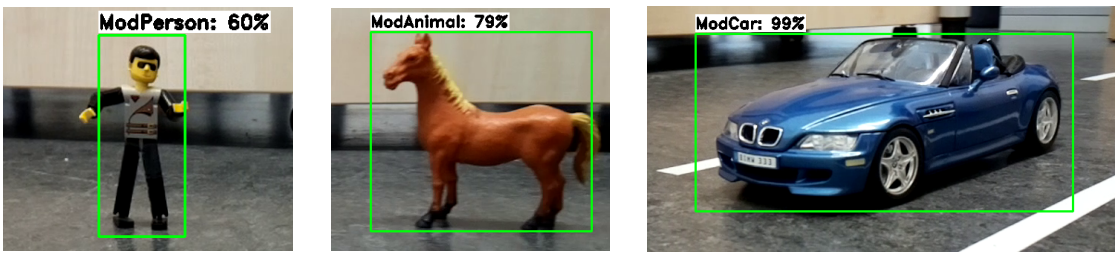
Auch die nie gesehenen Objekte konnten mit den in diesem Abschnitt trainierten Modellen erkannt werden. Zum Beispiel werden eine Lego-Person, ein Modellpferd und ein blauer BMW von dem *SSD MobileNet V2 224 × 224* Modell korrekt erkannt und klassifiziert. Dies ist in Abbildung 6.15 dargestellt.



(a) Modellperson. (b) Modellpferd. (c) Modellauto

Abbildung 6.15: Objekterkennung mit dem *SSD MobileNet V2 224 × 224* Modell (Nr. 2 in Tab. 6.4) bei nie gesehenen Objekten.

Das *SSD Lite MobileNet V2 224 × 224* Modell zeigt ebenfalls ein gutes Ergebnis bei Objekten, die während des Trainings nie gesehen wurden. Dies veranschaulicht die nachfolgende Abbildung 6.16.



(a) Modellperson. (b) Modellpferd. (c) Modellauto.

Abbildung 6.16: Objekterkennung mit dem *SSD Lite MobileNet V2 224 × 224* Modell (Nr. 6 in Tab. 6.4) bei nie gesehenen Objekten.

Während des Trainings haben die trainierten ConvNet-Modelle bereits Figuren von Modellpersonen, Modelltieren und Modellautos gesehen. Diese Modelle haben die ähnlichen

Formen der Objekte erlernt und nicht nur die Eingabebilder aus den Trainingsdatensätzen 4 (Mod 1) und 5 (Mod 2) auswendig gelernt. Die Klassifikation dieser nie gesehenen Objekte stimmt exakt überein. Natürlich ist diese Objekterkennung auf den nie gesehenen Objekten verbesserungswürdig, da die Größe der Objekte nicht immer übereinstimmt. Außerdem wird jedes andere Objekt mit einer unbekannten Form (Kontur) als die Klasse *PiCar* klassifiziert. Die Modelle haben während des Trainings noch nie ein Objekt mit einer ähnlichen Kontur gesehen. Daher kann das Objekt nicht eindeutig klassifiziert werden. Infolgedessen wird diesem unbekannten Objekt die erste Ausgabeklasse zugewiesen. In diesem Fall ist das die Klasse *PiCar*. Dieses Problem kann durch eine Vergrößerung des Datensatzes gelöst werden. Es werden mehr Trainingsdaten mit dem *PiCar* und vielen verschiedenen Objekten (unterschiedliche Formen) benötigt. So kann der Unterschied zu anderen Formen erlernt werden. Um auf die gestellte Forschungsfrage am Anfang des Abschnitts eingehen zu können, müssen einige Laufzeitmessungen dieser Modelle durchgeführt werden. Somit kann ein Gleichgewicht zwischen der Genauigkeit und der Geschwindigkeit dieser Modelle gefunden werden. Die Laufzeitmessungen der vorgestellten Modelle werden in dem Abschnitt 6.5.4 durchgeführt.

6.5 Experimente

Die folgenden Experimente wurden durchgeführt, um die Genauigkeit und die Geschwindigkeit der verschiedenen vorgestellten TensorFlow Modelle zu vergleichen. Bei den nachfolgenden Experimenten wurden die Auflösungen der einzelnen Modelle aus dem TensorFlow 2 Modellzoo übernommen. Somit wurde die Auflösung 512×512 Pixel für das *EfficientDet D0* ConvNet-Modell und 320×320 Pixel für das *SSD MobileNet V2* ConvNet-Modell beibehalten. Zusätzlich wurde die Auflösung der *SSD MobileNet V2* ConvNet-Modelle auf 224×224 Pixel verkleinert, da das *SSD MobileNet V2* ConvNet-Modell bereits gute Geschwindigkeit bei der Auswertung erzielt hat. Die Auflösung wird im Format Breite $\times$ Höhe angegeben. Die Genauigkeit wird als COCO mittlere durchschnittliche Genauigkeit (mAP) angegeben. Die Laufzeitmessungen der Modelle werden nur auf den realen Daten aus dem Modellbaubereich durchgeführt. Dadurch können bereits erste Vergleiche der Laufzeiten festgestellt werden, um somit ein geeignetes Objekterkennungssystem für Hardware mit beschränkten Ressourcen zu finden. Zusätzlich wird bei diesen Experimenten der bereits angesprochene Ansatz der Simulation-zu-Realität Übertragung bei der Objekterkennung durchgeführt. Alle Experimente werden auf der gleichen Hardware durchgeführt. Dies gibt die Möglichkeit, die Ergebnisse im Nachhinein zu vergleichen und das optimale Objekterkennungssystem für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte zu finden. Die Testeingabebilder für die jeweiligen Ansätze sind ebenfalls gleich gehalten. Für Hardware mit beschränkten Ressourcen wurden zwei Einplatinencomputer, ein Raspberry Pi 3 B und ein Raspberry Pi 4 B, verwendet. Ebenfalls zeigen einige Vorexperimente, dass die Trainingsschritte keine Auswirkung auf die Laufzeit haben. Die Laufzeit der Erkennung wird als Bilder pro Sekunde (FPS) angegeben. Diese Messungen der Laufzeit beinhalten nur die Zeit für die Objekterkennung und schließen das Laden und das Verarbeiten der Eingabebilder

aus. Die bereits in Abschnitt 2.6 vorgestellte Hardware für die Experimente sieht wie folgt aus:

- Training der ConvNet-Modelle auf Google Colab mit Intel Xeon 2.30 GHz Prozessor (CPU), 26 GB Arbeitsspeicher (RAM), NVIDIA Tesla P100-PCIE-16GB Grafikkarte (GPU).
- Laufzeitmessungen auf Dell G3 15 3590 Notebook mit Intel i7-9750H Prozessor (CPU), 16 GB Arbeitsspeicher (RAM), 256 GB SSD Festplatte und eine NVIDIA GeForce GTX 1660 Ti Grafikkarte (GPU).
- Laufzeitmessungen auf Raspberry Pi 3 B mit ARM Cortex-A53 1.2 GHz Prozessor (CPU), 1 GB Arbeitsspeicher (RAM), USB 2.0, 8 GB SD Speicherkarte mit Raspbian als Hardware mit beschränkten Ressourcen.
- Laufzeitmessungen auf Raspberry Pi 4 B mit ARM Cortex-A72 1.5 GHz Prozessor (CPU), 8 GB Arbeitsspeicher (RAM), USB 3.0, 16 GB SD Speicherkarte mit Raspbian als Hardware mit beschränkten Ressourcen.
- Laufzeitmessungen auf Intel Neural Compute Stick 2 (NCS2) mit Intel Movidius Myriad X 700 MHz Bildverarbeitungseinheit (VPU) und 4 GB Arbeitsspeicher (SDRAM) als Hardwareerweiterung für künstliche neuronale Netzwerke.

6.5.1 Sim-to-Real Übertragung

Beim Training der verschiedenen bereits vorgestellten Modelle mit den Daten aus der Simulation und den Daten aus dem Modellbaubereich stellte sich die folgende Forschungsfrage: *Wenn die Objekte in der Simulation den Modellobjekten aus der realen Welt ähneln, welches der vorgestellten ConvNet-Modelle kann die realen Modellobjekte am besten erkennen?*

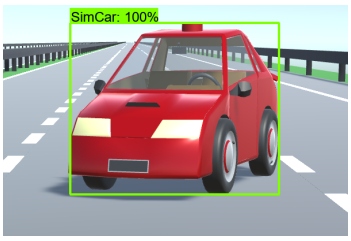
Um diese Frage zu beantworten, wurden sechs verschiedene Modelle trainiert, um einen Ansatz des Transferlernens durchzuführen. Durch die Beschaffung der Daten und automatische Annotation der Daten in der Simulation können viele verschiedene annotierte Datensätze mit vielen unterschiedlichen Objekten erstellt werden. Außerdem gibt es bereits viele modellierte Objekte in dem Unity Asset Store oder in dem Bereich der Videospiele. Die Modellierung von eigenen Objekten ist also nicht zwingend erforderlich. Für diese Sim-to-Real Übertragen wurden die beiden bereits vorgestellten *EfficientDet D0 512 × 512* und *SSD MobileNet V2 320 × 320* Modelle verwendet. Die Tabelle 6.5 zeigt diese Modelle inklusive der Trainingsschritte und der Genauigkeiten trainiert mit verschiedenen Datensätzen an. Das Training der Modelle wurde beendet, sobald sich die Verlustfunktion nicht mehr verändert hat. Diese Modelle wurden jeweils mit dem Datensatz 1 (Sim 1) mit der Klasse *SimCar* und dem Datensatz 3 (Sim 3) mit der Klasse *SimPig* aus der Simulation trainiert. Diese ConvNet-Modelle werden hinterher auf den Daten aus dem Modellbaubereich ausgewertet, um eine Simulation-zu-Realität Übertragung zu überprüfen und ein dafür geeignetes ConvNet-Modell zu erforschen. Alle

Versuche wurden mit den gleichen Trainingsparametern der Modelle durchgeführt, um die Ergebnisse anschließend zu vergleichen.

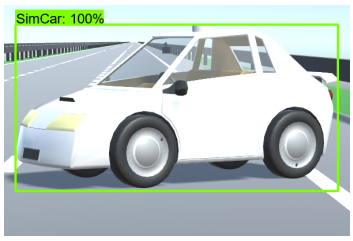
Tabelle 6.5: Überblick der trainierten TensorFlow Sim-to-Real Modelle, trainiert auf den Daten aus der Simulation. Die Genauigkeit ist als COCO mittlere durchschnittliche Genauigkeit (mAP) angegeben.

Exp. Nr.	**Modell**	**Auflösung [Pixel]**	**Schritte**	**Datensatz**	**Genauigkeit [mAP]**
1	EfficientDet D0	512 × 512	150 k	Sim 1	95,8
2	SSD MobileNet V2	320 × 320	150 k	Sim 1	93,4
3	EfficientDet D0	512 × 512	150 k	Sim 3	100,0
4	SSD MobileNet V2	320 × 320	150 k	Sim 3	99,6
5	EfficientDet D0	512 × 512	50 k	Sim 3	99,8
6	SSD MobileNet V2	320 × 320	50 k	Sim 3	99,6

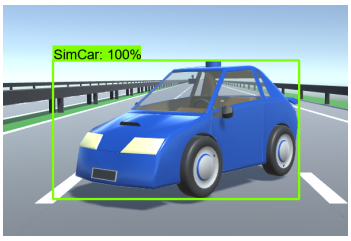
Wie in Abbildungen 6.17a, 6.17b und 6.17c zu sehen ist, erreicht das *EfficientDet D0 512 × 512* Modell (Exp. Nr. 1 in Tab. 6.5) eine sehr gute Objekterkennung auf dem Datensatz 1 (Sim 1). Das grüne Rechteck zeigt die Erkennung des Objekts durch das ConvNet-Modell. Die Position und Größe des simulierten Autos stimmt ebenfalls überein. Ebenfalls zeigen die Abbildungen 6.17d, 6.17e und 6.17f, dass die Objekte aus dem Modellbaubereich erfolgreich mit dem auf den Simulationsdaten trainiertem *EfficientDet D0 512 × 512* Modell erkannt werden können.



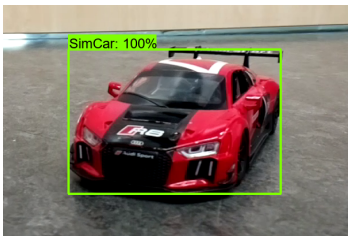
(a) Rotes Fahrzeug (beim Training gesehen).



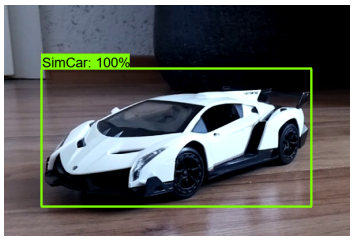
(b) Weißes Fahrzeug (beim Training gesehen).



(c) Blaues Fahrzeug (beim Training gesehen).



(d) Rotes Fahrzeug (beim Training nie gesehen).



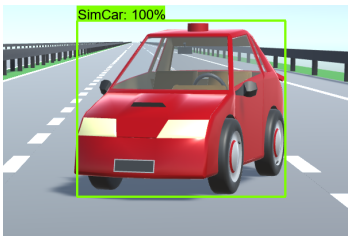
(e) Weißes Fahrzeug (beim Training nie gesehen).

(f) Blaues Fahrzeug (beim Training nie gesehen).

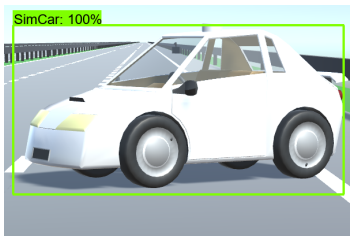
Abbildung 6.17: Sim-zu-Real Übertragung mit dem *EfficientDet D0 512 × 512* Modell (Exp. Nr. 1 in Tab. 6.5) trainiert mit dem Datensatz 1 (Sim 1).

Die erkannte Position und Größe der Objekte stimmt exakt überein. Die Eingabebilder für die Evaluation der Modellautos aus dem Modellbaubereich wurden von diesem ConvNet-Modell zuvor noch nie gesehen. Zusätzlich kann die Beschriftung der erkannten Klassen bei der Objekterkennung in dem Modellbaubereich angepasst werden. Somit ist das *EfficientDet D0 512 × 512* Modell für eine Sim-to-Real Übertragung bei der individuellen Objekterkennung geeignet.

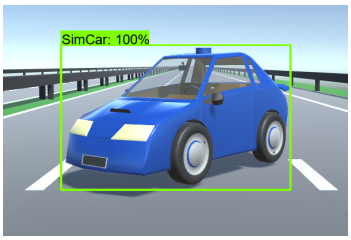
Andererseits wie die Abbildungen 6.18a, 6.18b und 6.18c zeigen, erreicht das *SSD MobileNet V2 320 × 320* Modell (Exp. Nr. 2 in Tab. 6.5) ebenfalls eine genaue Erkennung der Objekte auf den Daten aus der Simulation. Die Objekterkennung auf den Daten aus dem Modellbaubereich funktioniert jedoch nicht (Abb. 6.18d, 6.18e und 6.18f). Das *SSD MobileNet V2 320 × 320* Modell ist also nur für die Erkennung von bereits während des Trainings gesehenen Objekten geeignet. Nach diesem Experiment ist dieses Modell für die Sim-to-Real Übertragung nicht geeignet. Dies kann mit der Netzwerkarchitektur des Modells und die Eingabeauflösung der Eingabebilder des Modells zusammenhängen.



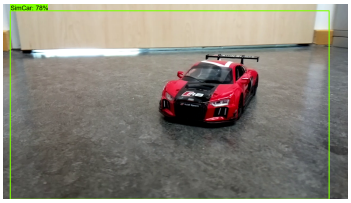
(a) Rotes Fahrzeug (beim Training gesehen).



(b) Weißes Fahrzeug (beim Training gesehen).

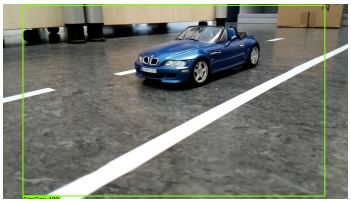


(c) Blaues Fahrzeug (beim Training gesehen).



(d) Rotes Fahrzeug (beim Training nie gesehen).

(e) Weißes Fahrzeug (beim Training nie gesehen).

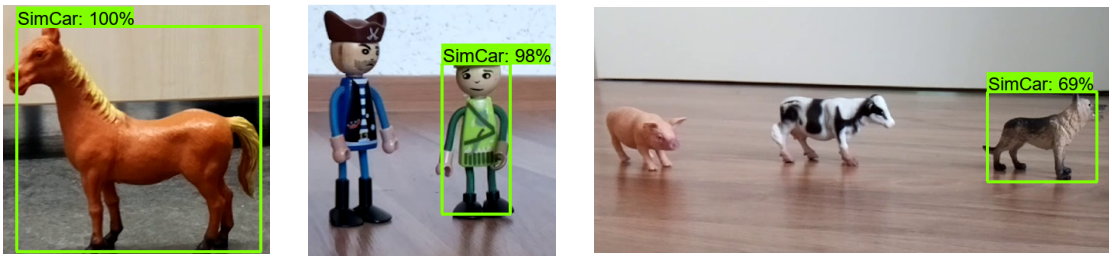


(f) Blaues Fahrzeug (beim Training nie gesehen).

Abbildung 6.18: Sim-zu-Real Übertragung mit dem *SSD MobileNet V2 320 × 320* Modell (Exp. Nr. 2 in Tab. 6.5) trainiert mit dem Datensatz 1 (Sim 1).

Wie man bereits sehen konnte, erkennt das *EfficientDet D0 512 × 512* Modell die Fahrzeuge aus dem Modellbaubereich sehr genau. Umgekehrt werden auch einige noch nie gesehenen Objekte (Tiere und Personen) teilweise erkannt und als *SimCar* klassifiziert. Dieses veranschaulicht die nachfolgende Abbildung 6.19. Dabei erkennt das Modell verschiedene Formen, die beim Training nie gesehen wurden und ordnet diesen unbekannten Objekten die erste Ausgabeklasse zu. In diesem Fall ist das die Klasse *SimCar*. Das Modell hat also nur die Form des Objekts *SimCar* erlernt und hatte wenige anderen Objekte zum Vergleich. Um dieses Problem zu umgehen, benötigt das Modell mehr Trainingsdaten mit verschiedenen Objekten. Durch die unterschiedliche Form und Kontur der Objekte

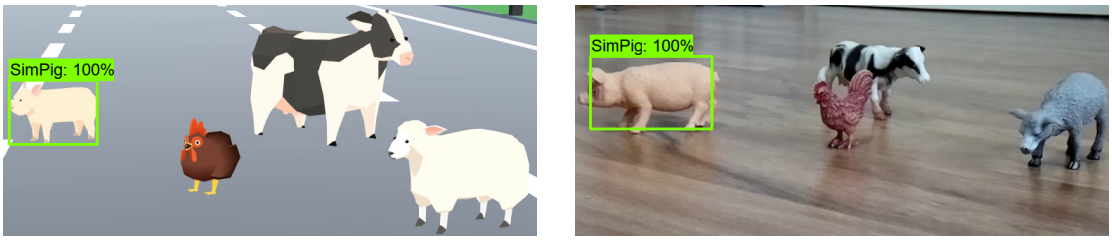
kann der Unterschied zum eigentlichen zu erkennenden Objekt erlernt werden.



(a) Modellpferd. (b) Modellperson. (c) Modellhund.

Abbildung 6.19: Sim-zu-Real Übertragung mit dem *EfficientDet D0 512 × 512* Modell (Exp. Nr. 1 in Tab. 6.5) bei nie gesehenen Objekten mit anderen Formen.

Die Sim-to-Real Übertragung wurde nicht nur mit den simulierten Autos durchgeführt. Zusätzlich wurde ein zweites Objekt hinzugezogen, um die Ergebnisse nicht dem Zufall zu überlassen. Somit wurden ähnliche Experimente mit einem simulierten Tier (Schwein) durchgeführt. Wie die Abbildungen 6.20a und 6.20b zeigen, bestätigt sich das Ergebnis mit dem *EfficientDet D0 512 × 512* Modell. Dieses *EfficientDet D0 512 × 512* Modell (Exp. Nr. 3 in Tab. 6.5) kann ebenfalls erfolgreich die Objekte aus der Simulation erkennen. Das Modellschwein aus dem Modellbaubereich kann ebenfalls erfolgreich erkannt und klassifiziert werden. Wobei dieses Modell nur die simulierten Objekte aus der Simulation gesehen hat. Allerdings war die erkannte Größe des Objekts bei dem Modellauto etwas genauer. Zusätzlich wurde dieses Modell absichtlich auf dem Datensatz mit dem Modelltier übertrainiert. Dieses ConvNet-Modell erreicht eine Genauigkeit (mAP) von 100 % auf dem Datensatz 3. Das nicht übertrainierte *EfficientDet D0 512 × 512* Modell (Exp. Nr. 5 in Tab. 6.5) mit einer Genauigkeit von 96,8 % erzielt ähnliche Ergebnisse.



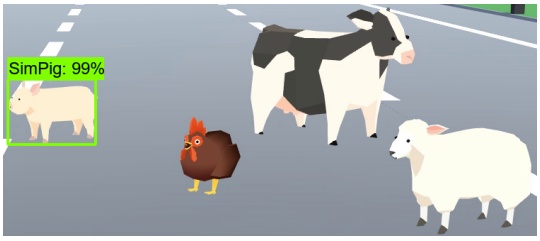
(a) Erfolgreich erkanntes Objekt (bereits gesehenen Objekte) des simulierten Tiers aus dem Datensatz 3 (*SimPig*).

(b) Erfolgreich erkanntes Objekt des Modelltiers aus dem Modellbaubereich (nie gesehenen Objekte).

Abbildung 6.20: Sim-zu-Real Übertragung mit dem *EfficientDet D0 512 × 512* Modell (Exp. Nr. 3 in Tab. 6.5) trainiert mit dem Datensatz 3 (Sim 3).

Mit dem *SSD MobileNet V2 320 × 320* Modell (Exp. Nr. 4 in Tab. 6.5), ausgewertet auf dem Modelltier, bekommt man ähnliche Ergebnisse wie zuvor mit dem Modellauto aus dem Modellbaubereich. Auf den Eingabebildern aus der Simulation kann das simulierte

Tier (Schwein) erfolgreich erkannt werden. Auf den echten Eingabebildern aus dem Modellbaubereich kann das Modellschwein mit diesem Modell nicht erkannt werden. Dieses Ergebnis veranschaulichen die nachfolgenden Abbildungen 6.21a und 6.21b.



(a) Erfolgreich erkanntes Objekt (bereits gesehenen Objekte) des simulierten Tiers aus dem Datensatz 3 (*SimPig*).

(b) Nicht erkanntes Objekt des Modelltiers aus dem Modellbaubereich (nie gesehenen Objekte).

Abbildung 6.21: Sim-zu-Real Übertragung mit dem *SSD MobileNet V2 320 × 320* Modell (Exp. Nr. 4 in Tab. 6.5) trainiert mit dem Datensatz 3 (Sim 3).

Wie die verschiedenen Experimente zeigen, ist diese Sim-to-Real Übertragung verbesserungswürdig. Diese Ungenauigkeiten könnten wahrscheinlich mit mehreren Trainingsdaten und verschiedenen Modellobjekten in der Simulation ausgebessert werden. Denn mehr Trainingsdaten verbessern normalerweise das ConvNet-Modell. Im Großen und Ganzen lässt sich aus den vorgestellten Experimenten ableiten: Die Sim-zu-Real Übertragung kann bei der Objekterkennung mit dem *EfficientDet D0 512 × 512* Modell erfolgreich durchgeführt werden. Das *SSD MobileNet V2 320 × 320* Modell ist für eine Sim-to-Real Übertragung nicht geeignet. Die Tabelle 6.6 stellt den vorgestellten Erkennungsvergleich der Modelle tabellarisch dar.

Tabelle 6.6: Erkennungsvergleich der trainierten TensorFlow Sim-to-Real Modelle auf Objekten aus der Simulation und Modellbaubereich. Die Modelle wurden mit den Daten aus der Simulation trainiert und mit den Daten aus dem Modellbaubereich ausgewertet.

Exp. Nr.	**Modell**	**Auflösung [Pixel]**	**Datensatz**	**Erkennt Sim**	**Erkennt Mod**
1	EfficientDet D0	512 × 512	Sim 1	Ja	Ja
2	SSD MobileNet V2	320 × 320	Sim 1	Ja	Nein
3	EfficientDet D0	512 × 512	Sim 3	Ja	Ja
4	SSD MobileNet V2	320 × 320	Sim 3	Ja	Nein
5	EfficientDet D0	512 × 512	Sim 3	Ja	Ja
6	SSD MobileNet V2	320 × 320	Sim 3	Ja	Nein

Diese ConvNet-Modelle wurden jeweils mit den Testdaten aus der Simulation und dem Modellbaubereich ausgewertet. Bei weniger durchgeführten Trainingsschritten der Modelle konnte ebenfalls die Sim-to-Real Übertragung durchgeführt werden und die Ergebnisse

bestätigt werden (Exp. Nr. 5 und 6 in Tab. 6.6).

6.5.2 Laufzeit auf Grafikkarte

Wie in Abschnitt 6.4.1 zu sehen ist, wurden bereits sehr gute Ergebnisse bei der Objekterkennung mit den *EfficientDet D0* und *SSD MobileNet V2* Modellen erzielt. Bei diesen Experimenten wird als erstes eine Messung der Laufzeit dieser Modelle auf einer leistungsstarken Hardware mit einer Grafikkarte durchgeführt. Die Hardware des dafür verwendeten Dell G3 15 3590 Notebooks wird in Abschnitt 2.6.1 genauer vorgestellt. Diese Laufzeitmessungen auf den realen Daten aus dem Modellbaubereich stellt die Tabelle 6.7 dar. Die erste Spalte enthält die Identifikationsnummer (ID) des Experiments (Exp. Nr.). Wie in Tabelle 6.7 Exp. Nr. 5 zu sehen ist, wird das beste Ergebnis mit dem *SSD MobileNet V2 224 × 224* Modell und einer Ausgabeklasse *PiCar* erreicht. Dieses Modell erreicht 40, 1 Bilder pro Sekunde (FPS). Diese Messungen beinhalten nicht das Laden und das Verarbeiten der Eingabebilder. Mit vier Klassen *PiCar*, *ModCar*, *ModPerson* und *ModAnimal* erreicht das gleiche Modell bis zu 39, 2 FPS (Exp. Nr. 6 in Tab. 6.7). Wie man ebenfalls erkennen kann, erreicht das *EfficientDet D0 512 × 512* Modell ca. 14 FPS auf einer leistungsstarken Hardware mit einer Grafikkarte (Exp. Nr. 1 und 2 in Tab. 6.7). Somit ist das *EfficientDet D0 512 × 512* ConvNet-Modell für eine Echtzeitobjekterkennung auf dieser Hardware nicht geeignet.

Tabelle 6.7: Übersicht der Laufzeiten der trainierten TensorFlow 2 Modelle auf leistungsstarken Hardware mit Grafikkarte. Die Laufzeit ist mit Bilder pro Sekunde (FPS) angegeben. Die Genauigkeit ist als COCO mittlere durchschnittliche Genauigkeit (mAP) angegeben.

Exp. Nr.	**Modell**	**Auflösung [Pixel]**	**Datensatz**	**Genauigkeit [mAP]**	**Laufzeit [FPS]**
1	EfficientDet D0	512 × 512	Mod 1	91,7	14,6
2	EfficientDet D0	512 × 512	Mod 2	76,4	14,2
3	SSD MobileNet V2	320 × 320	Mod 1	92,1	39,8
4	SSD MobileNet V2	320 × 320	Mod 2	69,3	38,1
5	SSD MobileNet V2	224 × 224	Mod 1	90,9	40,1
6	SSD MobileNet V2	224 × 224	Mod 2	60,6	39,2

6.5.3 Laufzeit auf beschränkten Ressourcen ohne Hardwareerweiterung

In folgenden Experimenten werden die Laufzeitmessungen der vorgestellten Modelle auf einer Hardware mit beschränkten Ressourcen durchgeführt. Für diese Tests wurde ein Raspberry Pi 3 B verwendet. Um die Laufzeit der Objekterkennung auf Hardware mit beschränkten Ressourcen zu erhöhen, wurden für diese Experimente ebenfalls zwei quantisierte *SSD MobileNet V2 224 × 224* TensorFlow 1 Modelle erstellt. Die Laufzeitmessungen dieser Modelle wurden mit der *tflite runtime-2.5.0-cp37*-Bibliothek durchgeführt [105].

Dafür wurden diese Modelle von TensorFlow Core nach TensorFlow Lite umgewandelt, um bessere Geschwindigkeiten zu erreichen. Die Quantisierung konvertiert die Gewichte des Modells von dem Datentyp *float* zu dem Datentyp *uint8*. Dabei werden die Gewichte nach dem Training des Modells von Fließkommazahlen nach Ganzzahlen umgewandelt. Dadurch können die Gewichte des faltenden neuronalen Netzwerks schneller berechnet und die Modellgröße reduziert werden, ohne dass die Modellgenauigkeit beeinträchtigt wird [106]. Die Laufzeitmessungen der normalen (Nicht-Lite) TensorFlow 2 Modelle wurden mit der *TensorFlow 2.4.0-rc2*-Bibliothek auf dem Raspberry Pi 3 B durchgeführt [104]. Die Tabelle 6.8 zeigt die Laufzeitmessungen auf einen Blick. Die Bezeichnungserweiterung *Q* Steht für ein quantisiertes Modell.

Tabelle 6.8: Übersicht der Laufzeiten der trainierten TensorFlow 2 Modelle auf Hardware mit beschränkten Ressourcen. Die Bezeichnung *Q* steht für ein quantisiertes TensorFlow 1 Modell. Die Laufzeit ist mit Bilder pro Sekunde (FPS) angegeben. Die Genauigkeit ist als COCO mittlere durchschnittliche Genauigkeit (mAP) angegeben.

Exp. Nr.	**Modell**	**Auflösung [Pixel]**	**Datensatz**	**Genauigkeit [mAP]**	**Laufzeit [FPS]**
1	EfficientDet D0	512 × 512	Mod 1	91,7	-
2	EfficientDet D0	512 × 512	Mod 2	76,4	-
3	SSD MobileNet V2	320 × 320	Mod 1	92,1	0,9
4	SSD MobileNet V2	320 × 320	Mod 2	69,3	0,9
5	SSD MobileNet V2	224 × 224	Mod 1	90,9	1,4
6	SSD MobileNet V2	224 × 224	Mod 2	60,6	1,4
7	SSD MobileNet V2 Q	224 × 224	Mod 1	90,9	3,2
8	SSD MobileNet V2 Q	224 × 224	Mod 2	60,6	2,8

Bei den *EfficientDet D0 512 × 512* Modellen wurde die Messung nach einiger Zeit automatisch vom Raspberry Pi 3 B beendet. Es konnten keine Laufzeitmessungen für diese Modelle auf dem Raspberry Pi 3 B durchgeführt werden. Auf diesem System ist der Nicht-genügend-Arbeitsspeicher (engl. Out of Memory) Fehler aufgetreten. Somit sind die *EfficientDet D0 512 × 512* Modelle für den Einsatz auf Hardware mit beschränkten Ressourcen nicht geeignet. Wie man in Tabelle 6.8 Exp. Nr. 7 erkennen kann, ist das *SSD MobileNet V2 Q 224 × 224* Modell mit einer Ausgabeklasse *PiCar* das schnellste ConvNet-Modell. Hier schafft das Modell ca. $3,2$ Bilder pro Sekunde. Auch bei diesen Messungen ist das Laden und das Verarbeiten der Eingabebilder nicht berücksichtigt. Mit vier Ausgabeklassen *PiCar*, *ModCar*, *ModPerson* und *ModAnimal* erreicht das gleiche Modell bis zu $2,8$ FPS. TensorFlow gibt mindestens einen doppelten Geschwindigkeitsvorteil nach der Quantisierung des Modells an [106]. Dies kann beim Vergleich von Exp. Nr 5 und 7 oder Exp. Nr 6 und 8 in Tabelle 6.8 bestätigt werden.

6.5.4 Laufzeit auf beschränkten Ressourcen mit Hardwareerweiterung

In diesen Experimenten werden die im Abschnitt 6.4.2 vorgestellten Modelle für die Objekterkennung auf der Hardware mit beschränkten Ressourcen mit einer Intel NCS2 Hardwareerweiterung evaluiert. Diese Modelle wurden mit TensorFlow 1 trainiert und anschließend mit der *OpenVINO*-Bibliothek in das *OpenVINO*-Format konvertiert, um diese Modelle auf der Hardwareerweiterung Intel NCS2 lauffähig zu machen. Dadurch kann ein ConvNet-Modell mit einem optimalen Gleichgewicht zwischen der Genauigkeit und der Geschwindigkeit für die Objekterkennung gefunden werden. Die Tabelle 6.9 zeigt einen Überblick über die Genauigkeit und die Laufzeit der trainierten *SSD MobileNet V2* und *SSD Lite MobileNet V2* Modelle auf dem Raspberry Pi 3 B mit Intel NCS2 und Raspberry Pi 4 B mit Intel NCS2.

Tabelle 6.9: Laufzeitübersicht der trainierten TensorFlow 1 Modelle auf Raspberry Pi (RPI) 3 B mit Intel NCS2 und Raspberry Pi (RPI) 4 B mit Intel NCS2. Die erste Spalte enthält die Nummer (ID) des Experiments (Exp. No.). Die Genauigkeit ist als COCO mittlere durchschnittliche Genauigkeit (mAP) angegeben.

Exp. Nr.	**Modell**	**Auflösung [Pixel]**	**Daten-satz**	**Genauig-keit [mAP]**	**Laufzeit [FPS]**	
					RPI 3 B	**RPI 4 B**
1	SSD MobileNet V2	224 × 224	Mod 1	93,3	17,4	31,9
2	SSD MobileNet V2	224 × 224	Mod 2	80,8	17,2	31,7
3	SSD MobileNet V2	320 × 320	Mod 1	96,7	9,8	18,0
4	SSD MobileNet V2	320 × 320	Mod 2	88,5	9,7	17,9
5	SSD Lite MobileNet V2	224 × 224	Mod 1	99,4	16,3	29,3
6	SSD Lite MobileNet V2	224 × 224	Mod 2	83,3	16,3	29,1
7	SSD Lite MobileNet V2	320 × 320	Mod 1	99,8	9,4	16,7
8	SSD Lite MobileNet V2	320 × 320	Mod 2	93,6	9,3	16,7

Die Tabelle 6.9 zeigt, dass der Raspberry Pi 4 B mit der Intel NCS2 Hardwareerweiterung eine Echtzeitauswertung bei der Objekterkennung von ca. 32 FPS mit den *SSD MobileNet V2 224 × 224* Modellen (Exp. Nr 1 und 2 in Tab. 6.9) und ca. 29 FPS mit den *SSD Lite MobileNet V2 224 × 224* Modellen (Exp. Nr 5 und 6 in Tab. 6.9) erreicht. Wie man in der Tabelle 6.9 ebenfalls erkennen kann, ist die Laufzeit auf dem Raspberry Pi 4 B mit Intel NCS2 fast doppelt so hoch wie auf dem Raspberry Pi 3 B mit Intel NCS2 (Vergleich zwischen RPI 3 B und RPI 4 B in Exp. Nr. 1 in Tab. 6.9). Die Vermutung an dieser Stelle: Der Raspberry Pi 3 B hat nur eine USB 2.0 Schnittstelle. Der Raspberry Pi 4 B hingegen verfügt über eine USB 3.0 Schnittstelle. Die Intel NCS2 Hardwareerweiterung verfügt ebenfalls über eine USB 3.0 Schnittstelle. Durch die USB 3.0 Schnittstelle können höhere Übertragungsgeschwindigkeiten erzielt werden. Dies könnte der Engpass in Bezug auf die Laufzeit auf dem Raspberry Pi 3 B sein. Um diese Vermutung zu überprüfen, wurde

zusätzlich die Laufzeit auf dem Raspberry Pi 4 B und der Intel NCS2 Hardwareerweiterung, verbunden über die USB 2.0 Schnittstelle, durchgeführt. Für diesen Test wurde das *SSD MobileNet V2 224 × 224* Modell verwendet. An dieser Stelle ist aber das verwendete Modell nicht relevant. Die Tabelle 6.10 zeigt, dass die Vermutung richtig war.

Tabelle 6.10: Laufzeitübersicht vom Raspberry Pi 4 B mit Intel NCS2 verbunden über die USB 2.0 und USB 3.0 Schnittstelle. Die erste Spalte enthält die Nummer (ID) des Experiments (Exp. No.).

Exp. Nr.	**Modell**	**Auflösung [Pixel]**	**Datensatz**	**Genauigkeit [mAP]**	**Laufzeit [FPS]**	
					USB 2.0	**USB 3.0**
1	SSD MobileNet V2	224 × 224	Mod 1	93,3	18,9	31,9

Der Engpass der Laufzeit auf dem Raspberry Pi 3 B mit der Intel NCS2 Hardwareerweiterung ergibt sich durch die USB 2.0 Schnittstelle (Vergleich zwischen Exp. Nr. 1 in Tab. 6.9 und Exp. Nr. 1 in Tab. 6.10). Die Echtzeitauswertung der Objekterkennung wird auf dem Raspberry Pi 4 B mit der Intel NCS2 Hardwareerweiterung verbunden durch die USB 2.0 Schnittstelle mit dem *SSD MobileNet V2 224 × 224* Modell nicht mehr erreicht. Die Geschwindigkeit der Objekterkennung sinkt durch diesen Engpass von 31, 9 FPS auf 18, 9 FPS.

6.5.5 Auswertung der Ergebnisse

Nach dem Durchführen der Experimente kann mit der Auswertung der Ergebnisse begonnen werden. Wie die Experimente der Sim-to-Real Übertragung aus Abschnitt 6.5.1 zeigen, kann die folgende Forschungsfrage: *Wenn die Objekte in der Simulation den Modellobjekten aus der realen Welt ähneln, welches der vorgestellten ConvNet-Modelle kann die realen Modellobjekte am besten erkennen?* positiv beantwortet werden. Es kann erfolgreich ein ConvNet-Modell für die Sim-to-Real Übertragung gefunden werden. Das *EfficientDet D0 512 × 512* Modell eignet sich für den Sim-to-Real Ansatz mit den vorgestellten Datensätzen 1 (Sim 1) und 3 (Sim 3). Dieses Modell kann mit den Daten aus der Simulation trainiert werden, um anschließend die Objekte aus der realen Umgebung (Modellbaubereich) zu erkennen.

Zusätzlich, nachdem die Leistungsexperimente auf Hardware mit beschränkten Ressourcen ohne Hardwareerweiterung in Abschnitt 6.5.3 der vorgestellten Modelle abgeschlossen waren, konnte mit der Auswertung der Laufzeiten dieser verschiedenen Modelle begonnen werden. Hierbei ist es wichtig, ein Gleichgewicht zwischen ausreichender Genauigkeit und der optimalen Laufzeit der Modelle für die Objekterkennung zu finden. Außerdem zeigen einige Experimente, dass das längere Training (mehr Trainingsschritte) keinen Einfluss auf die Laufzeitmessung hat. Die Laufzeit hängt nur von der Modellarchitektur, von der Größe der Eingabeauflösung des Netzwerks und von der Größe der Ausgabeklassen der ConvNet-Modelle ab. Auf einer leistungsstarken Hardware mit einer Grafikkarte

konnte eine Echtzeitobjekterkennung mit den *SSD MobileNet V2 224 × 224* und *SSD MobileNet V2 320 × 320* Modellen erreicht werden (Exp. Nr. 3 bis 6 in Tab. 6.7). Auf Hardware mit beschränkten Ressourcen ohne Grafikkarte und ohne eine Hardwareerweiterung, können ca. 3,2 FPS mit dem *SSD MobileNet V2 Q 224 × 224* Modell auf dem Raspberry Pi 3 B erreicht werden (Exp. Nr. 7 in Tab. 6.8). Die für die Hardware mit beschränkten Ressourcen vorgestellte Objekterkennung ist langsam und erreicht keine Echtzeitauswertung ohne den Zusatz der Hardwareerweiterung. Die Objekterkennung kann auf den IoT-Geräten aber trotzdem durchgeführt werden, sobald nicht jede Bildsequenz ausgewertet werden soll. Wenn zum Beispiel Anomalien erkannt werden, kann diese Objekterkennung vorgenommen werden. In einem autonomen Fahrzeug kann dies zum Beispiel mit dem Radarsensor geschehen. Bei intelligenten Überwachungskameras, zum Beispiel im Bereich der Land- und Forstwirtschaft, könnte es der Bewegungssensor sein.

Nachdem die Experimente und Leistungstests auf Hardware mit beschränkten Ressourcen mit einer Hardwareerweiterung in Abschnitt 6.5.4 abgeschlossen sind, kann mit der Bewertung der Laufzeiten dieser verschiedenen Modelle begonnen werden. An dieser Stelle ist es ebenso wichtig, ein Gleichgewicht zwischen ausreichender Genauigkeit und der Geschwindigkeit der Modelle zu finden. Auch hier zeigen einige Experimente, dass das längere Training der Modelle keinen Einfluss auf die Laufzeit hat. Die Laufzeit hängt ebenfalls von der Modellarchitektur, von der Größe der Eingabeauflösung des Modells und von der Größe der Ausgabeklassen der ConvNet-Modelle ab. Betrachtet man die *SSD MobileNet V2 224 × 224* Modelle, so erreichen diese eine gute Genauigkeit (mAP) von 93,3 % beziehungsweise 80,8 % und eine Geschwindigkeit von ca. 31,9 FPS beziehungsweise 31,7 FPS. Dieser Vergleich wurde der Nummer 1 und 2 der Experimente in Tabelle 6.9 entnommen. Die *SSD Lite MobileNet V2 224 × 224* Modelle sind etwas genauer gegenüber den *SSD MobileNet V2 224 × 224* Modellen. Diese sind aber auch zum Vergleich etwas langsamer. Diese erreichen eine Genauigkeit (mAP) von 99,4 % beziehungsweise 83,3 % und eine Geschwindigkeit von ca. 29,3 FPS beziehungsweise 29,1 FPS. Das zeigen die Nummern 5 und 6 der Experimente in der Tabelle 6.9. Videos mit viel Bewegung werden standardmäßig mit 30 FPS aufgenommen [55]. Um diese Bildsequenzen zu analysieren und Objekte in Echtzeit zu erkennen, eignen sich die vier mit TensorFlow 1 trainierten *SSD MobileNet V2 224 × 224* und *SSD Lite MobileNet V2 224 × 224* Modelle für eine Echtzeitanwendung auf dem Raspberry Pi 4 B mit Intel NCS2 (Exp. Nr. 1, 2, 5 und 6 in Tab. 6.9). Hier konnten maximal bis zu ca. 32 FPS erreicht werden. An dieser Stelle kann man sagen, dass eine Objekterkennung in Echtzeit auf Hardware mit beschränkten Ressourcen erfolgreich durchgeführt werden konnte. An dieser Stelle kann die zuvor gestellte Forschungsfrage: *Kann die bereits vorgestellte Objekterkennung für Hardware mit beschränkten Ressourcen von ca. 3 FPS auf eine Echtzeitobjekterkennung von ca. 30 FPS auf Hardware mit beschränkten Ressourcen durch die Intel NCS2 Hardwareerweiterung beschleunigt werden?* mit ja beantwortet werden.

Wenn man sich die *SSD MobileNet V2 320 × 320* und *SSD Lite MobileNet V2 320 × 320* ConvNet-Modelle anschaut, sind diese Modelle gegenüber den *SSD MobileNet V2 224 × 224* und *SSD Lite MobileNet V2 224 × 224* Modellen etwas genauer. Allerdings sind diese, wie erwartet, auch in Bezug auf die Laufzeit etwas schlechter (Vergleich

zwischen Exp. Nr. 6 und 8 in Tab. 6.9). Diese Modelle erreichen bis zu ca. 18 FPS auf dem Raspberry Pi 4 B mit Intel NCS2 (Exp. Nr. 3 in Tab. 6.9). Eine Echtzeitauswertung der Objekterkennung ist mit diesen ConvNet-Modellen auf der verwendeten Hardware nicht zu erreichen.

6.6 Schlussfolgerungen

Dieser Abschnitt fasst noch einmal die Punkte zusammen, die in diesem Kapitel vorgestellt wurden. Die Nutzung der entwickelten individuellen Objekterkennung auf Hardware mit beschränkten Ressourcen für IoT-Geräte mit geringem Stromverbrauch war die erste Priorität. Die Beschaffung von automatisch annotierten Trainingsdaten mit der Simulation aus Kapitel 3 wurde ebenfalls vorgestellt. Zusätzlich wurden eigene Datensätze aus dem Modellbaubereich erstellt und manuell annotiert. So konnten die verschiedenen Ergebnisse anschließend verglichen werden. Es wurden verschiedene ConvNet-Modelle sowohl mit TensorFlow 1 als auch mit TensorFlow 2 trainiert, um ein Gleichgewicht zwischen der Genauigkeit und der Laufzeit dieser Modelle zu finden. Nach den durchgeführten Experimenten ist sowohl das *SSD MobileNet V2 mit 224 × 224* Modell als auch das *SSD MobileNet V2 mit 320 × 320* für die Objekterkennung in Echtzeit auf einer leistungsstarken Hardware mit einer Grafikkarte geeignet. Dabei konnten bis zu ca. 40 FPS erreicht werden. Für eine Hardware mit beschränkten Ressourcen ohne eine Hardwareerweiterung konnten ebenfalls einige Modelle gefunden werden. Diese *SSD MobileNet V2 Q 224 × 224* Modelle schaffen allerdings keine Echtzeitauswertung auf der vorgestellten Hardware. Diese erreichen bis zu 3,2 FPS auf einem Raspberry Pi 3 B ohne Hardwareerweiterung.

Den vorgestellten Experimenten zufolge sind die mit TensorFlow 1 trainierten *SSD MobileNet V2 224 × 224* und *SSD Lite MobileNet V2 224 × 224* Modelle für eine Objekterkennung in Echtzeit auf dem Raspberry Pi 4 B mit dem Intel Neural Compute Stick 2 als Hardwareerweiterung geeignet. Hier konnten mit der verwendeten Hardware bis zu 32 FPS erreicht werden. Dies ist für eine Echtzeitanwendung völlig ausreichend. Diese Modelle erreichen eine effektive Balance zwischen der Genauigkeit und der Geschwindigkeit der Objekterkennung auf Hardware mit beschränkten Ressourcen mit Hardwareerweiterung. Schließlich wurde in dieser Arbeit erfolgreich eine Sim-to-Real Übertragung bei der Objekterkennung durchgeführt. Das heißt, das faltende neuronale Netzwerk wurde mit den Trainingsdaten aus der Simulation trainiert und mit den Testdaten aus der echten Welt (Modellbaubereich) ausgewertet. Mit dem *EfficientDet D0 512 × 512* Modell kann dieser Sim-to-Real Ansatz erfolgreich durchgeführt werden. Der Vorteil in der simulierten Umgebung zu arbeiten ist, dass viele annotierte Trainingsdaten mit verschiedenen Objekten in kurzer Zeit erstellt werden können. Es können auch mehrere Objekte in einem Bild erstellt und automatisch annotiert werden. Da die Position der Objekte bekannt ist, können diese Objekte verschoben oder ebenfalls untereinander ausgetauscht werden. Die Position und Größe der Objekte kann ebenfalls verändert werden.

Wie bereits angesprochen, für eine optische Entfernungsmessung zu einem erkannten Objekt in einem zweidimensionalen Bild müssen, zum Beispiel, die erkannte Position

und Größe der Objekte exakt mit der tatsächlichen Position und Größe der Objekte übereinstimmen. Dies ist die Voraussetzung für das Radarkontrollsystem im nächsten Kapitel 7. Diese individuelle Objekterkennung mit dem Einsatz auf autonomen Fahrzeugen beziehungsweise dem Modellbaubereich zeigt nur einen möglichen Anwendungsbereich der neuronalen Netzwerke auf der Hardware mit beschränkten Ressourcen. Zusätzlich stellt dieses Kapitel einen kostengünstigen Ansatz vor, um ein System mit Echtzeitauswertung durch Hardwareerweiterung auf stromsparenden IoT-Geräten zu entwickeln. An dieser Stelle kann man ebenfalls sagen, dass die neuronalen Netzwerke nicht überdimensioniert werden sollten. Denn je größer das neuronale Netzwerk ist, desto langsamer ist dieses bei der Auswertung. Die verwendete Netzwerkarchitektur und die Konfiguration dieser ConvNet-Modelle sollten für jeden spezifischen Anwendungszweck erforscht und angepasst werden. Das nachfolgende Kapitel 7 stellt das bereits angesprochene Radarkontrollsystem für autonome Fahrzeuge vor.

Mathematik ist Musik des Geistes, Musik ist Mathematik der Seele.

Daniil Charms
1905 – 1942

Kapitel 7

Kontrollsystem für Radarsensor auf Hardware mit beschränkten Ressourcen

7.1 Einführung

Die autonomen Fahrzeuge werden jeden Tag in der autonomen Fahrzeugindustrie weiterentwickelt. Dazu gehört auch das Verbauen verschiedener Sensoren und neuer elektronischer Bauteile. Diese Sensoren benötigt das autonome Fahrzeug, um die Umwelt zu sehen und mit ihr interagieren zu können. Dadurch ist es möglich, die autonomen Fahrzeuge, ohne menschliche Intervention, eigenständig fahren zu lassen. Doch wie verhält sich das autonome Fahrzeug, wenn diese Sensoren ausfallen oder falsche Messwerte liefern? Der Radarsensor kann die Entfernung zum vorderen Objekt durch elektromagnetische Wellen messen. Dadurch kann zum Beispiel automatisch in einem autonomen Fahrzeug der Abstand zum vorderen Fahrzeug gehalten oder der Brems- beziehungsweise Notbremsassistent entwickelt werden. Um diese Messwerte zu überprüfen, wird in diesem Kapitel ein optisches Kontrollsystem für den Radarsensor für autonome Fahrzeuge vorgestellt. Dieses Kontrollsystem ist realisiert durch die optische Entfernungsmessung in einem zweidimensionalen Bild. Ebenfalls kann zu diesem Zweck eine Stereokamera verwendet werden. Hierbei ist der Ansatz das optische Kontrollsystem mit nur einer Kamera zu realisieren und in einem zweidimensionalen Bild die Entfernungsmessungen zum vorderen Objekt auf der selben Fahrspur durchzuführen. Durch dieses Überprüfungssystem ist es möglich, die Messwerte des Radarsensors zu überprüfen. Dabei benötigt dieses System auch kein Referenzobjekt, welches benötigt wird, um die Pixel auf die echte Entfernung in der realen Welt umzurechnen. Zusätzlich werden verschiedene Funktionen für die Entfernungsmessung approximiert, die für die optische Entfernungsmessung verwendet werden. Bei diesem Ansatz ist die Entwicklung einer Echtzeitentfernungsmessung für

Hardware mit beschränkten Ressourcen im Bereich der Internet der Dinge (IoT) im Vordergrund. Das in diesem Kapitel vorgestellte Kontrollsystem für den Radarsensor (Abstandsmesssystem) wird mit simulierten Daten, mit realen Daten aus dem Modellbaubereich und mit Daten eines realen Personenkraftfahrzeugs aus der realen Umgebung evaluiert und getestet. Zu Beginn werden verwandte Arbeiten dieser Thematik diskutiert. Die Datenbeschaffung und Annotation der verschiedenen simulierten und realen Datensätze wird ebenfalls vorgestellt. Darüber hinaus werden die unterschiedlichen Ansätze zur Entfernungsmessung in Bezug auf Genauigkeit und Laufzeit verglichen, um das bessere und schnellere System zur Entfernungsmessung in einem zweidimensionalen Bild auf Hardware mit beschränkten Ressourcen für IoT-Geräte mit geringem Stromverbrauch zu finden. Zusätzlich durch die durchgeführten Experimente wird der Vergleich der Laufzeit in Abhängigkeit von unterschiedlicher IoT-Hardware vorgestellt. Zu diesem Zweck werden der Raspberry Pi 3 B und der Raspberry Pi 4 B als IoT-Geräte mit geringem Stromverbrauch verwendet [57].

Um die Entfernungsmessungen durchführen zu können, muss zusätzlich die Position des vorderen Objekts bekannt sein. Diese Objekterkennung muss natürlich in Echtzeit erfolgen, da die Distanzmessung ebenfalls in Echtzeit erfolgen muss. Für die Objekterkennung wird die bereits in Kapitel 6 vorgestellte Echtzeitobjekterkennung für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte genutzt. Ebenfalls ist die Position dieser Objekte in dem Bild sehr wichtig. Zum Beispiel, ob sich das vordere Fahrzeug auf der selben Fahrbahn befindet oder es nur an der Seite der Straße geparkt ist. Um die Fahrspur in Echtzeit zu erkennen, kann die bereits in Kapitel 4 vorgestellte Spurerkennung, der gefilterte Canny-Algorithmus, verwendet werden. Ebenfalls handelt es sich bei diesem System nicht um den Ersatz des Radarsensors, sondern lediglich um ein optisches Überprüfungssystem. Dieses entwickelte Kontrollsystem für den Radarsensor für autonome Fahrzeuge soll einen möglichen Einsatzbereich dieser Entfernungsmessung präsentieren. Diese Entfernungsmessung ist natürlich nicht nur für diesen Bereich interessant. Ein solches System kann in beliebiger IoT-Hardware, die eine Kamera enthält, verwendet werden. Zum Beispiel können Drohnen, die ein Paket ausliefern, die Entfernung zu einem beliebigen Objekt in einem zweidimensionalen Bild messen. Denkbar ist auch der Einsatz dieser optischen Entfernungsmessung in smarten Überwachungskameras, um verschiedene Entfernungen zu den Objekten zu messen. Ebenfalls kann diese optische Entfernungsmessung in Wildkameras verwendet werden, die keinen Radarsensor beziehungsweise Ultraschallsensor enthalten. Dabei können die verschiedenen Tiere erkannt und verfolgt werden. Die Idee hinter dieser optischen Entfernungsmessung ist die Verwendung von teilweise trivialen traditionellen Methoden und nicht die Verwendung von künstlichen neuronalen Netzwerken, wo immer dieses möglich ist.

7.2 Verwandte Arbeiten

Es gibt einige wissenschaftliche Arbeiten, die sich mit der Abstandsmessung befassen, zum Beispiel gibt es eine physische Abstandserkennung unter Verwendung von YOLO v3 und Transformation in die Vogelperspektive [67] oder eine Abstandsmessung zwischen

Person und Kamera auf der Grundlage des Augenabstands [88]. Darüber hinaus gibt es eine Objektabstandsmessung, die durch Stereosehen (engl. Stereo Vision) implementiert ist. Dieser Ansatz erfordert eine dreidimensionale Kamera, die aus zwei Kameras mit parallelen optischen Achsen besteht [64]. Ebenfalls gibt es wissenschaftliche Arbeiten zu Verarbeitung von Stereobildern für die Messung von Entfernung und Größe von Objekten in Echtzeit [74]. Zusätzlich gibt es ein Messverfahren welches in einem Drei-Phasen-Prozess, Objekterkennung, Segmentierung und Abstandsberechnung, aufgebaut ist. Dieser Ansatz stellt eine Abstandsberechnung mit einem Algorithmus vor, um den Fehler bei der Entfernungsmessung zu reduzieren [37]. Die Entfernungsmessungen dieser Forschungsarbeiten funktionieren durch eine Stereokamera. Darüber hinaus wurde bereits eine Methode zur Entfernungsmessung für Digitalkameras vorgestellt, die auf der Veränderung der Pixelzahl der Bilder des ladungsgekoppelten Bauelements (engl. Charge Coupled Device - CCD) basiert, indem auf zwei willkürlich festgelegte Punkte in dem Bildrahmen Bezug genommen wird. Diese Bilder werden durch den CCD-Sensor bei der Bildaufnahme mit einer Digitalkamera erzeugt. Durch die Herstellung einer Beziehung zwischen der Verschiebung der Kamerabewegung entlang der Aufnahmerichtung und der Differenz der Pixelanzahl zwischen den Referenzpunkten in den Bildern kann die Entfernung zu einem Objekt mit der vorgeschlagenen Methode berechnet werden [36]. Der Ansatz dieser Forschungsarbeit in diesem Kapitel besteht darin, eine Entfernungsmessung mit nur einer Kamera durch eine lineare Regression ohne ein Referenzobjekt zu implementieren und ein Radarkontrollsystem in Echtzeit für Hardware mit beschränkten Ressourcen zu entwickeln. Dadurch ist es in kurzer Zeit möglich, eine kostengünstige Echtzeitentfernungsmessung in einer Überwachungskamera, zum Beispiel für den Modellbaubereich, IoT-Bereich oder den Land- und Forstbetrieb zu entwickeln.

7.3 Datensätze

Bevor die Entfernungsmessung durchgeführt werden kann, werden verschiedene Datensätze für die Entfernungsmessung benötigt. Eine höhere Qualität der Daten sorgt für eine erfolgreiche Entfernungsmessung mit einer geringeren Fehlerrate. Für diesen Zweck wurden verschiedene Datensätze aus der Simulation, dem Modellbaubereich und aus der realen Welt erstellt. Für jedes Bild gibt es eine tatsächlich gemessene Entfernung (Annotation in Tab. 7.1), die für die spätere Umrechnung der gemessenen Pixel auf die tatsächliche Entfernung relevant ist. Die nachfolgende Tabelle 7.1 zeigt diese Datensätze für die optische Entfernungsmessung einschließlich der Beschreibung, der Annotation und der Anzahl des einzelnen Datensatzes im Überblick. Die Daten aus der Simulation konnten mit dem bereits in Kapitel 3 vorgestellten Simulator automatisch erzeugt und annotiert werden. Die Daten aus dem Modellbaubereich und der realen Welt, wurden für diesen Zweck manuell erstellt und annotiert. Diese beiden Vorgehen sind in den nächsten Abschnitten 7.3.1 und 7.3.2 genauer erklärt. Die Auflösung ist in dem Format Breite × Höhe angegeben. Die Abbildung 7.1 zeigt einige Bilder aus den erstellten Datensätzen. Der Datensatz 1 (Sim) enthält ein Simulationsfahrzeug mit den entsprechenden Entfernungen (Unity Einheit) mit 795 Bildern (Abb. 7.1a, 7.1b und 7.1c). Der Datensatz 2

(Mod) enthält ein Modellauto (*PiCar*) aus dem Modellbaubereich mit den entsprechenden Entfernungen (Meter) mit 30 Bildern (Abb. 7.1d, 7.1e und 7.1f).

Tabelle 7.1: Erstellte Datensätze für die Entfernungsmessung. Die Anzahl steht für die Menge der annotierten Bilder. Die Annotation ist als Einheit der Entfernung angegeben.

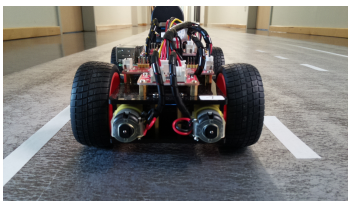
Nr.	Name	Beschreibung	Annotation	Anzahl
1	Sim	Entfernungen aus dem Unity Simulator (simuliertes Fahrzeug)	Unity Einheit	795
2	Mod	Entfernungen aus dem Modellbaubereich (Modellfahrzeug)	Meter	30
3	Real	Entfernungen aus der realen Umgebung (echtes Kraftfahrzeug)	Meter	24

Der letzte Datensatz 3 (Real) enthält ein echtes Fahrzeug aus der realen Welt mit den entsprechenden Entfernungsmessungen (Meter) mit 24 Bildern (Abb. 7.1g, 7.1h und 7.1i). Die Auflösung der Datensätze 1 und 2 ist 1280 × 720 Pixel.

(a) Datensatz 1 (Sim) nah.

(b) Datensatz 1 (Sim) mittig.

(c) Datensatz 1 (Sim) fern.



(d) Datensatz 2 (Mod) nah.

(e) Datensatz 2 (Mod) mittig.

(f) Datensatz 2 (Mod) fern.

(g) Datensatz 3 (Real) nah.

(h) Datensatz 3 (Real) mittig.

(i) Datensatz 3 (Real) fern.

Abbildung 7.1: Annotierte Datensätze für die Entfernungsmessung mit Datensatz 1 (Sim) aus dem Simulator mit simuliertem Auto, Datensatz 2 (Mod) aus dem Modellbaubereich mit einem Modellfahrzeug (*PiCar*) und Datensatz 3 (Real) aus der realen Umgebung mit einem echten Personenkraftfahrzeug.

In den Abbildungen 7.1g, 7.1h und 7.1i kann man ebenfalls erkennen, dass die Bilder in der Höhe etwas kleiner sind. Die Auflösung hierbei ist 1280 × 500 Pixel. Die für die Entfernungsmessung irrelevanten Stellen (Motorhaube des eigenen Fahrzeugs) wurden aus diesen Bildern entfernt und somit das Bild verkleinert. An dieser Stelle ist diese Auflösung völlig ausreichend, da alle Bilder im nächsten Schritt auf die Auflösung 320 × 160 Pixel komprimiert werden. Auf diese Auflösung wird in dem Abschnitt 7.4 mehr eingegangen.

7.3.1 Automatische Annotation

Ein Vorteil in der simulierten Umgebung zu arbeiten, ist die schnelle Erzeugung automatisch annotierter Datensätze. Durch das Rapid Prototyping können somit schnell aussagekräftige Ergebnisse, ohne einen aufwändigen Experimentaufbau in der realen Welt, erzeugt werden. Um die automatisch annotierten Daten für die Entfernungsmessung zu bekommen, wurde an dieser Stelle der Simulator für Rettungsgassenbildung und Unfallsimulationen auf Autobahnen aus Kapitel 3 verwendet. Die tatsächliche Entfernung zum vorderen simulierten Fahrzeug wurde mit Raycasts gemessen (Abb. 7.2). Die Raycasts kann man sich als eine Linie in eine bestimmte Richtung mit einer bestimmten Länge vorstellen. Dies veranschaulicht die gelbe Linie zwischen dem roten simulierten Fahrzeug und dem grauen simulierten Fahrzeug in Abbildung 7.2.

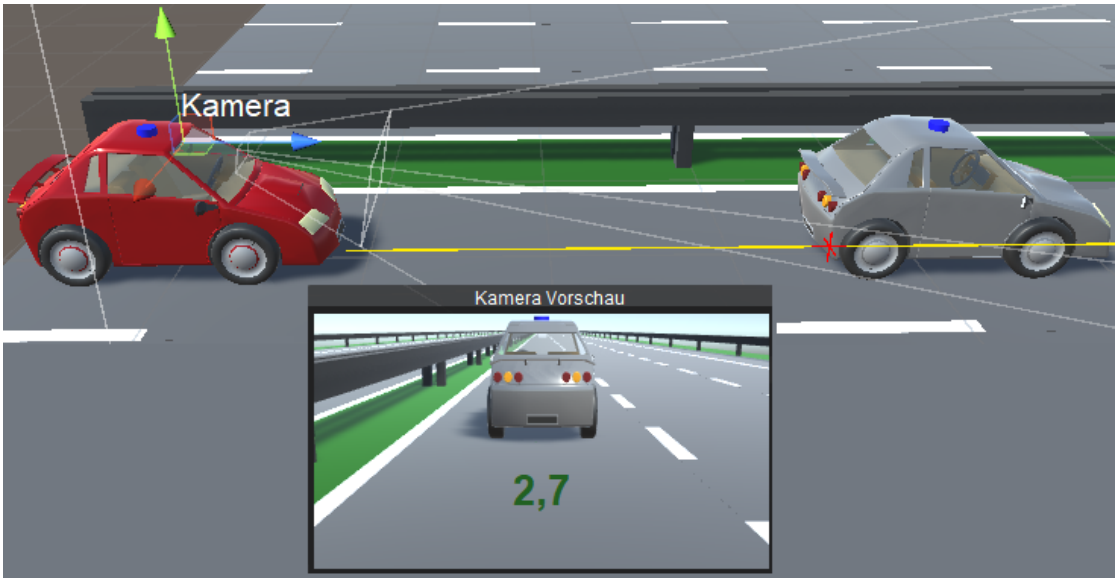


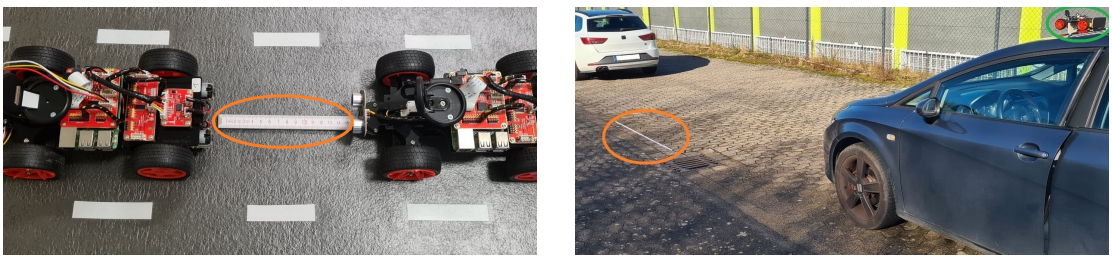
Abbildung 7.2: Automatische Datenbeschaffung und Annotation mit dem entwickelten Simulator aus Kapitel 3. Der Raycast für die Entfernungsmessung ist in gelb dargestellt. Die Entfernung in Unity Einheit ist in grün abgebildet.

Dabei kann die Entfernung zwischen zwei Objekten ermittelt werden. Hierbei handelt es sich um die Entfernung in der Unity Einheit (Nr. 1 in Tab. 7.1). Somit konnte automatisch ein Bild mit der entsprechenden Entfernung erzeugt und gespeichert werden. Mehr zu

den Raycasts beschreibt der Abschnitt 2.5.3. Ebenfalls können die verschiedenen Objekte ohne einen großen Aufwand ausgetauscht werden. Somit kann der Datensatz aus der Simulation beliebig um weitere Objekte erweitert und automatisch annotiert werden. Das ist zusätzlich ein weiterer Vorteil der simulierten Umgebung.

7.3.2 Manuelle Annotation

Die Datensätze aus dem Modellbaubereich und aus der echten Umgebung wurden manuell erstellt und annotiert. Dieses Vorgehen veranschaulicht die Abbildung 7.3. Um die Datensätze aus dem Modellbaubereich für ein Modellauto zu erhalten, wurde eine kleine Teststrecke mit zwei Modellfahrzeugen aufgebaut (Abb. 7.3a). Wie bereits in Kapitel 5 angesprochen, haben diese Modellautos jeweils eine Kamera verbaut. Dadurch ist es möglich, Bilder in einer bestimmten Entfernung von dem vorderen Objekt (Modellauto) zu erzeugen. Zuerst wurden diese Modellfahrzeuge in einem bestimmten Abstand von einander gestellt. Als nächstes wurde die Entfernung zum vorderen Modellfahrzeug mit einem Maßband gemessen (orangefarbiger Kreis in Abb. 7.3a). Anschließend wurde mit dem hinteren Modellauto manuell das vordere Modellfahrzeug fotografiert. Somit hatte man ein zweidimensionales Eingabebild mit einer tatsächlichen Entfernung als Annotation in Meter. Diese Schritte wurden solange wiederholt, bis ein notwendiger Datensatz entstanden ist (Nr. 2 in Tab. 7.1). Ein ähnliches Vorgehen wurde mit den echten Fahrzeugen aus der realen Welt realisiert (Abb. 7.3b). Die Messung der Entfernung für die Annotation wurde an dieser Stelle ebenfalls mit einem Maßband durchgeführt (orangefarbiger Kreis in Abb. 7.3b). Um die Randbedingungen gleich zu halten, wurden die Bilder gleichermaßen mit dem Modellfahrzeug, mit der gleichen Kamera erzeugt (Nr. 3 in Tab. 7.1). Dieses Modellauto befand sich auf dem Dach des echten grauen Fahrzeugs (grünfarbiger Kreis in Abb. 7.3b). Es wurde darauf geachtet, dass das Modellauto parallel zu der Fahrbahn ausgerichtet war, um die Ergebnisse nicht zu verfälschen. Als eine Alternative, könnte eine Dashcam in dem echten Fahrzeug verbaut werden. Diese könnte auch die notwendigen Bilder liefern.



(a) Mit echten Modellfahrzeugen.

(b) Mit echten Personenkraftfahrzeugen.

Abbildung 7.3: Manuelle Datenbeschaffung und Annotation für die Entfernungsmessung in der echten Umgebung. Orangefarbiger Kreis markiert das Maßband für die Messung. Grünfarbiger Kreis zeigt das Modellfahrzeug mit einer Kamera für die Aufnahme der Bilder.

7.4 Funktionsweise

Nachdem die Beschaffung der notwendigen Daten abgeschlossen war, konnte mit der Entfernungsmessung in einem zweidimensionalen Bild in Simulation, Modellbaubereich und realen Welt begonnen werden. Der Ansatz hierbei ist es die Entfernungsmessungen ohne ein Referenzobjekt zu realisieren. Dabei wird die echte Größe eines Objekts in dem gleichen Eingabebild nicht benötigt. Der Grundgedanke ist es hierbei eine Funktion für die Umrechnung der gemessenen Pixel auf die echte Distanz durch nicht lineare Regression zu approximieren. Dabei wurden zwei Vorgehen für die Entfernungsmessung, ohne der Transformation in die Vogelperspektive und mit der Transformation in die Vogelperspektive jeweils für die Simulationsdaten, Daten aus dem Modellbaubereich und Datensätzen aus der realen Welt, überlegt. Durch die Transformation in die Vogelperspektive bekommt man die Sicht auf die Fahrbahn von oben. Dieses Vorgehen wird in dem Abschnitt 2.3.5 mehr erläutert. Die Pixelmessungen für die Entfernung beginnen jeweils am unteren Rand des Eingabebilds. Zusätzlich wurde das Eingabebild auf die Auflösung 320 × 160 Pixel komprimiert. Diese Auflösung wurde bereits für die Echtzeitspurerkennung mit dem erweiterten Canny-Algorithmus verwendet. Diese Spurerkennung wird in dem Kapitel 4 mehr erläutert. Aus diesem Grund ist die Entscheidung für diese Auflösung für die Entfernungsmessung als Ausgangspunkt gefallen. Dadurch kann im nächsten Schritt die Position der Objekte auf der Fahrbahn bestimmt werden. Einige Vorexperimente zeigen: Die Umrechnung der Pixel auf die tatsächliche Entfernung hängt von der Auflösung des Eingabebilds, von der Neigung der Kamera zur Fahrbahn und der Höhe der Kamera von der Fahrbahn ab. Stimmen diese überein, können die approximierten Funktionen verwendet werden. Somit kann zum Beispiel das jeweilige autonome Fahrzeug einmalig bei der Auslieferung im Werk kalibriert werden. Die Pixelberechnung erfolgt als eine euklidische Distanz in Pixel zum vorderen Fahrzeug in einem Bild [65]. Hierbei wurde eine lineare Entfernungsmessung zum vorderen Objekt durchgeführt. Auch wenn die Fahrbahn gekrümmt ist, kann die lineare Entfernungsmessung durchgeführt werden. Die Entfernungsmessung erfolgt bis zu der unteren Kante des erkannten Objekts. Für die Erkennung des Objekts wurde die in Kapitel 6 vorgestellte Echtzeitobjekterkennung für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte genutzt. Somit musste die Bounding Box des Objekts nicht manuell definiert werden. Wie bereits angesprochen wurde die Stoßstange des Modellfahrzeugs bei der Objekterkennung entfernt, um dem Fahrzeug aus der hinteren Sicht mehr Merkmale zu verleiten.

Für die Transformation des Eingabebilds in die Vogelperspektive wird eine Transformationsmatrix benötigt. Die notwendigen Source (src)- und Destination (dst)-Parameter für die Erzeugung der Umrechnungsmatrix (Transformationmatrix) wurden experimentell durch Probieren gefunden. Diese Parameter sehen für die 320 × 160 Pixel Bilder in dem Format $xmin$, $xmax$, $ymin$ und $ymax$ wie folgt aus:

```
src = [[0, 0], [320, 0], [0,     135], [320,   135]]
dst = [[0, 0], [320, 0], [144,   135], [176,   135]]
```

Für die Daten aus der echten Welt mit echten Fahrzeugen ändert sich die Destination auf:

```
dst = [[0, 0], [320, 0], [156.8, 135], [163.2, 135]]
```

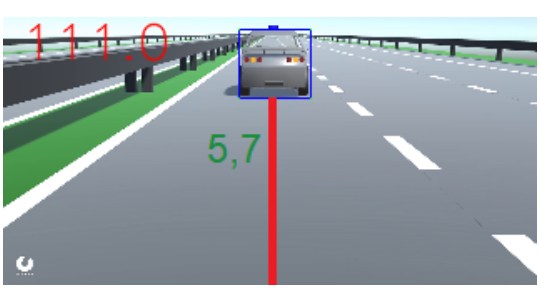
Sobald die notwendigen Parameter für die Transformation gefunden und gesetzt sind, kann mit der Entfernungsmessung in Simulation, im Modellbaubereich und in der echten Umgebung begonnen werden. Dieses Vorgehen beschreiben die nächsten Abschnitte.

7.4.1 Simulation

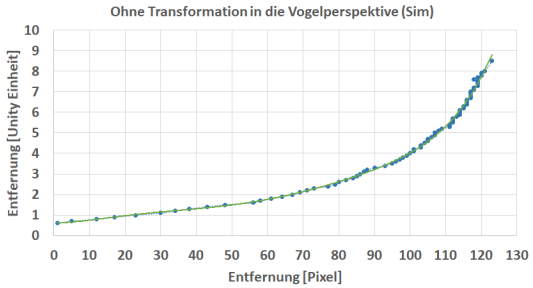
Die Entfernungsmessung wurde zuerst auf den Simulationsdaten durchgeführt, um zu überprüfen, ob eine optische Entfernungsmessung ohne ein Referenzobjekt überhaupt möglich ist. Wie bereits angesprochen, ist an dieser Stelle eine Echtzeitobjekterkennung die Voraussetzung für eine Echtzeitentfernungsmessung. Für die Messung der Entfernung in Pixel muss die Position des Objekts in Pixel bekannt sein. Als Erstes wurde geprüft, ob die Entfernungsmessung direkt auf dem Eingabebild (Abb. 7.4a), ohne die Transformation des Eingabebilds in die Vogelperspektive, erfolgen kann. Die Abbildung 7.4 zeigt die direkte Berechnung der euklidischen Distanz in einem Eingabebild, ohne die Umrechnung des Eingabebilds in die Vogelperspektive. Wie man in dem Diagramm in der Abbildung 7.4b erkennen kann, ist die maximale euklidische Distanz 123 Pixel und die tatsächliche Entfernung $8,5$ Unity Einheiten (ue). Aus den gesammelten Daten ergibt sich somit die folgende approximierte Funktion für $0 \leq x \leq 123$:

$$s_1(x) = 4\text{e}{-11}x^6 - 1\text{e}{-08}x^5 + 2\text{e}{-06}x^4 - 9\text{e}{-05}x^3 + 0.0023x^2 - 0.003x + 0.6281, \quad x \in \{0, 1, \ldots, 123\} \tag{7.1}$$

In der Gleichung 7.1 ist zu erkennen, dass die approximierte Funktion ein Polynom sechsten Grades (Abb. 7.4b) ist. Mit der zunehmenden Entfernung in Unity Einheit gibt es immer weniger Pixel, die gemessen werden können.



(a) Eingabebild mit Objekterkennung (blau). Rot: Gemessene Entfernung in Pixel. Grün: Tatsächliche Entfernung in Unity Einheit.



(b) Graph (Gl. 7.1) eines approximierten Polynoms sechsten Grades (grün). Blau: Tatsächlich gemessenen Entfernungen.

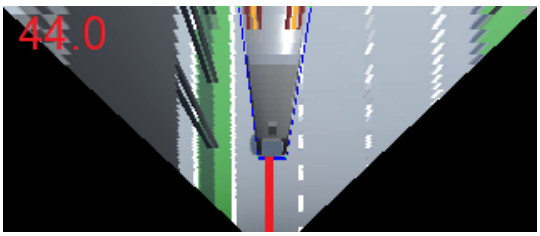
Abbildung 7.4: Entfernungsmessung auf dem Datensatz 1 (Sim) aus der Simulation ohne der Transformation in die Vogelperspektive.

Dies liegt daran, dass sich das Objekt in die „Tiefe“ des Bilds bewegt. Somit wurde als nächstes der zweite Ansatz, eine Entfernungsmessung mit der Transformation des

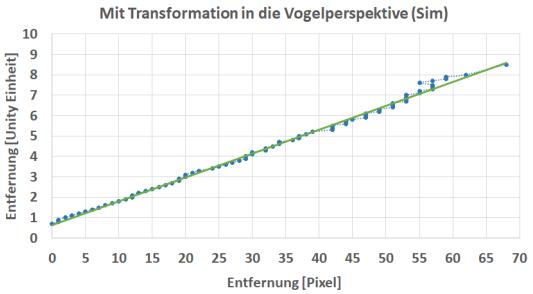
Eingabebilds in die Vogelperspektive, ausprobiert. Die nachfolgende Abbildung 7.5a zeigt die Transformation des Eingabebilds (Abb. 7.4a) in die Vogelperspektive aus dem Datensatz mit simulierten Fahrzeugen. Die rote Linie zeigt die errechnete euklidische Distanz zum gemessenen Objekt in Pixel. Das umgerechnete Eingabebild in die Vogelperspektive ist in der Höhe kleiner. Dies ist eine Folge der Transformation. Die Position des Objekts kann ebenfalls mit der erzeugten Matrix aus der Transformation in die Vogelperspektive umgerechnet werden. Die Objekterkennung erfolgt auf dem originalen Eingabebild. Somit ist die Umrechnung der beiden unteren Koordinaten der Bounding Box in die Vogelperspektive für die Funktionalität völlig ausreichend. Die Abbildung 7.5a zeigt ebenfalls die komplette Umrechnung aller Pixel des Eingabebilds inklusive der Position des Objekts in die Transformation der Vogelperspektive. An dieser Stelle ist die Umrechnung aller Pixel nur für die bessere Darstellung notwendig und wird bei der Laufzeitmessung in Abschnitt 7.5.2 ausgeschlossen. Die maximale euklidische Distanz ändert sich hierbei auf 68 Pixel bei einer tatsächlichen maximalen Entfernung von $8,5$ Unity Einheiten. Nach der erfolgreichen Berechnung der euklidischen Distanz in Pixel, kann eine lineare Funktion für $0 \leq x \leq 68$ approximiert werden:

$$s_2(x) = 0.117x + 0.6465, \quad x \in \{0, 1, \ldots, 68\} \tag{7.2}$$

Diese lineare Funktion (Gl. 7.2) dient der Umrechnung aus der errechneten euklidischen Distanz in die tatsächliche echte Distanz. In der Simulation ist es die Unity Einheit. Die nachfolgende Abbildung 7.5b zeigt ein Diagramm dieser approximierten linearen Funktion für den Datensatz aus der Simulation. Die Punkte in blau zeigen die echten gemessenen Distanzen aus dem Datensatz 1.



(a) Transformiertes Bild in die Vogelperspektive. Blau: Transformiertes erkanntes Objekt. Rot: Euklidische Distanz in Pixel.



(b) Graph (Gl. 7.2) einer approximierten linearen Funktion (grün). Blau: Tatsächlich gemessenen Entfernungen.

Abbildung 7.5: Entfernungsmessung auf dem Datensatz 1 (Sim) aus der Simulation mit der Transformation in die Vogelperspektive.

Wie man erkennen kann, wird durch die Transformation in die Vogelperspektive eine Linearität in die Messung hineingebracht. Eine lineare Regression ist leichter zu bestimmen und zusätzlich auch schneller umzurechnen. Aus diesem Grund wird nur der Ansatz mit der Umrechnung der Eingabebilder in die Transformation der Vogelperspektive bei der

Entfernungsmessung weiterverfolgt. Die Neigung der Kamera in der Simulation ist 11°. Die Höhe der Kamera von der Fahrbahn ist 0,67 Unity Einheiten.

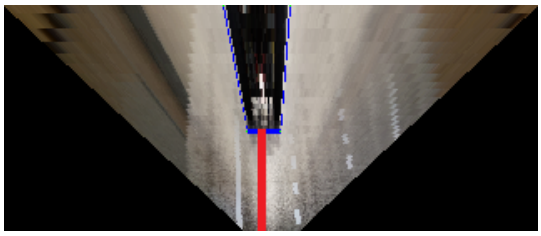
7.4.2 Modellfahrzeuge

Ähnlich wie bei den Daten aus der Simulation, wurde die Entfernungsmessung ebenfalls auf dem Datensatz 2 (Mod) aus dem Modellbaubereich durchgeführt. Das Vorgehen für die Entfernungsmessung ist hierbei das gleiche. Wie bereits angesprochen, wird nur der Ansatz der Entfernungsmessung in dem transformierten Eingabebild in die Vogelperspektive verfolgt. Als erstes muss die Position des vorderen Objekts im Eingabebild bekannt sein. Anschließend kann das Eingabebild in die Vogelperspektive transformiert werden. Wie bereits erwähnt wird für die Transformation in die Vogelperspektive eine Transformationsmatrix benötigt. Mit dieser, in der Transformation erzeugten Transformationsmatrix, kann die Position des zuvor erkannten Objekts ebenfalls in die Vogelperspektive umgerechnet werden. Anschließend kann die euklidische Distanz in Pixel von der unteren Kante des Eingabebilds bis zur unteren Kannte des erkannten Objekts gemessen werden. Diese gemessene Entfernung in Pixel kann anschließend mit einer approximierten Funktion (Gl. 7.3) auf die tatsächliche Entfernung in Meter umgerechnet werden. Diese lineare Funktion kann erfolgreich für den Datensatz 2 aus dem Modellbaubereich für $0 \leq x \leq 97$ approximiert werden:

$$m(x) = 0.0142x + 0.0092, \quad x \in \{0, 1, \ldots, 97\} \tag{7.3}$$

Die maximale euklidische Distanz ist 97 Pixel und die tatsächliche Entfernung ist 1,37 Meter. Die nachfolgende Abbildung 7.6 zeigt die Entfernungsmessung inklusive der Transformation in die Vogelperspektive auf dem Datensatz aus dem Modellbaubereich im Einsatz. Die Abbildung 7.6a veranschaulicht das Eingabebild inklusive der Objekterkennung (blau) und der euklidischen Distanz aus der Vogelperspektive (rot) mit einem Modellfahrzeug (*PiCar*).

(a) Eingabebild mit Objekterkennung. Blau: Erkanntes Objekt. Rot: Euklidische Distanz in Pixel aus der Vogelperspektive.



(b) Transformiertes Eingabebild in die Vogelperspektive. Blau: Transformiertes erkanntes Objekt. Rot: Euklidische Distanz.

Abbildung 7.6: Entfernungsmessung auf dem Datensatz 2 (Mod) aus dem Modellbaubereich mit der Transformation in die Vogelperspektive.

Die rote Linie in der vorherigen Abbildung 7.6b zeigt die euklidische Distanz zum gemessenen transformierten Objekt (blau) in Pixel. Die approximierte Funktion (Gl. 7.3) ist in der Abbildung 7.7 als ein Diagramm dargestellt. Der Graph dieser linearen Funktion ist in grün angegeben. Die blauen Punkte beschreiben die tatsächlich gemessenen Entfernungen. Sofern die Neigung der Kamera und Höhe der Kamera zum Boden übereinstimmt, kann diese für die Umrechnung der Pixel auf die tatsächliche Distanz verwendet werden. Die Neigung der Kamera ist an dieser Stelle 20° und die Höhe der Kamera von der Fahrbahn ist 10 Zentimeter (cm). An dieser Stelle wird die Entfernung in Meter angegeben. Auf die Fehler der Umrechnung wird bei den Experimenten in Abschnitt 7.5 eingegangen.

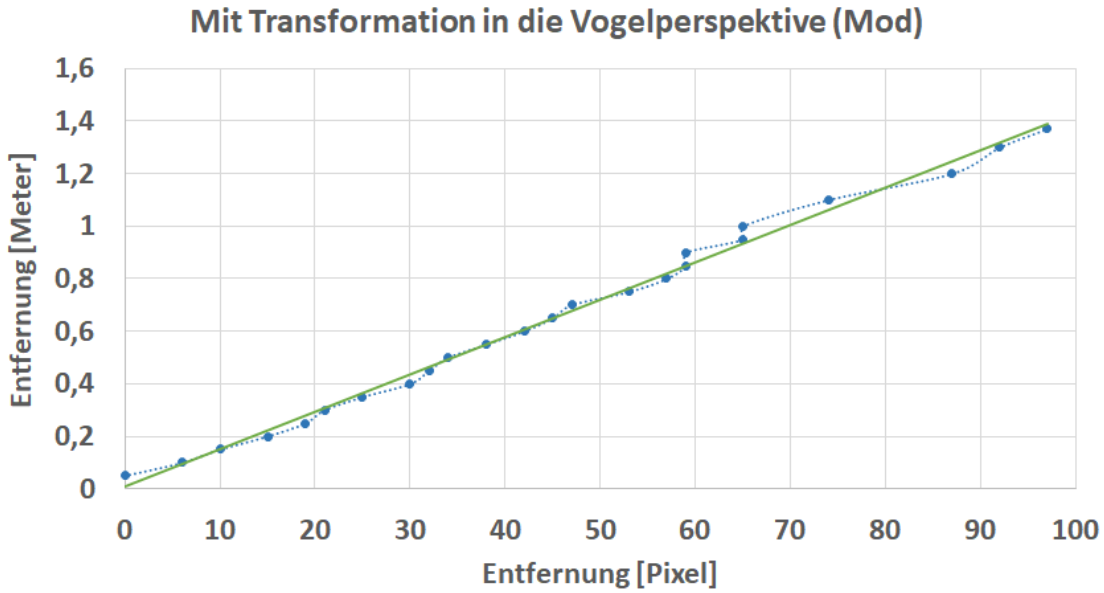


Abbildung 7.7: Funktionsapproximation auf dem Datensatz 2 (Mod) aus dem Modellbaubereich. Grün: Graph einer angenäherten linearen Funktion (Gl. 7.3). Blau: Tatsächlich gemessenen Entfernungen.

7.4.3 Echte Fahrzeuge

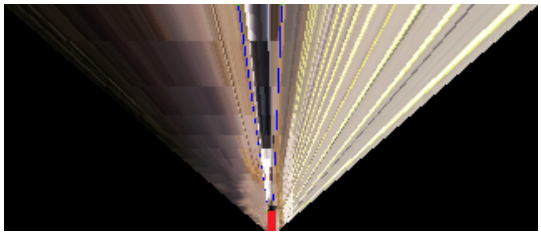
Nachdem die Entfernungsmessung für die Daten aus der Simulation und dem Modellbaubereich erfolgreich durchgeführt werden konnte, wurde der Datensatz 3 (Real) aus der echten Welt mit echten Fahrzeugen untersucht. Bei der Entwicklung waren die Simulation und der Modellbaubereich im Fokus. Doch es war ebenfalls spannend zu wissen, ob auch mit echten Fahrzeugen in der realen Welt die Entfernung erfolgreich gemessen werden kann. Das Vorgehen ist hierbei das gleiche wie in der Simulation und dem Modellbaubereich. Auch für die echten Fahrzeuge aus der echten Welt, kann erfolgreich eine lineare Funktion (Gl. 7.4) approximiert werden. Wie bereits aus den vorherigen Abschnitt bekannt, kann anschließend diese lineare Funktion für die Umrechnung der euklidischen Distanz in Pixel auf die tatsächliche Entfernung in Meter verwendet werden. An dieser

Stelle ist die maximale euklidische Distanz 40 Pixel und die tatsächliche Entfernung ist 60 Meter. Diese approximierte lineare Funktion für die Personenkraftfahrzeuge aus der realen Welt sieht für $0 \leq x \leq 40$ wie folgt aus:

$$r(x) = 1.483x + 0.6982, \quad x \in \{0, 1, \ldots, 40\} \tag{7.4}$$

Die nachfolgende Abbildung 7.8 zeigt diese erfolgreiche Entfernungsmessung im Einsatz. In diesem Fall ist das Objekt ein echtes Personenkraftfahrzeug aus der realen Umgebung. An dieser Stelle kann man erkennen, dass die Entfernungsmessung auch auf den Daten aus der echten Welt mit echten Fahrzeugen erfolgreich funktioniert. Die Abbildung 7.8a veranschaulicht das Eingabebild inklusive der Objekterkennung (blau) und der euklidischen Distanz aus der Vogelperspektive (rot) mit einem echten Fahrzeug. Die rote Linie in Abbildung 7.8b zeigt die euklidische Distanz zum gemessenen transformierten Objekt (blau) in Pixel.

(a) Eingabebild mit Objekterkennung. Blau: Erkanntes Objekt. Rot: Euklidische Distanz in Pixel aus der Vogelperspektive.



(b) Transformiertes Eingabebild in die Vogelperspektive. Blau: Transformiertes erkanntes Objekt. Rot: Euklidische Distanz.

Abbildung 7.8: Entfernungsmessung auf dem Datensatz 3 (Real) aus der realen Umgebung mit der Transformation in die Vogelperspektive.

Ein Diagramm dieser linearen Funktion für die Entfernungsmessung mit echten Fahrzeugen aus der echten Umgebung zeigt die nachfolgende Abbildung 7.9. Der Graph dieser Funktion ist in grün angegeben. Die blauen Punkte beschreiben die tatsächlich gemessenen Entfernungen in Meter. In dem Diagramm kann man ebenfalls die maximale euklidische Distanz von 40 Pixel und die tatsächliche Entfernung von 60 Meter erkennen. Wie bereits angesprochen, wurde für diese Messungen das Modellfahrzeug auf das Dach des echten grauen Fahrzeugs (grünfarbiger Kreis in Abb. 7.3b) befestigt. Somit ergibt sich ebenfalls eine Neigung der Kamera von 20° bei einer Höhe der Kamera von der Fahrbahn von 142 Zentimeter (cm). An dieser Stelle ist ebenfalls wichtig zu erwähnen, dass für den Echtwelteinsatz des Systems mit echten autonomen Fahrzeugen mehr Experimente durchgeführt werden sollten.

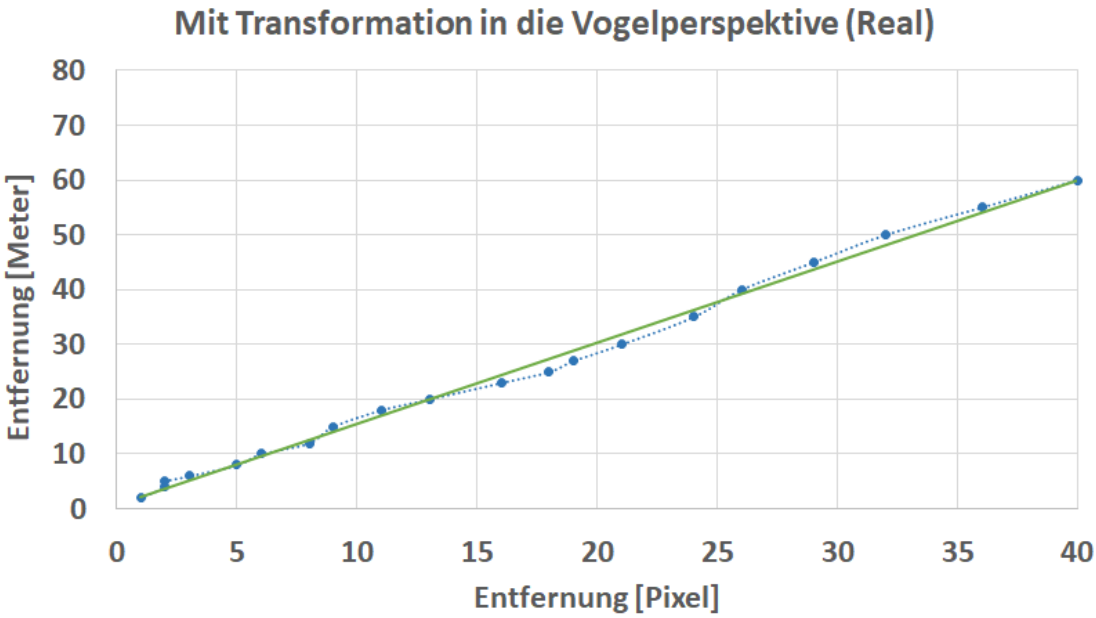


Abbildung 7.9: Funktionsapproximation auf dem Datensatz 3 (Real) aus der echten Umgebung. Grün: Graph einer angenäherten linearen Funktion (Gl. 7.4). Blau: Tatsächlich gemessenen Entfernungen.

7.5 Experimente

Die folgenden Experimente wurden durchgeführt, um die Genauigkeit und die Laufzeit der verschiedenen Auflösungen auf unterschiedlicher IoT-Hardware zu vergleichen. Die Auflösung wird in dem Format Breite × Höhe angegeben. Alle Experimente wurden auf der gleichen Hardware durchgeführt. Dies gibt die Möglichkeit, die Ergebnisse hinterher zu vergleichen und die optimale Entfernungsmessung für einen bestimmten Einsatz zu finden. Auch die Eingabebilder für die jeweiligen Experimente wurden gleich gehalten. Für die Hardware mit beschränkten Ressourcen wurden zwei Einplatinencomputer, Raspberry Pi 3 B und ein Raspberry Pi 4 B, verwendet. Die Laufzeit der Entfernungsmessung ist in Millisekunden (ms) angegeben. Diese Laufzeitmessungen beinhalten nur die Zeit für die Entfernungsmessung und nicht die Zeit für das Laden der Eingabebilder, die Zeit für die Spur- und die Objekterkennung. Die IoT-Hardware mit beschränkten Ressourcen für diese Experimente ist in Abschnitt 2.6.2 im Detail beschrieben.

7.5.1 Auflösung und Genauigkeit

In diesen Experimenten werden verschiedene Auflösungen für die Eingabebilder bei der Entfernungsmessung untersucht. Zusätzlich wird die Genauigkeit dieser Bilder gegenübergestellt. Um die Qualität des optischen Kontrollsystems des Radarsensors und der Entfernungsmessung für verschiedene Auflösungen zu überprüfen, wurde der mittlere

absolute Fehler (MAE) für den Fehler der Messung errechnet. Dieser Messfehler ist für den Datensatz 1 (Sim) in Unity Einheit (ue) und für die Datensätze 2 (Mod) und 3 (Real) in Meter (m) angegeben. Zusätzlich wurde dieser MAE-Wert auf die maximale Entfernung prozentual umgerechnet (MAE in %). Mehr dazu findet man in dem Abschnitt 2.2.3. Durch diese prozentuale Umrechnung ist zum Beispiel der MAE von 0,02 (Exp. Nr. 6 in Tab. 7.2) mit 0,88 (Exp. Nr. 9 in Tab. 7.2) vergleichbar. Der Datensatz 1 (Sim) liefert eine maximale Entfernung von 8,5 Unity Einheiten. Der Datensatz 2 (Mod) hat eine Gesamtentfernung von 1,37 Meter. Der Datensatz 3 (Real) hingegen liefert eine maximale Entfernung von 60 Meter. Die gemessenen Werte sind auf zwei Stellen nach dem Komma gerundet.

Tabelle 7.2: Übersicht der Auflösung und des Messfehlers der Entfernungsmessung. Die erste Spalte enthält die Identifikationsnummer (ID) des Experiments (Exp. Nr.). Der Messfehler ist für den Datensatz 1 (Sim) in Unity Einheit (ue) und für die Datensätze 2 (Mod) und 3 (Real) in Meter (m) angegeben.

Exp. Nr.	**Auflösung [Pixel]**	**Datensatz**	**Messfehler**	
			MAE	**MAE [%]**
1	320 × 160	Sim	0,09	1,06
2	320 × 320	Sim	0,09	1,06
3	320 × 640	Sim	0,08	0,94
4	320 × 160	Mod	0,02	1,46
5	320 × 320	Mod	0,02	1,46
6	320 × 640	Mod	0,02	1,46
7	320 × 160	Real	0,97	1,62
8	320 × 320	Real	0,89	1,48
9	320 × 640	Real	0,88	1,47

Schaut man sich die Ergebnisse aus der Tabelle 7.2 an, kann man erkennen: Bei größerer Auflösung in der Höhe gibt es mehr Pixel für die Entfernungsmessung. Somit gibt es weniger doppelte Pixel für verschiedene tatsächliche Entfernungen. Dies liegt daran, dass sich das Objekt in die „Tiefe“ des Bilds bewegt. Aus diesem Grund ist der mittlere absolute Fehler (MAE) auch bei höherer Auflösung kleiner (Vergleich zwischen Exp. Nr. 1 und 3 in Tab. 7.2).

7.5.2 Auflösung und Laufzeit

Um einen geeigneten Ansatz für die Entfernungsmessung zu finden, sollte zusätzlich zu der Genauigkeit der Entfernungsmessung ebenfalls die Laufzeit angeschaut werden. Somit wurden bei diesen Experimenten einige Messungen auf Hardware mit beschränkten Ressourcen durchgeführt. Wie bereits angesprochen, wurden diese Laufzeitmessungen auf dem Raspberry Pi 3 B und Raspberry Pi 4 B durchgeführt. Kleine Vorexperimente zeigen: Die Berechnung der euklidischen Distanz hängt nicht von der Entfernung des Objekts in dem Eingabebild ab. Somit wurde das Objekt fest an die obere Kante des Eingabebilds

gesetzt. Die Messung der euklidischen Distanz wurde somit für alle Datensätze und für jede Laufzeitmessung gleich gehalten, um die Ergebnisse hinterher vergleichen zu können. Aus dem jeweiligen Datensatz wird die durchschnittliche Laufzeit auf verschiedener Hardware errechnet. Die Laufzeitmessungen beinhalten nur die Transformation der beiden unteren Koordinaten des entfernten Objekts in die Vogelperspektive, die Berechnung der euklidischen Distanz zum entfernten Objekt und die Umrechnung der euklidischen Distanz auf die tatsächliche Entfernung. Für die Umrechnung der gemessenen Pixel auf die tatsächliche Entfernung werden die bereits vorgestellten approximierten linearen Funktionen aus dem Kapitel 7.4 verwendet. Die Laufzeitmessungen beinhalten nicht die Transformation der Pixel des kompletten Eingabebilds in die Vogelperspektive. Dies ist nur für die bessere Darstellung der Ergebnisse und nicht für die Entfernungsmessung notwendig. Ebenfalls ist die Laufzeit abhängig von der Anzahl der Objekte im Eingabebild. Für jedes Objekt muss eine Transformation in die Vogelperspektive, die Berechnung der euklidischen Distanz und die Umrechnung der euklidischen Distanz auf die tatsächliche Entfernung erfolgen. Die nachfolgende Tabelle 7.3 zeigt die Auflösung und die Laufzeit der Entfernungsmessung auf einen Blick.

Tabelle 7.3: Übersicht der Auflösung und der Laufzeit der Entfernungsmessung auf dem Raspberry Pi 3 B und dem Raspberry Pi 4 B. Die erste Spalte enthält die Identifikationsnummer (ID) des Experiments (Exp. Nr.). Die Laufzeit ist in Millisekunden (ms) angegeben.

Exp. Nr.	**Auflösung [Pixel]**	**Datensatz**	**Laufzeit [ms]**	
			RPI 3 B	**RPI 4 B**
1	320 × 160	Sim	0,7	0,4
2	320 × 320	Sim	0,7	0,4
3	320 × 640	Sim	0,7	0,4
4	320 × 160	Mod	0,7	0,4
5	320 × 320	Mod	0,7	0,4
6	320 × 640	Mod	0,7	0,4
7	320 × 160	Real	0,7	0,4
8	320 × 320	Real	0,7	0,4
9	320 × 640	Real	0,7	0,4

7.5.3 Auswertung der Ergebnisse

Nach dem die Experimente und die Tests der Performance abgeschlossen wurden, kann mit der Evaluation der Laufzeiten und der Messfehler begonnen werden. Hierbei ist es wichtig eine Balance zwischen der Genauigkeit und der Geschwindigkeit herzustellen, um ein geeignetes Kontrollsystem für den Radarsensor zu finden. Die Tabelle 7.2 zeigt, dass die Entfernungsmessung mit verschiedenen Auflösungen auf den vorgestellten Datensätzen erfolgreich durchgeführt werden konnte. Der Fehler der Messung schwankt hierbei zwischen 0,94 und 1,62 % (Exp. Nr. 3 und 7 in Tab. 7.2), was für eine experimentelle Anwendung

völlig akzeptabel ist. Diese Entfernungsmessung wurde erfolgreich untersucht und kann sowohl auf den Simulationsdaten, Daten aus dem Modellbaubereich und auf Daten aus der realen Welt auf echten Fahrzeugen angewendet werden. Ebenfalls kann man erkennen: Die Auflösung wird größer, die Laufzeit der Entfernungsmessung vergrößert sich nicht, da die Pixel des Eingabebilds nicht komplett in die Vogelperspektive umgerechnet werden (Vergleich zwischen Exp. Nr. 7 und 9 in Tab. 7.3). Es werden nur die Pixel der erkannten Objekte in die Vogelperspektive transformiert. Zusätzlich kann man erkennen, dass der Raspberry Pi 3 B langsamer bei der Berechnung der Entfernung als der Raspberry Pi 4 B ist, was auch anhand der technischen Daten zu erwarten ist (Exp. Nr. 1 in Tab. 7.3).

Als nächstes werden die Eingabebilder aus dem Datensatz 2 (Mod) mit der Auflösung von 320 $\times$ 160 Pixel auf dem Raspberry Pi 4 B betrachtet. Dieser benötigt für die Entfernungsmessung ca. 0,4 Millisekunden (Exp. Nr. 4 in Tab. 7.3). Der Messfehler bei dieser optischen Entfernungsmessung ist ca. 1,46 % (Exp. Nr. 4 in Tab. 7.2). Somit enthält man zum Beispiel eine Abweichung von ca. $1,5$ Zentimeter auf 1 Meter. An dieser Stelle kann ebenfalls gesagt werden, dass der Messfehler der Entfernungsmessung auch von der Genauigkeit der verwendeten Objekterkennung abhängt. Je genauer die Objekterkennung ist, desto genauer ist auch die Entfernungsmessung zu diesem Objekt, da die Entfernung von der unteren Kante des Eingabebilds bis zu der unteren Kante des Objekts gemessen wird. Dabei spielt diese Position eine wichtige Rolle, denn die euklidische Distanz wird genau bis zu dieser Linie gemessen.

Sobald die bereits in Kapitel 6 vorgestellte Echtzeitobjekterkennung für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte mit ca. 30 Bilder pro Sekunde (FPS) verwendet wird, kann die tatsächliche Laufzeit für die Entfernungsmessung abhängig von der Objekterkennung errechnet werden. Jedes Bild wird um ca. 0,4 ms verlangsamt. Somit sinkt die tatsächliche Bildwechselfrequenz (Bildrate) auf ca. 29,6 FPS auf einem Raspberry Pi 4 B. Bei einem Raspberry Pi 3 B wird jedes Bild um ca. 0,7 ms verlangsamt. Somit wird eine Bildrate von ca. 29,4 FPS erreicht. Sobald bei der Ausführung auf die Technologie des Multithreadings zurückgegriffen wird, darf die Laufzeit der Objekterkennung und der Entfernungsmessung zusammen $33,\overline{3}$ Millisekunden bei einer Videoaufnahme von 30 Bilder pro Sekunde nicht überschreiten, um eine Echtzeitausführung zu erreichen. Alle vorgestellten Ansätze erreichen somit eine Echtzeitentfernungsmessung auf der verwendeten Hardware mit beschränkten Ressourcen.

7.6 Schlussfolgerungen

Die Entwicklung eines Kontrollsystems für den Radarsensor für autonome Fahrzeuge konzentrierte sich auf die Entfernungsmessung in Echtzeit. Die Nutzung dieses Systems auf Hardware mit beschränkten Ressourcen für IoT-Geräte mit geringem Stromverbrauch war die erste Priorität. Zu diesem Zweck wurden auch eigene Datensätze aus der Simulation, aus dem Modellbaubereich und von einem echten Personenkraftfahrzeug in einer realen Umgebung erstellt. Darüber hinaus wurden verschiedene Auflösungen und approximierte lineare Funktionen analysiert, um ein Gleichgewicht zwischen ausreichender Genauigkeit und der Laufzeit der Entfernungsmessung zu finden. Die Entfernungsmessung funktioniert

nach einem einfachen Prinzip. Dafür muss zuerst die Position des Objekts im Vordergrund als Ausgangspunkt bekannt sein. Als nächstes werden die Koordinaten des Objekts in die Vogelperspektive mit der vorgestellten Transformationsmatrix transformiert. Anschließend kann mit der Entfernungsmessung begonnen werden, indem die euklidische Distanz in Pixel von der unteren Kante des Eingabebilds bis zu der unteren Kante des Objekts berechnet wird. Diese euklidische Distanz kann im nächsten Schritt auf die tatsächliche Entfernung mit den vorgestellten linearen Funktionen umgerechnet werden. So benötigt die vorgestellte Entfernungsmessung aus dem Modellbaubereich ca. 0,4 Millisekunden mit einem 1,46 % Messfehler auf der vorgestellten IoT-Hardware. Durch diesen Messfehler erhält man eine Abweichung von ca. 1,5 Zentimeter auf 1 Meter. Die Entwicklung dieses Kontrollsystems konzentrierte sich auf die Simulation und den Modellbaubereich. Der Ansatz mit echten Fahrzeugen sollte zeigen, dass die Entfernungsmessung ebenfalls für den Echtwelteinsatz geeignet ist. Sobald diese Entfernungsmessung auf echten autonomen Fahrzeuge angewendet wird, sollte an dieser Stelle noch mehr Daten beschafft und mehrere Experimente durchgeführt werden.

Zusammenfassend, anhand der durchgeführten Experimente erkennt man, dass die Entfernungsmessung ohne ein Referenzobjekt in Simulation, Modellbaubereich und in der echten Umgebung funktioniert. Somit konnte erfolgreich ein optisches Kontrollsystem für den Radarsensor inklusive der Entfernungsmessung auf Hardware mit beschränkten Ressourcen präsentiert werden. Mit diesem Radarkontrollsystem wird ein effektives Gleichgewicht zwischen der Genauigkeit und der Geschwindigkeit erreicht. Wie bereits erwähnt, zeigt dieses Kapitel ein weiteres Szenario lauffähig auf Hardware mit beschränkten Ressourcen. Bei der Funktionsweise kommt man hierbei mit traditionellen Methoden aus, ohne die tiefen neuronalen Netzwerke zu verwenden. Wie die Experimente zeigen, ist dieses Vorgehen sehr schnell und in der Funktionalität in diesem Bereich völlig ausreichend. Solche Systeme können für weitere IoT-Geräte mit eingebauter Kamera, wie zum Beispiel smarte Drohnen, Überwachungskameras oder Wildkameras, relevant sein. In dem nächsten Kapitel 8 werden weitere mögliche Erweiterungen dieser Thematik vorgestellt, die in dieser Arbeit nicht mehr behandelt werden.

Was wir mathematisch festlegen, ist nur zum kleinen Teil ein objektives Faktum, zum größeren Teil eine Übersicht über Möglichkeiten.

Werner Karl Heisenberg
1901 – 1976

Kapitel 8

Zusammenfassung und Ausblick

8.1 Analyse der Innovationen

Dieser Abschnitt analysiert und fasst die vorgestellten Innovationen dieser Forschungsarbeit zusammen. Laut der Recherchen vor Beginn dieser Forschungsarbeit gab es bislang keine wissenschaftliche Forschung im Bereich der Rettungsgassenbildung mit autonomen Fahrzeugen auf Autobahnen. Zuallererst wurden in dem Kapitel 3 zwei verschiedene Algorithmen zur Rettungsgassenbildung mit autonomen Fahrzeugen in dieser Forschungsarbeit eingeführt, welche bislang in der wissenschaftlichen Forschung nicht publiziert waren. Wie die vorgestellten Experimente mit den simulierten autonomen Fahrzeugen und autonomen Modellfahrzeugen aus der Realität zeigen, ist es völlig ausreichend die Bildung der Rettungsgasse bei einem stockenden Verkehr durchzuführen. Selbst bei einer Gefahrenbremsung der autonomen Fahrzeuge, kann das autonome Fahrzeug gleichzeitig die Gefahrenbremsung durchführen und den notwendigen Algorithmus für die Bildung der Rettungsgasse ausführen. Falls es zu einer unvorhersehbaren Situation auf Autobahnen kommt und die Rettungsgasse durch autonome Fahrzeuge nicht gebildet werden konnte, kann der zweite eingeführte Algorithmus, die Bildung der Rettungsgasse bei einem stehenden Verkehr, auf Knopfdruck ausgeführt werden. Um die entwickelten Algorithmen zu evaluieren, wurde ebenfalls ein Simulator in Unity für verschiedene Anwendungsfälle entwickelt. Dieser in Kapitel 3 vorgestellte Simulator für Rettungsgassenbildung und Unfallsimulationen wurde ebenfalls inklusive des entwickelten und getesteten Algorithmus für die Bildung der Rettungsgasse bei einem stockenden Verkehr mit autonomen Fahrzeugen auf Autobahnen publiziert [58].

Wie bereits angesprochen wurden die entwickelten Algorithmen für die Rettungsgassenbildung ebenfalls in der Praxis mit echten Modellfahrzeugen untersucht. Dafür war eine bildliche Spurerkennung durch eine Kamera von Nöten. In Kapitel 4 konnte ein Vorgehen für eine schnelle und robuste Spurerkennung in Echtzeit erforscht und erfolgreich in dem entwickelten Simulator demonstriert werden. Dies konnte ebenfalls durch die

durchgeführten Experimente bestätigt werden. Dabei handelt es sich um die Erweiterung des Canny-Kantenerkennungsalgorithmus um einen Filter. Diese Spurerkennung hat aus diesem Grund den Namen der gefilterte Canny-Algorithmus (engl. Filtered Canny Edge Detection) bekommen. Der gefilterte Canny-Algorithmus wurde ebenfalls inklusive des Vorgehens, des Filterungsalgorithmus und einem Laufzeitenvergleich zu einem faltenden neuronalen Netzwerk veröffentlicht [55].

Viele wissenschaftliche Arbeiten stellen Algorithmen und Experimente vor, die auf einer leistungsstarken und kostenintensiven Hardware entwickelt und getestet wurden. Der Aspekt der Laufzeit wird teilweise auch nicht angesprochen. In dieser Forschungsarbeit wurde dieser Aspekt bewusst nicht außer Acht gelassen. Aus diesem Grund wurde auch in dieser Arbeit das Thema der Entwicklung der Software für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte eingeführt. Dadurch können bewusst die Kosten für den Stromverbrauch und die Hardware gesenkt werden. Bei allen in dieser Forschungsarbeit vorgestellten Experimenten in Kapitel 4, 6 und 7 wurden Vergleiche der Laufzeiten durchgeführt. Dadurch konnten verschiedene Systeme mit einem Gleichgewicht zwischen der Genauigkeit und der Geschwindigkeit gefunden werden. Zusätzlich stand in Kapitel 6 und 7 die Echtzeitausführung der entwickelten Systeme auf Hardware mit beschränkten Ressourcen im Vordergrund.

Damit die autonomen Modellfahrzeuge verschiedene Objekte auf deren Fahrbahn erkennen können, wurde eine Echtzeitobjekterkennung unter dem Aspekt der Entwicklung der Software für Hardware mit beschränkten Ressourcen in Kapitel 6 entwickelt. Es gibt sehr viele wissenschaftliche Forschungsarbeiten, die sich mit der Objekterkennung in Echtzeit beschäftigen. Doch die entwickelten Systeme werden meist auf einer leistungsstarken Hardware mit einer Grafikkarte erforscht. Laut den Recherchen während der Entwicklung gab es auf diesem Gebiet ebenfalls keine wissenschaftliche Forschung zu einer Echtzeitobjekterkennung auf Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte sowohl mit als auch ohne der Intel Neural Compute Stick 2 (Intel NCS2) Hardwareerweiterung. Die vorgestellte Objekterkennung kann ebenfalls auf eigenen Objekten durchgeführt und auf der Intel NCS2 Hardwareerweiterung ausgeführt werden. Für die Experimente wurden die Objekte sowohl aus der Simulation als auch aus dem Modellbaubereich verwendet. Zusätzlich wurde die Eingabegröße der verwendeten TensorFlow Modelle für die Objekterkennung verkleinert, um schnellere Auswertung und trotzdem eine gute Genauigkeit bei der Objekterkennung zu bekommen. Diese Modelle gab es zum Zeitpunkt der Entwicklung ebenfalls nicht in dem TensorFlow Erkennungsmodellzoo. Die Echtzeitobjekterkennung auf Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte wurde ebenfalls ohne [56] und mit der Hardwareerweiterung [54] publiziert.

Durch die Objekterkennung in Echtzeit auf Hardware mit beschränkten Ressourcen konnte anschließend in Kapitel 7 ein Kontrollsystem für den Radarsensor für autonome Personenkraftfahrzeuge und autonome Modellfahrzeuge entwickelt werden. Dadurch können Entfernungsmessungen in einem zweidimensionalen Bild zu den erkannten Objekten durch einfache und laufzeitarme Methoden der Funktionsapproximation durchgeführt werden. Dieses System zeigt ein gutes Beispiel dafür, dass nicht unbedingt die künstlichen neuronalen Netzwerke für jede Anwendung verwendet werden müssen. Das Radarkontrollsystem wurde ebenfalls inklusive der Entfernungsmessungen in einem zweidimensionalen

Bild durch die Funktionsapproximation veröffentlicht [57]. Durch die Entwicklung der Objekterkennung und des Radarkontrollsystems konnten zwei weitere Anwendungsfälle mit dem Aspekt der Entwicklung der Software für Hardware mit beschränkten Ressourcen für stromsparende IoT-Geräte vorgestellt werden.

8.2 Mögliche Erweiterungen der Arbeit

Dieser Abschnitt präsentiert mögliche Erweiterungen der einzelnen vorgestellten Themen dieser Forschungsarbeit. Der in dem Kapitel 3 vorgestellte Simulator wurde entwickelt, um im ersten Schritt, die in dieser Arbeit entwickelten und vorgestellten Algorithmen, ohne einen aufwendigen experimentellen Aufbau, zu untersuchen. Dieser Simulator ist für die Bildung der Rettungsgasse und Unfallsimulationen mit autonomen simulierten Fahrzeugen geeignet. Bevor dieser Simulator zum Beispiel als eine Open Source Software angeboten werden kann, können einige Funktionalitäten erweitert werden. Die grafische Bedienungsoberfläche könnte verbessert werden. Für die verschiedenen Einstellungs- und Auswahlmöglichkeiten könnte zusätzlich eine Konfigurationsoberfläche für den Benutzer programmiert werden. Zusätzlich könnten verschiedene vorprogrammierte Szenarien für die verschiedenen Unfälle zur Auswahl bereitgestellt werden. Der Simulator könnte ebenfalls um weitere Anwendungsfälle mit autonomen Fahrzeugen erweitert werden. Dabei wäre es interessant zu sehen, wie sich die autonomen Fahrzeuge verhalten, wenn zum Beispiel auf einer Baustelle das Reißverschlussverfahren angewendet wird oder diese im fließenden Verkehr auf eine Autobahn auffahren.

In dem Kapitel 4 wurden zwei Methoden für die Spurerkennung für den entwickelten Simulator aus Kapitel 3 vorgestellt. Diese beiden Methoden, der gefilterte Canny-Algorithmus und das faltende neuronale Netzwerk, könnten um ein Gedächtnis zur Überprüfung des Spurverlaufs erweitert werden. Für den gefilterten Canny-Algorithmus wäre es möglich, eine bestimmte Anzahl an zuvor bestimmten Punkten des Spurverlaufs temporär zu speichern. Dafür könnte sich eine Warteschlange als Datenstruktur eignen. Anschließend kann dafür eine maximal zulässige Veränderung gefunden und verglichen werden, denn pro Bildsequenz kann die Veränderung des Spurverlaufs nicht beliebig groß sein. Durch diesen Ansatz könnten Schwankungen und Fehler dieser Methode extrahiert werden und der Filterungsalgorithmus verbessert werden. Das faltende neuronale Netzwerk hingegen könnte zum Beispiel durch ein rekurrentes neuronales Netzwerk (engl. Recurrent Neural Network - RNN) mit einem langen Kurzzeitgedächtnis (engl. Long Short-term Memory - LSTM) ersetzt werden. Durch die Verbindungen von Neuronen der vorangegangenen Schicht können zeitlich zusammenhängende Informationen besser gefunden werden.

Um die entwickelten Vorgehensweisen und Algorithmen nicht nur in der Simulation, sondern auch in der Praxis zu untersuchen, wurden einige Modellfahrzeuge in dem Kapitel 5 zusammengebaut und präsentiert. Für den experimentellen Aufbau wurden kostengünstige Modellfahrzeuge verwendet, um den finanziellen Aspekt ebenfalls nicht zu überspannen. Die Karosserie dieser Modellfahrzeuge ist durch mehrere gestanzte Acrylplatten realisiert, die anschließend mit einigen Schrauben zusammengehalten werden.

Dadurch gibt es teilweise Ungenauigkeiten bei der Lenkung der Modellfahrzeuge. Ebenfalls sind die Räder nicht 100 % orthogonal zur Fahrbahn ausgerichtet, wie das bei einem echten Fahrzeug der Fall ist. Zusätzlich erkennt der verbaute Ultraschallsensor teilweise fehlerhafte Impulse in der echten Umgebung. Dies erschwerte die Kontrolle über das autonome Modellfahrzeug zu behalten. Ebenso kann die horizontale Erfassung dieses Sensors nicht zugeordnet werden. Dabei kann das erkannte Objekt nicht lokalisiert werden. Somit sollten mehrere Ultraschallsensoren vorne und hinten verbaut werden, um die Position der erkannten Hindernisse zu zuordnen.

Wie bereits in Kapitel 5 vorgestellt erfolgt die Kommunikation der Modellfahrzeuge über WLAN. Dieses kann im nächsten Schritt durch die Kommunikation durch eine 433 MHz Funkübertragung realisiert werden. Dazu wird eine 433 MHz fähige Hardware benötigt. Jedes Modellfahrzeug muss einen 433 MHZ-Sender und einen 433 MHZ-Empfänger enthalten. Zusätzlich können die Technologien der Bluetooth Low Energy (BLE) in Betracht gezogen werden. Diese Technologien verbrauchen möglicherweise weniger Energie und lassen die Modellfahrzeuge mit den gleichen Akkus länger laufen. Dabei muss an die Sicherheit und Zuverlässigkeit gedacht werden. Ebenfalls ist dabei die Geschwindigkeit der Übertragung sehr wichtig. Zusätzlich kann für die Modellfahrzeuge eine zentrale Stelle für die verschiedenen Sensorwerte, Warnungen, Debug- und Fehlermeldungen implementiert werden. Dabei kann die Funktion an die echten autonomen Fahrzeuge angelehnt werden. Für den Anwendungsfall und die Experimente dieser Forschungsarbeit war die Hardware und Konstruktion dieser Modellfahrzeuge völlig ausreichend. Es konnten alle notwendigen Untersuchungen mit den Modellfahrzeugen durchgeführt und vorgestellt werden. Für weitere Forschung mit den Modellfahrzeugen könnten die genannten Punkte allerdings verbessert werden.

In Kapitel 7 wurde ein Radarkontrollsystem für autonome Personenkraftfahrzeuge und autonome Modellfahrzeuge vorgestellt. Dieses Kontrollsystem, realisiert durch Pixelmessung in einem zweidimensionalen Bild und anschließender Umrechnung auf die tatsächliche Entfernung, ist auf die Fahrspur des eigenen Fahrzeugs beschränkt. Dies entspricht ebenfalls der Funktionsweise des echten Radarsensors. Als eine mögliche Erweiterung ist es auch denkbar, dieses Kontrollsystem um eine Abstandsmessung zu den erkannten Objekten außerhalb der eigenen Fahrspur zu erweitern. Dies kann zum Beispiel als eine Erweiterung der Funktionalität des Radarsensors in selbstfahrenden Fahrzeugen genutzt werden. Dadurch könnte zusätzlich das Objekt seitlich beobachtet und verfolgt werden. Denn der Erfassungsbereich einer Kamera ist wesentlich größer als der Erfassungsbereich des Radarsensors. Anschließend könnte die Bewegungsrichtung dieses Objekts ermittelt werden, um gegebenenfalls eine Kollision des Objekts mit dem eigenen Fahrzeug oder den Fahrzeugen in der Umgebung vorherzusehen. Um diese Erweiterung zu realisieren, sind weitere Experimente mit den Bildern aus der Vogelperspektive notwendig.

8.3 Fazit

Dieser Abschnitt stellt die Schlussfolgerungen dieser Forschungsarbeit vor und fasst die wichtigsten vorgestellten Ergebnisse fokussiert nochmal zusammen. Der Schwerpunkt

dieser Forschungsarbeit liegt auf der Entwicklung der Algorithmen und Software für stromsparende IoT-Geräte angewendet auf der Problematik der Rettungsgassenbildung auf Autobahnen mit autonomen Fahrzeugen. Dabei wurden zwei Algorithmen für die Bildung der Rettungsgasse beim stockenden und stehenden Verkehr entwickelt, implementiert und analysiert. Um diese Algorithmen zu überprüfen, wurde in dieser Forschungsarbeit ein geeigneter Simulator für die Rettungsgassenbildung und Unfallsimulationen mit autonomen Fahrzeugen entwickelt. Die entwickelten Algorithmen wurden sowohl in der Simulation als auch in der Realität, durch einen erfolgreichen Sim-to-Real Transfer, überprüft. Die Bildung der Rettungsgasse konnte sowohl mit den simulierten autonomen Fahrzeugen als auch mit den autonomen Modellfahrzeugen aus dem Modellbaubereich zu jedem Zeitpunkt erfolgreich durchgeführt werden. Ebenfalls konnte, der in dieser Forschungsarbeit entwickelte Filter für den Canny-Algorithmus, sowohl in der Simulation als auch in der Realität für die Spurerkennung eingesetzt werden. Diese Spurerkennung hat auf der verwendeten Hardware mit beschränkten Ressourcen ebenfalls eine Echtzeitauswertung erreicht.

Anschließend wurden zwei weitere sinnvolle Anwendungen für Hardware mit beschränkten Ressourcen entwickelt. Dadurch kann bewusst das Budget so gering wie möglich gehalten werden und dabei ebenfalls der Umwelt zur Liebe Strom und Geld gespart werden. Hierbei handelt es sich um eine Objekterkennung für stromsparende IoT-Geräte mit anschließender bildlichen Entfernungsmessung zu dem erkannten Objekt. Auch diese beiden Vorgehensweisen erreichen eine Echtzeitauswertung auf der verwendeten Hardware mit beschränkten Ressourcen. Diese beiden Vorgehensweisen können anschließend zu einem Radarkontrollsystem in Echtzeit in autonomen Fahrzeugen zusammengefasst werden.

Wie diese Forschungsarbeit zeigt, müssen nicht ausschließlich die künstlichen neuronalen Netzwerke für bestimmte Probleme, wie zum Beispiel die Spurerkennung, verwendet werden. Die traditionellen Methoden können ebenfalls bei den gleichen Problemen vergleichbare Ergebnisse liefern. Hierbei sollte für jedes individuelle Problem der Vergleich dieser Vorgehensweisen aufgestellt werden, da die traditionellen Methoden, zum Beispiel im Vergleich auf die Laufzeit, schneller sein könnten. Dies ist bezogen auf die entwickelte Spurerkennung in dieser Arbeit auch der Fall. Für die Objekterkennung eignen sich am besten die faltenden neuronalen Netzwerke. Dabei können die Netzwerke so verkleinert werden, dass diese ebenfalls eine notwendige Laufzeit bei der Ausführung erreichen. Aus diesem Grund muss für die Entwicklung der Anwendungen für Hardware mit beschränkten Ressourcen ein Gleichgewicht zwischen der Genauigkeit und der Geschwindigkeit gefunden werden, je nach dem was an dieser Stelle wichtiger ist.

Alle in dieser Arbeit durchgeführten Experimente wurden wissenschaftlich analysiert. An dieser Stelle kann gesagt werden, dass alle Ziele dieser Forschungsarbeit erfüllt wurden. Es wurden zusätzlich alle zuvor gestellten Forschungsfragen experimentell belegt und wissenschaftlich beantwortet. Zusätzlich wurden einige ausgewählte Beispiele der Experimente dieser Forschungsarbeit auf der Online-Videoplattform YouTube mit der Angabe wissenschaftlicher Publikationen veröffentlicht [53]. Diese Forschungsarbeit hat ebenfalls gezeigt, dass die Sensoren die wichtigsten Bauteile des autonomen Fahrzeugs sind, wie das Auge und das kontaktlose Empfinden für uns Menschen. Ebenfalls bestätigte

diese Arbeit, dass die Simulation sich von der Realität unterscheidet. Selbst wenn die Simulation sehr stark an den echten Anwendungsfall angepasst ist, enthält die Simulation nicht die Reaktion auf die unerwarteten Hardwarefehler oder Sensorungenauigkeiten, die bei der hardwarenahen Entwicklung entstehen können. Die beiden entwickelten Algorithmen für die Bildung der Rettungsgasse mit autonomen Fahrzeugen sollen dazu beitragen, Menschenleben rechtzeitig zu retten. Dabei ist bei schwerwiegenden Unfällen jede Minute für die Unfallbeteiligten sehr wichtig und entscheidend. Denn die Frage ist nicht ob, sondern wann die autonomen Fahrzeuge auf unsere Straßen kommen.

Literaturverzeichnis

[1] S. S. Al-amri1, N. Kalyankar und S. Khamitkar. “Image Segmentation by Using Threshold Techniques”. In: *Journal of Computing*. Bd. 2. 5. 2010 (siehe S. 54).

[2] R. A. Asmara, B. Syahputro, D. Supriyanto und A. N. Handayani. “Prediction of Traffic Density Using YOLO Object Detection and Implemented in Raspberry Pi 3b + and Intel NCS 2”. In: *Proceedings of the IEEE 4th International Conference on Vocational Education and Training (ICOVET)*. IEEE, 2020. ISBN: 978-1-7281-8131-8. DOI: 10.1109/ICOVET50258.2020.9230145 (siehe S. 90).

[3] A. A. M. Assidiq, O. O. Khalifa, M. R. Islam und S. Khan. “Real time lane detection for autonomous vehicles”. In: *Proceedings of the IEEE International Conference on Computer and Communication Engineering (ICCE)*. IEEE, 2008, S. 82–88. ISBN: 978-1-4244-1691-2. DOI: 10.1109/ICCCE.2008.4580573 (siehe S. 52).

[4] Audiworld. *The Audi Q7 4.2 TDI*. Verfügbar unter: https://www.audiworld.com/articles/the-audi-q7-4-2-tdi. Zugriff am 22.10.2020. 2007 (siehe S. 11).

[5] M. Azmat und C. Schuhmayer. “Future Scenario: Self Driving Cars - The future has already begun”. In: *Innovationsfördernder öffentlicher Beschaffung – 4.0 Platform innovation to e-mobility (Carrie Cockburn/The Globe and Mail - Sources: Google, Articlesbase.com, Wheels.ca)*. 2015, S. 8. DOI: 10.13140/RG.2.1.3143.6965 (siehe S. 21).

[6] M. Behrisch, L. Bieker, J. Erdmann und D. Krajzewicz. “Sumo - Simulation of Urban MObility: an overview”. In: *The Third International Conference on Advances in System Simulation (SIMUL)*. 2011, S. 63–68. ISBN: 978-1-61208-169-4 (siehe S. 31).

[7] J. Beltrán, C. Guindel, F. M. Moreno, D. Cruzado, F. García und A. D. L. Escalera. “BirdNet: A 3D Object Detection Framework from LiDAR Information”. In: *Proceedings of the IEEE 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2018. ISBN: 978-1-7281-0324-2. DOI: 10.1007/978-3-540-88688-4_57 (siehe S. 52, 89).

[8] M. Bonacina. *C# Programmieren für Einsteiger: Der leichte Weg zum C#-Experten*. BMU Media GmbH, 2019, S. 9–12. ISBN: 978-3-96645-001-0 (siehe S. 24).

[9] A. Buchenscheit, F. Schaub, F. Kargl und M. Weber. “A VANET-based emergency vehicle warning system”. In: *Proceedings of the IEEE Vehicular Networking Conference (VNC)*. IEEE, 2009, S. 125–129. ISBN: 978-1-4244-5685-7. DOI: 10.1109/VNC.2009.5416384 (siehe S. 30).

[10] H. M. Bui, M. Lech, E. Cheng, K. Neville und I. S. Burnett. "Using grayscale images for object recognition with convolutional-recursive neural network". In: *Proceedings of the IEEE Sixth International Conference on Communications and Electronics (ICCE)*. 2016, S. 321–325. ISBN: 978-1-5090-1801-7. DOI: 10.1109/CCE.2016.7562656 (siehe S. 66).

[11] Bundesministerium der Justiz und Bundesamt für Justiz. *Straßenverkehrs-Ordnung (StVO): § 18 Autobahnen und Kraftfahrstraßen*. Verfügbar unter: https://www.gesetze-im-internet.de/stvo_2013/__18.html. Zugriff am 10.10.2023. 2013 (siehe S. 44).

[12] Bundesministerium für Klimaschutz, Umwelt, Energie, Mobilität, Innovation und Technologie. *Rettungsgasse*. Verfügbar unter: https://www.oesterreich.gv.at/themen/freizeit_und_strassenverkehr/kfz/10/Seite.063130/Seite.065000.html. Zugriff am 20.10.2023. 2023 (siehe S. 30).

[13] W. Burger, M. J. Burge und W. Burger. *Digitale Bildverarbeitung: Eine algorithmische Einführung mit Java*. Springer, 2015, S. 137–150. ISBN: 978-3-642046-04-9 (siehe S. 12).

[14] W. Burger, M. J. Burge und W. Burger. *Digitale Bildverarbeitung: Eine algorithmische Einführung mit Java*. Springer, 2015, S. 170–183. ISBN: 978-3-642046-04-9 (siehe S. 14).

[15] J. Canny. "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Bd. PAMI-8. 6. 1986, S. 679–698. DOI: 10.1109/TPAMI.1986.4767851 (siehe S. 11).

[16] S. Canu. *Lines detection with Hough Transform – OpenCV 3.4 with python 3 Tutorial 21*. Verfügbar unter: https://pysource.com/2018/03/07/lines-detection-with-hough-transform-opencv-3-4-with-python-3-tutorial-21. Zugriff am 15.09.2020. 2018 (siehe S. 14).

[17] L. Chen. "Road vehicle recognition algorithm in safety assistant driving based on artificial intelligence". In: *Soft Computing*. Springer, 2021, S. 1153–1162. DOI: 10.1007/s00500-021-06011-w (siehe S. 52).

[18] Y. H. Chen. *TensorFlow 2 Detection Model Zoo*. Verfügbar unter: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md. Zugriff am 03.03.2021. 2021 (siehe S. 27).

[19] N. Colella und D. A. Herman. "Emergency corridor utilizing vehicle-to-vehicle communication". US-Pat. US20170352268A1. Ford Global Technologies LLC. 27. Feb. 2018 (siehe S. 32).

[20] J. Dallmeyer und I. J. Timm. "MAINSIM – MultimadAl INnercity SIMulation". In: *Conference on Artificial Intelligence (KI2012)*. 2012, S. 125–129. ISBN: 978-3-642-33347-7 (siehe S. 31).

[21] DATACOM Buchverlag. *Cooperative awareness message (C2C) (CAM)*. Verfügbar unter: https://www.itwissen.info/en/cooperative-awareness-message-C2C-CAM-126127.html. Zugriff am 14.10.2023. 2023 (siehe S. 23).

[22] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez und V. Koltun. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. Bd. 78. PMLR, 2017, S. 1–16 (siehe S. 32).

[23] M. H. Eiza, Q. Ni, T. Owens und G. Min. "Investigation of routing reliability of vehicular ad hoc networks". In: *EURASIP Journal on Wireless Communications and Networking*. Bd. 2013. 179. Springer, 2013 (siehe S. 22).

[24] ELEC Freaks. *Ultrasonic Ranging Module HC - SR04*. Verfügbar unter: `https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf`. Zugriff am 30.11.2022. 2022 (siehe S. 73).

[25] Emil. *Emil's Projects & Reviews: OpenHardware & OpenSource*. Verfügbar unter: `http://uglyduck.vajn.icu/ep/archive/2014/01/Making_a_better_HC_SR04_Echo_Locator.html`. Zugriff am 22.05.2023. 2014 (siehe S. 73).

[26] W. Ertel. *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. Springer, 2016, S. 265–271. ISBN: 978-3-658-13549-2 (siehe S. 6).

[27] W. Ertel. *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. Springer, 2016, S. 291–293. ISBN: 978-3-658-13549-2 (siehe S. 9 f.).

[28] M. Everingham, L. V. Gool, C. K. I. Williams, J. Winn und A. Zisserman. "The PASCAL visual object classes (VOC) challenge". In: *International Journal of Computer Vision (IJCV)*. Bd. 88. 2. Springer, 2010, S. 303–338. DOI: `10.1007/s11263-009-0275-4` (siehe S. 93).

[29] M. Fink, Y. Liu, A. Engstle und S.-A. Schneider. "Deep Learning-Based Multi-scale Multi-object Detection and Classification for Autonomous Driving". In: *Fahrerassistenzsysteme 2018*. Springer, 2019, S. 233–242. ISBN: 978-3-658-23751-6. DOI: `10.1007/978-3-658-23751-6_20` (siehe S. 89).

[30] W. Gieseke. *Raspberry Pi - Schnelle Erfolge erzielen*. Markt+Technik Verlag GmbH, 2015, S. 1–11. ISBN: 978-3-945384-29-9 (siehe S. 28).

[31] Google Colab. *Willkommen bei Colab!* Verfügbar unter: `https://colab.research.google.com`. Zugriff am 24.06.2021. 2021 (siehe S. 29).

[32] A. Gruebler, K. D. McDonald-Maier und K. M. A. Alheeti. "An Intrusion Detection System against Black Hole Attacks on the Communication Network of Self-Driving Cars". In: *Proceedings of the IEEE Sixth International Conference on Emerging Security Technologies (EST)*. IEEE, 2015. ISBN: 978-1-4673-9799-5. DOI: `10.1109/EST.2015.10` (siehe S. 22).

[33] H. Hartenstein und K. P. Laberteaux. "A tutorial survey on vehicular ad hoc networks". In: *IEEE Communications Magazine*. Bd. 46. 6. IEEE, 2008. DOI: `10.1109/MCOM.2008.4539481` (siehe S. 22).

[34] T. Hope, Y. S. Resheff und I. Lieder. *Einführung in TensorFlow: Deep-Learning-Systeme programmieren, trainieren, skalieren und deployen*. O'Reilly, 2018, S. 1–8. ISBN: 978-3-96009-074-8 (siehe S. 6, 27).

[35] T. Hope, Y. S. Resheff und I. Lieder. *Einführung in TensorFlow: Deep-Learning-Systeme programmieren, trainieren, skalieren und deployen*. O'Reilly, 2018, S. 51–59. ISBN: 978-3-96009-074-8 (siehe S. 8).

[36] C.-C. Hsu, M.-C. Lu, W.-Y. Wang und Y.-Y. Lu. "Distance measurement based on pixel variation of CCD images". In: *ISA Transactions*. Bd. 48. 4. ScienceDirect, 2009, S. 389–395. DOI: 10.1016/j.isatra.2009.05.005 (siehe S. 122).

[37] T.-S. Hsu und T.-C. Wang. "Stereo vision images processing for real-time object distance and size measurements". In: *International Journal of Automation and Smart Technology*. Bd. 5. Semanticscholar, 2015, S. 85–90. DOI: 10.5875/AUSMT.V5I2.460 (siehe S. 122).

[38] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama und K. Murphy. "Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017. ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.351 (siehe S. 27).

[39] R. Huang, J. Pedoeem und C. Chen. "YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers". In: *Proceedings of the IEEE International Conference on Big Data (IEEE Big Data 2018)*. IEEE, 2018. ISBN: 978-1-5386-5036-3. DOI: 10.1109/BigData.2018.8621865 (siehe S. 90).

[40] J. Hui. *mAP (mean Average Precision) for Object Detection. COCO mAP*. Verfügbar unter: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173. Zugriff am 10.05.2021. 2018 (siehe S. 16 f., 95).

[41] Intel. *Intel Movidius Myriad X Vision Processing Unit 4GB*. Verfügbar unter: https://www.intel.de/content/www/de/de/products/sku/125926/intel-movidius-myriad-x-vision-processing-unit-4gb/specifications.html. Zugriff am 25.10.2023. 2023 (siehe S. 29).

[42] Intel. *Intel Movidius Vision Processing Units (VPUs)*. Verfügbar unter: https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu.html. Zugriff am 24.06.2021. 2021 (siehe S. 29).

[43] Intel. *Intel Neural Compute Stick 2 (Intel NCS2)*. Verfügbar unter: https://www.intel.com/content/www/us/en/developer/tools/neural-compute-stick/overview.html. Zugriff am 16.11.2022. 2022 (siehe S. 29).

[44] IVsource.net. *Nissan Demos New Lane Keeping Products*. Verfügbar unter: https://web.archive.org/web/20050110073214/http://ivsource.net/archivep/2001/feb/010212_nissandemo.html. Zugriff am 22.10.2020. 2001 (siehe S. 11).

[45] Y. Jin, Y. Wen und J. Liang. "Embedded Real-Time Pedestrian Detection System Using YOLO Optimized by LNN". In: *Proceedings of the IEEE International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*. IEEE, 2020. ISBN: 978-1-7281-7117-3. DOI: 10.1109/ICECCE49384.2020.9179384 (siehe S. 90).

[46] G. Jose, A. Kumar, S. Kruthiventi, S. Saha und H. Muralidhara. "Real-Time Object Detection On Low Power Embedded Platforms". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 2019. ISBN: 978-1-7281-5024-6. DOI: 10.1109/ICCVW.2019.00304 (siehe S. 90).

[47] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar und D. Lange. "Unity: A General Platform for Intelligent Agents". In: *arXiv, Computer Science, Machine Learning*. arXiv, 2020. ISBN: 978-1-5386-0457-1. DOI: 10.48550/arXiv.1809.02627 (siehe S. 26).

[48] G. Kahn, P. Abbeel und S. Levine. "Learning to Navigate from Disengagements". In: *IEEE Robotics and Automation Letters*. Bd. 6. 2. IEEE, 2021, S. 1872–1879. DOI: 10.1109/LRA.2021.3060404 (siehe S. 90).

[49] H. Khodabakhsh und Y. LeCun. *MNIST Dataset: The MNIST database of handwritten digits (http://yann.lecun.com)*. Verfügbar unter: https://www.kaggle.com/datasets/hojjatk/mnist-dataset. Zugriff am 14.10.2023. 2018 (siehe S. 6).

[50] Z. Kim. "Robust Lane Detection and Tracking in Challenging Scenarios". In: *Proceedings of the IEEE Transactions on Intelligent Transportation Systems*. Bd. 9. 1. IEEE, 2008, S. 16–26. DOI: 10.1109/TITS.2007.908582 (siehe S. 52).

[51] M. Kofler, C. Kühnast und C. Scherbeck. *Raspberry Pi: Das umfassende Handbuch*. Galileo Press, 2021, S. 17–44. ISBN: 978-3-8362-8353-3 (siehe S. 28).

[52] R. Kruse, C. Borgelt, C. Braune, F. Klawonn, C. Moewes und M. Steinbrecher. *Computational Intelligence: Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*. Springer, 2015, S. 43–49. ISBN: 978-3-658109-03-5 (siehe S. 5).

[53] J. Kuzmic. *Rettungsgassenbildung mit autonomen Fahrzeugen*. Verfügbar unter: https://www.youtube.com/playlist?list=PLIzKtWM9Wfqen3TxLNnE-noxu980jd_gy. Zugriff am 20.10.2023. 2023 (siehe S. 141).

[54] J. Kuzmic, P. Brinkmann und G. Rudolph. "Real-Time Object Detection with Intel NCS2 on Hardware with Limited Resources for Low-power IoT Devices". In: *Proceedings of the 7th International Conference on Internet of Things, Big Data and Security (IoTBDS)*. INSTICC. SciTePress, 2022, S. 110–118. ISBN: 978-989-758-564-7. DOI: 10.5220/0010979900003194 (siehe S. 138).

[55] J. Kuzmic und G. Rudolph. "Comparison between Filtered Canny Edge Detector and Convolutional Neural Network for Real Time Lane Detection in a Unity 3D Simulator". In: *Proceedings of the 6th International Conference on Internet of Things, Big Data and Security (IoTBDS)*. INSTICC. SciTePress, 2021, S. 148–155. ISBN: 978-989-758-504-3. DOI: 10.5220/0010383701480155 (siehe S. 52, 57, 65, 117, 138).

[56] J. Kuzmic und G. Rudolph. "Object Detection with TensorFlow on Hardware with Limited Resources for Low-power IoT Devices". In: *Proceedings of the 13th International Joint Conference on Computational Intelligence (NCTA)*. INSTICC. SciTePress, 2021, S. 302–309. ISBN: 978-989-758-534-0. DOI: 10.5220/0010653500003063 (siehe S. 138).

[57] J. Kuzmic und G. Rudolph. "Real-time Distance Measurement in a 2D Image on Hardware with Limited Resources for Low-power IoT Devices (Radar Control System)". In: *Proceedings of the 3rd International Conference on Deep Learning Theory and Applications (DeLTA)*. INSTICC. SciTePress, 2022, S. 94–101. ISBN: 978-989-758-584-5. DOI: 10.5220/0011188100003277 (siehe S. 121, 139).

[58] J. Kuzmic und G. Rudolph. "Unity 3D Simulator of Autonomous Motorway Traffic Applied to Emergency Corridor Building". In: *Proceedings of the 5th International Conference on Internet of Things, Big Data and Security (IoTBDS)*. INSTICC. SciTePress, 2020, S. 197–204. ISBN: 978-989-758-426-8. DOI: 10.5220/0009349601970204 (siehe S. 137).

[59] Leftlaneadvisors. *NHTSA Levels of Vehicle Autonomy Infographic*. Verfügbar unter: https://leftlaneadvisors.com/project/nhtsa-levels-of-vehicle-autonomy-infographic. Zugriff am 20.10.2019. 2013 (siehe S. 23).

[60] P. Li, X. Chen und S. Shen. "Stereo R-CNN Based 3D Object Detection for Autonomous Driving". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2019, S. 7644–7652. ISBN: 978-1-7281-3293-8. DOI: 10.1109/CVPR.2019.00783 (siehe S. 52, 89).

[61] K. Lichtenberg, W. McKinney, K. Rother und C. Tismer. *Datenanalyse mit Python: Auswertung von Daten mit Pandas, NumPy und IPython*. O'Reilly, 2019, S. 2–7. ISBN: 978-3-9600-9080-9 (siehe S. 24, 63).

[62] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár und C. L. Zitnick. "Microsoft COCO: common objects in context". In: *European Conference on Computer Vision (ECCV)*. Springer, 2014, S. 740–755. ISBN: 978-3-319-10602-1 (siehe S. 93).

[63] Machine Learning for Artists (ml4a). *THE MNIST DATABASE of handwritten digits*. Verfügbar unter: https://ml4a.github.io/ml4a/looking_inside_neural_nets/. Zugriff am 14.10.2023. 2021 (siehe S. 6).

[64] M. A. Mahammed, A. I. Melhum und F. A. Kochery. "Object Distance Measurement by Stereo VISION". In: *International Journal of Science and Applied Information Technology (IJSAIT)*. Bd. 2. 2. 2013, S. 5–8 (siehe S. 122).

[65] M. D. Malkauthekar. "Analysis of euclidean distance and Manhattan Distance measure in face recognition". In: *Proceedings of the IEEE 3rd International Conference on Computational Intelligence and Information Technology (CIIT 2013)*. IEEE, 2013. ISBN: 978-1-84919-859-2. DOI: 10.1049/cp.2013.2636 (siehe S. 126).

[66] R. T. Marin. "Emergency vehicle proximity warning and communication system". US-Pat. US4794394A. 20. Okt. 1998 (siehe S. 32).

[67] J. C. Marutotamtama und I. Setyawan. "Physical Distancing Detection using YOLO v3 and Bird's Eye View Transform". In: *Proceedings of the IEEE 2nd International Conference on Innovative and Creative Information Technology (ICITech)*. IEEE, 2021. ISBN: 978-1-7281-9748-7. DOI: 10.1109/ICITech50181.2021.9590157 (siehe S. 121).

[68] MATSim. *Agent-Based Transport Simulations*. Verfügbar unter: https://www.matsim.org/about-matsim. Zugriff am 05.04.2019. 2019 (siehe S. 31).

[69] M. Maurer, J. C. Gerdes, B. Lenz und H. Winner. *Autonomes Fahren: Technische, rechtliche und gesellschaftliche Aspekte*. Springer, 2015, S. 1–5. ISBN: 978-3-66245-854-9. DOI: DOI10.1007/978-3-662-45854-9_1 (siehe S. 11, 20).

[70] M. Mitschke und H. Wallentowitz. *Dynamik der Kraftfahrzeuge: Lineares Einspurmodell, objektive Kenngrößen, Subjektivurteile*. Springer, 2014, S. 613–623. ISBN: 978-3-658-05067-2 (siehe S. 21).

[71] Movidius. *TensorFlow Support*. Verfügbar unter: https://movidius.github.io/ncsdk/tensorflow.html. Zugriff am 10.04.2021. 2019 (siehe S. 104).

[72] A. C. Müller und S. Guido. *Einführung in Machine Learning mit Python*. O'Reilly, 2017, S. 1–6. ISBN: 978-3-96010-111-6 (siehe S. 4 f.).

[73] A. C. Müller und S. Guido. *Einführung in Machine Learning mit Python*. O'Reilly, 2017, S. 27–32. ISBN: 978-3-96010-111-6 (siehe S. 7).

[74] Y. M. Mustafah, R. Noor, H. Hasbi und A. W. Azma. "Stereo vision images processing for real-time object distance and size measurements". In: *Proceedings of the International Conference on Computer and Communication Engineering (ICCCE)*. IEEE, 2012, S. 659–663. ISBN: 978-1-4673-0478-8. DOI: 10.1109/ICCCE.2012.6271270 (siehe S. 122).

[75] S. Nedevschi, R. Schmidt, T. Graf, R. Danescu, D. Frentiu, T. Marita, F. Oniga und C. Pocol. "3D lane detection system based on stereovision". In: *Proceedings of the 7th International IEEE Conference on Intelligent Transportation Systems (IEEE Cat. No.04TH8749)*. IEEE, 2004, S. 161–166. ISBN: 0-7803-8500-4. DOI: 10.1109/ITSC.2004.1398890 (siehe S. 52).

[76] J. D. Oliveira. *Jump start your autonomous simulation development with Unity's SimViz Solution Template*. Verfügbar unter: https://blog.unity.com/manufacturing/jump-start-your-autonomous-simulation-development-with-unitys-simviz-solution. Zugriff am 10.04.2019. 2018 (siehe S. 32).

[77] J. D. Oliveira und R. Duong. *AirSim on Unity: Experiment with autonomous vehicle simulation*. Verfügbar unter: https://blog.unity.com/manufacturing/airsim-on-unity-experiment-with-autonomous-vehicle-simulation. Zugriff am 10.04.2019. 2018 (siehe S. 32).

[78] OpenCV. *Canny Edge Detection*. Verfügbar unter: `https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html`. Zugriff am 28.02.2023. 2023 (siehe S. 12 f.).

[79] OpenCV. *Geometric Image Transformations: getPerspectiveTransform()*. Verfügbar unter: `https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html#ga20f62aa3235d869c9956436c870893ae`. Zugriff am 22.02.2023. 2023 (siehe S. 19).

[80] OpenCV. *Geometric Image Transformations: warpPerspective()*. Verfügbar unter: `https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html#gaf73673a7e8e18ec6963e3774e6a94b87`. Zugriff am 22.02.2023. 2023 (siehe S. 19).

[81] OpenCV. *Hough Line Transform*. Verfügbar unter: `https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html`. Zugriff am 28.02.2023. 2023 (siehe S. 14 f.).

[82] OpenVINO. *Converting a TensorFlow Model*. Verfügbar unter: `https://docs.openvinotoolkit.org/latest/openvino_docs_MO_DG_prepare_model_convert_model_Convert_Model_From_TensorFlow.html#Convert_From_TF`. Zugriff am 05.05.2021. 2021 (siehe S. 104).

[83] X. Pan, J. Shi, P. Luo, XiaogangWang und X. Tang. "Spatial As Deep: Spatial CNN for Traffic Scene Understanding". In: *Proceedings of the Thirty-Second AAI Conference on Artificial Intelligence (AAAI)*. Bd. 32. 1. 2018, S. 7276–7283. DOI: `10.1609/aaai.v32i1.12301` (siehe S. 61).

[84] L. Priese. *Computer Vision: Einführung in die Verarbeitung und Analyse digitaler Bilder*. Springer, 2015, S. 1–3. ISBN: 978-3-662451-28-1 (siehe S. 10 f.).

[85] L. Priese. *Computer Vision: Einführung in die Verarbeitung und Analyse digitaler Bilder*. Springer, 2015, S. 204–222. ISBN: 978-3-662451-28-1 (siehe S. 13).

[86] PSA Peugeot Citroën Sustainable Development. *Avoiding accidents*. Verfügbar unter: `https://web.archive.org/web/20051017024140/http://www.developpement-durable.psa.fr/en/realisation.php?niv1=5&niv2=52&niv3=2&id=2708`. Zugriff am 22.10.2020. 2005 (siehe S. 11).

[87] PTV Group. *Traffic Flow Simulation with PTV Vissim*. Verfügbar unter: `https://www.ptvgroup.com/en/solutions/products/ptv-vissim/areas-of-application/traffic-flow-simulation`. Zugriff am 05.04.2019. 2019 (siehe S. 31).

[88] K. A. Rahman, M. S. Hossain, M. A.-A. Bhuiyan, T. Zhang, M. Hasanuzzaman und H. Ueno. "Person to Camera Distance Measurement Based on Eye-Distance". In: *Proceedings of the IEEE 3rd International Conference on Multimedia and Ubiquitous Engineering (MUE 2009)*. IEEE, 2009, S. 137–141. ISBN: 978-0-7695-3658-3. DOI: `10.1109/MUE.2009.34` (siehe S. 122).

[89] Readthedocs.io. *Probabilistic Hough Transform*. Verfügbar unter: `https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_houghlines/py_houghlines.html`. Zugriff am 28.02.2023. 2016 (siehe S. 14 f.).

[90] A. Rosebrock. *Zero-parameter, automatic Canny edge detection with Python and OpenCV*. Verfügbar unter: `https://pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/`. Zugriff am 13.04.2023. 2015 (siehe S. 77 f.).

[91] D. Shah. *Mean Average Precision (mAP) Explained: Everything You Need to Know*. Verfügbar unter: `https://www.v7labs.com/blog/mean-average-precision#:~:text=letÅŽ%20dive%20in!-,What%20is%20Mean%20Average%20Precision%20(mAP)%3F,values%20from%200%20to%201.`. Zugriff am 24.02.2023. 2023 (siehe S. 16 f.).

[92] Y. Shi. *TensorFlow 1 Detection Model Zoo*. Verfügbar unter: `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md`. Zugriff am 23.06.2021. 2020 (siehe S. 27).

[93] M. A. Siegel. "Emergency vehicle proximity warning and communication system". US-Pat. US20030043056A1. 2. Sep. 2003 (siehe S. 32).

[94] A. Spizhevoi und A. Rybnikov. *OpenCV 3 Computer Vision with Python Cookbook*. Packt Publishing, 2018. ISBN: 978-1-788478-75-5 (siehe S. 28).

[95] Stackoverflow. *Why do python programs run very slow the first time?* Verfügbar unter: `https://stackoverflow.com/questions/19684408/why-do-python-programs-run-very-slow-the-first-time`. Zugriff am 06.02.2023. 2022 (siehe S. 65).

[96] SunFounder. *Raspberry Pi Smart Car Kit (Picar-S) for Intermediate*. Verfügbar unter: `https://www.sunfounder.com/products/raspberrypi-sensor-car`. Zugriff am 28.11.2022. 2022 (siehe S. 72).

[97] SunFounder. *SunFounder PiCar-S*. Verfügbar unter: `https://docs.sunfounder.com/_/downloads/picar-s/en/latest/pdf/`. Zugriff am 30.11.2022. 2022 (siehe S. 73).

[98] H. Süße und E. Rodner. *Bildverarbeitung und Objekterkennung: Computer Vision in Industrie und Medizin*. Springer, 2014, S. 515–595. ISBN: 978-3-8348-2606-0 (siehe S. 15).

[99] Swapna K. E. *Convolution Neural Network (CNN)*. Verfügbar unter: `https://developersbreach.com/convolution-neural-network-deep-learning/`. Zugriff am 14.10.2023. 2023 (siehe S. 7).

[100] D. Tan. *A Hands-On Application of Homography: IPM*. Verfügbar unter: `https://towardsdatascience.com/a-hands-on-application-of-homography-ipm-18d9e47c152f`. Zugriff am 22.02.2023. 2020 (siehe S. 18).

[101] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez und V. Vanhoucke. "Sim-To-Real: Learning Agile Locomotion For Quadruped Robots". In: *Proceedings of Robotics: Science and System XIV*. IEEE, 2018. ISBN: 978-0-9923747-4-7 (siehe S. 90).

[102] Tanoshimi. *Is unity magnitude in km/h or m/s?* Verfügbar unter: `https://answers.unity.com/questions/767297/is-unity-magnitude-in-kmh-or-ms.html`. Zugriff am 17.02.2023. 2014 (siehe S. 34).

[103] TensorFlow Core. *TensorFlow Core v2.3.0, API Documentation*. Verfügbar unter: `https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D`. Zugriff am 13.09.2020. 2020 (siehe S. 8).

[104] TensorFlow Core. *TensorFlow Core v2.4.1, API Documentation*. Verfügbar unter: `https://www.tensorflow.org/api_docs`. Zugriff am 07.05.2021. 2021 (siehe S. 114).

[105] TensorFlow Lite. *For Mobile & IoT*. Verfügbar unter: `https://www.tensorflow.org/lite`. Zugriff am 09.05.2021. 2021 (siehe S. 27, 113).

[106] TensorFlow Performance. *For Mobile & IoT, Post-training quantization*. Verfügbar unter: `https://www.tensorflow.org/lite/performance/post_training_quantization`. Zugriff am 09.05.2021. 2021 (siehe S. 114).

[107] Texas Instruments. *Front camera. Automotive front camera integrated circuits and reference designs. Camera SerDes (DS90UB953-Q1)*. Verfügbar unter: `https://www.ti.com/solution/automotive-front-camera`. Zugriff am 09.02.2021. 2021 (siehe S. 67).

[108] T. Theis. *Einstieg in Unity: 2D- und 3D-Spiele entwickeln*. Rheinwerk Verlag, 2021. ISBN: 978-3-8362-8332-8 (siehe S. 25).

[109] K. N. Tripathi und S. C. Sharma. "A trust based model (TBM) to detect rogue nodes in vehicular ad-hoc networks (VANETS)". In: *International Journal of System Assurance Engineering and Management*. Bd. 11. Springer, 2020, S. 426–440 (siehe S. 22).

[110] Tzutalin. *LabelImg*. Verfügbar unter: `https://github.com/tzutalin/labelImg`. Zugriff am 07.05.2021. 2015 (siehe S. 94).

[111] Unity Asset Store. *Car'Toon : The Sport Car with interior*. Verfügbar unter: `https://assetstore.unity.com/packages/3d/vehicles/land/car-toon-the-sport-car-with-interior-62697`. Zugriff am 25.09.2019. 2019 (siehe S. 33 f.).

[112] Unity Asset Store. *Farm Animals Set*. Verfügbar unter: `https://assetstore.unity.com/packages/3d/farm-animals-set-97945`. Zugriff am 25.04.2019. 2017 (siehe S. 34).

[113] S. Urooj, I. Feroz und N. Ahmad. "Systematic literature review on user interfaces of autonomous cars: Liabilities and responsibilities". In: *Proceedings of the IEEE International Conference on Advancements in Computational Sciences (ICACS)*. IEEE, 2018, S. 1–10. ISBN: 978-1-5386-2172-1. DOI: 10.1109/ICACS.2018.8333489 (siehe S. 23).

[114] M. Venkatesh und P. Vijayakumar. "A Simple Bird's Eye View Transformation Technique". In: *International Journal of Scientific and Engineering Research*. Bd. 3. 5. 2012 (siehe S. 18).

[115] S. R. Vuppala. *Getting data annotation format right for object detection tasks*. Verfügbar unter: https://medium.com/analytics-vidhya/getting-data-annotation-format-right-for-object-detection-tasks-f41b07eebbf5. Zugriff am 03.03.2021. 2020 (siehe S. 95).

[116] R. J. Wang, X. Li und C. X. Ling. "Pelee: A Real-Time Object Detection System on Mobile Devices". In: *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS)*. 2018 (siehe S. 90).

[117] Y. Wang, E. K. Teoh und D. Shen. "Lane detection and tracking using B-Snake". In: *Image and Vision Computing*. Bd. 22. 4. ScienceDirect, 2004, S. 269–280. DOI: 10.1016/j.imavis.2003.10.003 (siehe S. 52, 58).

[118] Wikipedia. *Tesla Autopilot, Driving features*. Verfügbar unter: https://en.wikipedia.org/wiki/Tesla_Autopilot#cite_note-:2-70. Zugriff am 05.03.2021. 2021 (siehe S. 16).

[119] C. J. Willmott und K. Matsuura. "Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance". In: Bd. 30. 1. Inter-Research Science Center, 2005, S. 79–82 (siehe S. 9).

[120] W. Xu, J. Wei, J. M. Dolan, H. Zhao und H. Zha. "A real-time motion planner with trajectory optimization for autonomous vehicles". In: *Proceedings of the IEEE International Conference on Robotics and Automation*. IEEE, 2012. ISBN: 978-1-4673-1403-9. DOI: 10.1109/ICRA.2012.6225063 (siehe S. 21).

[121] P. Yadav. *Re: Which is best for image classification, RGB or grayscale?* Verfügbar unter: https://www.researchgate.net/post/Which-is-best-for-image-classification-RGB-or-grayscale/55e962155dbbbd562d8b4591/citation/download. Zugriff am 24.11.2020. 2015 (siehe S. 67).

[122] H. Yu, C. Chen, X. Du, Y. Li, A. Rashwan, L. Hou, P. Jin, F. Yang, F. Liu, J. Kim und J. Li. *TensorFlow Model Garden*. Verfügbar unter: https://github.com/tensorflow/models. Zugriff am 19.05.2021. 2020 (siehe S. 27).

[123] N. Zaghari, M. Fathy, S. M. Jameii und M. Shahverdy. "The improvement in obstacle detection in autonomous vehicles using YOLO non-maximum suppression fuzzy algorithm". In: *The Journal of Supercomputing*. Springer, 2021. DOI: 10.1007/s11227-021-03813-5 (siehe S. 89).

[124] Q. Zhu, L. Wang, Y. Wu und J. Shi. "Contour Context Selection for Object Detection: A Set-to-Set Contour Matching Approach". In: *Forsyth D., Torr P., Zisserman A. (eds) Computer Vision – ECCV 2008. Lecture Notes in Computer Science*. Bd. 5303. Springer, 2008. ISBN: 978-3-540-88685-3. DOI: 10.1007/978-3-540-88688-4_57 (siehe S. 89).

[125] Q. Zou, H. Jiang, Q. Dai, Y. Yue, L. Chen und Q. Wang. "Robust Lane Detection from Continuous Driving Scenes Using Deep Neural Networks". In: *IEEE Transactions on Vehicular Technology*. Bd. 69. 1. IEEE, 2020, S. 41–54. DOI: 10.1109/TVT.2019.2949603 (siehe S. 52).