Rücklichterkennung per Video-Daten

*Eine **Abschlussarbeit** zum **Bachelor of Science**.*

Kaan Dönmez, Freie Universität Berlin, Deutschland

Matrikelnummer: 5335113

kaad01@zedat.fu-berlin.de

05.12, 2021

Betreuer:

Claas-Norman Ritter[1], Freie Universität Berlin, Deutschland

Gutachter:

Prof. Dr. Daniel Goehring[2], Freie Universität Berlin, Deutschland

Prof. Dr. Dr. (h.c.) habil. Raúl Rojas[3], Freie Universität Berlin, Deutschland

Zitation:

Kaan Dönmez, *Rücklichterkennung per Video-Daten*, Freie Universität Berlin, Bachelor Thesis, 2021,
Quelltext: https://git.imp.fu-berlin.de/kaad01/bachelorarbeit_ruecklichterkennungpervideodaten

Version: 1

[1]Dahlem Center for Machine Learning and Robotic, Autonomous Cars
[2]Dahlem Center for Machine Learning and Robotic, Autonomous Cars
[3]Dahlem Center for Machine Learning and Robotic, Autonomous Cars

Eigenständigkeitserklärung

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Berlin (Deutschland), 05.12, 2021

Kaan Dönmez

Inhaltsangabe

1 Abkürzungsverzeicheinis

MIG MadeInGermany

LiDAR Light Detection and Ranging

RADAR Radio Detection and Ranging

GPS Global Positioning System

ROS Robot Operating System

AdaBoost Adaptives Boosting

CNN Convolutional Neural Network

MLP Multilayer Perceptron

LSTM Long short-term memory

ROS Robot Operating System

GPU Graphics Processing Unit

2 Gliederung

In Kapitel 3 wird eine kurze Einführung in die Domäne und Arbeit, sowie Zielsetzung gegeben.

Auf die Konzepte und Begriffe wird in Kapitel 4 eingegangen. Zudem wird diese Arbeit in den Kontext von verwandten Arbeiten gesetzt.

Eine ausführliche Beschreibung des Modells und der Integration in das ROS Framework wird in Kapitel 5 gegeben.

In Kapitel 6 werden die Definitionen der durchzuführenden Experimente beschrieben, dessen Auswertung in Kapitel 7 zu finden sind.

Mit der Diskussion und der Beschreibung von möglichen zukünftigen Arbeiten wird in Kapitel 8 ein Fazit gezogen.

3 Einführung

Das Feld des autonomen Fahrens wird immer wichtiger, vor allem in Deutschland. Im Mai 2021 wurde ein Gesetz verabschiedet, welches autonomen Kraftfahrzeugen die Teilnahme am Verkehr (teilweise) ermöglicht ([3]). Das Potenzial dieser Technologie ist enorm. Man könnte die Anzahl der Unfälle reduzieren. Da 90% der Verkehrsunfälle durch menschliches Versagen hervorgerufen werden, kann man durch das Ersetzen dieser Fehlerquelle die Anzahl der Unfälle erheblich reduzieren. Dazu kommt, dass durch die verstärkte Kommunikation der Fahrzeuge der Verkehr flüssiger und umweltschonender werden würde ([25]). Diese Bachelorarbeit wurde im Rahmen und mit der Hilfe von AutoNOMOS Labs geschrieben. Dies ist ein Projekt der AG für Künstliche Intelligenz der Freien Universität Berlin, welches sich mit der Entwicklung von autonomen Systemen beschäftigt. Darunter fällt auch das Fahrzeug „MadeInGermany“ (kurz MIG).

Das Ziel dieser Arbeit ist die Integration einer virtuelle Rücklichterkennung in das MIG-System.

3.1 MadeInGermany

MIG ist ein modifizierter Volkswagen Passat Variante 3c (siehe [23]). Das Fahrzeug ist mit mehreren Sensoren (Abb. 3.1) ausgestattet.
Darunter fallen unter anderem LiDAR[1]- bzw. RADAR[2]-Sensoren um die Umgebung wahrzunehmen. Mit einem speziellen GPS[3]-System wird die genaue Position des Fahrzeugs ermittelt. Auch werden Kameras für die Aufnahme von Fahrten genutzt. Diese Arbeit nutzt die Videodaten der Fahrten, die das MIG-Fahrzeug auf den vorgeschriebenen Teststrecken aufgenommen hat. Die zugrundeliegende Datenmenge beträgt dabei 11 GB.

[1] Light Detection and Ranging (LiDAR) ist eine Methode zur Abstandmessung mithilfe der Lichtlaufzeitmessung. Dazu werden Laserstrahlen ausgesendet. LiDAR-Sensoren detektieren das zurückgestreute Licht. Aus der Laufzeit des Lichts wird die Entfernung zum Ort der Streuung ermittelt.

[2] Radio Detection and Ranging (RADAR) ist die Bezeichnung für verschiedene Erkennungs- und Ortungsverfahren und -geräte auf der Basis elektromagnetischer Wellen im Radiofrequenzbereich (Funkwellen).

[3] Global Positioning System (GPS) ist ein globales Navigationssatellitensystem zur Positionsbestimmung.

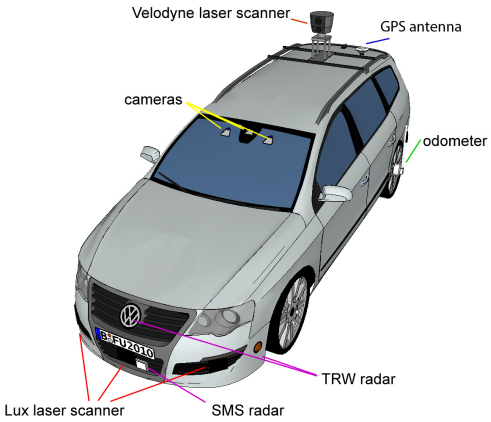

Abbildung 3.1: Sensoren Konfiguration von MIG [23]

3.2 Rücklichterkennung

Rücklichter sind eines der wichtigsten Kommunikationsmittel zwischen Fahrzeugnutzern. Für das autonome Fahren ist die Rücklichterkennung unumgänglich, denn autonome Fahrzeuge müssen sich Straßen mit nicht autonomen Fahrzeugen teilen. Eine brauchbare Rücklichterkennung sollte mit großer Sicherheit auswerten können, ob der Blinker eines Fahrzeuges gesetzt ist oder nicht. Dazu sollte sie auch erkennen, ob die Bremsrücklichter aufleuchten. Die Auswertung der Rücklichter sollte unabhängig von Umweltweinflüssen, wie Nebel, Regen, Schnee usw. sein.
Dies ist mit Hilfe von Machine-Learning Modellen umsetzbar. Diese sogenannten Modelle sind komplexe Funktionen mit sehr vielen Eingabeparametern, welche bei einer passenden Eingabe entsprechende Resultate liefern. Diese Parameter muss man jedoch herausfinden bzw. trainieren.

3.3 Ziel

Das Ziel dieser Arbeit ist es, ein Modell zu trainieren, welches eine Video-Sequenz eines fahrenden Fahrzeuges als Eingabe bekommt und den Zustand der aufleuchtenden Rücklichter zuverlässig zurück gibt. Dieses Modell soll anschließend in das von der Uni gegebene ROS[4] Projekt integriert werden. Die Vorhersagen der Zustände sollen visuell in das Projekt miteingebunden werden, damit auch Menschen die Vorgehensweise verstehen und mitverfolgen können. Ein großer Abschnitt der Arbeit wird dem Experimentieren mit dem Modell bzw. für die Auswertung dieser Experimente gewidmet.

[4] Robot Operating System (ROS) ist ein ist ein Framework für persönliche Roboter und Industrieroboter.

3.3.1 Einschränkungen

Da es große Unterschiede zwischen den vielen Fahrzeuggruppen gibt (Position und Anzahl der Rücklichter) und der genutzte Datensatz [10] nur PKWs beinhaltet, wird sich diese Arbeit auch nur auf PKWs beschränken.

In dieser Arbeit werden detaillierte Betrachtungen nicht-essenzieller Themen unterlassen. Dazu zählen unter anderem LSTMs, CNNs, Detectron2 und SIFT-FLow.

Manche Themen bzw. Begriffe werden vorrausgesetzt und gar nicht erklärt. Darunter fallen unter anderem Begriffe wie Loss-function und One-Hot-Vektor.

4 Grundlagen

In folgendem Kapitel wird diese Arbeit in den Kontext verwandter Arbeiten gesetzt und wichtige Begriffe wie auch Konzepte erklärt.

4.1 Verwandte Arbeiten

Das Problem der Rücklichtererkennung wurde mit den unterschiedlichsten Verfahren versucht zu lösen. Alte Verfahren haben versucht, die Farbtöne und Lichterpositionen zu erkennen und demnach bestimmt, ob ein Fahrzeug bremst oder nicht. ([5], [17] und [15])

Zum Beispiel wurden in „Vehicle Detection at Night Based on Tail Light Detection“ ([18]) die roten und weißen Regionen bzw. RGB Filter erkannt und ausgewertet. Wenn eine weiße Region umgeben von roten Pixeln ist, dann wird es sich wahrscheinlich um ein Rücklicht handeln (siehe Abb. 4.1).

Abbildung 4.1: Ergebnisse aus [18]

Das Paper „Vision Based Method for Forward Vehicle Brake Lights Recognition“’ ([15]) konzentriert sich dabei ganz allein auf die Bremslichter und benutzt einen AdaBoost[1] (Machine-Learning) Ansatz. Mit ca. 80% Genauigkeit ist dieses Modell auch recht zuverlässig, doch noch lange nicht auf dem neusten Stand. Das große Problem ist hierbei, dass die Umweltbedingungen und Lichteinflüsse sich dauernd ändern und teilweise auch sehr ungünstig ausfallen können. Erkennbar ist das am Modell aus dem besagten Paper, welches mit Input-Sequenzen aus sonnigen Tagen besser funktioniert als mit Inputs aus regnerischen Tagen.

Die neueren Verfahren sind zuverlässiger und funktionieren mit Deep-Learning[2] Modellen. Diese Arbeit bezieht sich insbesondere auf „Learning to tell brake and turn signals in videos using CNN-LSTM structure“ ([10]) und „An Attentionbased Recurrent

[1] Adaptives Boosting (AdaBoost) ist ein Algorithmus für das Ensemble-Lernen, der für die Klassifikation oder Regression verwendet werden kann. Obwohl AdaBoost beständiger gegenüber Überanpassung ist als viele Algorithmen des maschinellen Lernens, ist er häufig empfindlich gegenüber verrauschten Daten und Ausreißern.

[2] Deep Learning bezeichnet eine Methode des maschinellen Lernens, die künstliche neuronale Netze mit zahlreichen Zwischenschichten zwischen Eingabeschicht und Ausgabeschicht einsetzt und dadurch eine umfangreiche innere Struktur herausbildet.

Convolutional Network for Vehicle Taillight Recognition"([13]). Im folgenden werden diese Paper kurz zusammengefasst:

4.1.1 Learning to tell brake and turn signals in videos using CNN-LSTM structure

Die Autoren von „Learning to tell brake and turn signals in videos using CNN-LSTM structure" ([10]) haben gezeigt, dass man mit Hilfe einer CNN-LSTM[3] Struktur eine höhere Erkennungsgenauigkeit als mit einem einfachen CNN[4] ereichen kann. Die Autoren benutzen zwei Modelle anstatt nur einem. Eins für die Bremslicht- und eins für die Blinkererkennung.

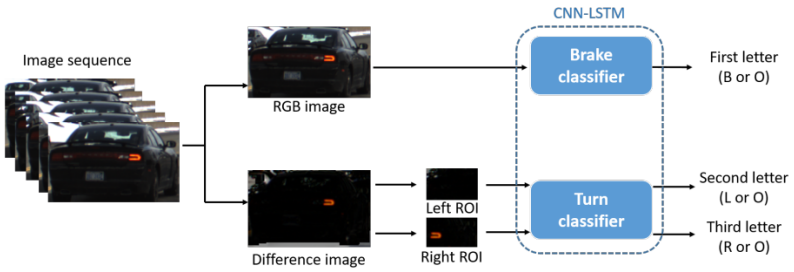

Abbildung 4.2: Modell-Architektur aus [10].

Für das Training der Modelle wurde der sogenannte „UC Merced Vehicle Rear Signal" Datensatz [10] erstellt, welcher im Internet frei zur Verfügung steht. Auch diese Arbeit verwendet diesen Datensatz. Die Autoren vermehren ihren Datensatz per zufälliger Bildbearbeitung (wie zum Beispiel Rotation bzw. Verschiebung). Dadurch erlangt man mehr als dreihundert Mal so viele Bilder. Für die Bremslichterkennung werden einfach nur die Bildsequenzen als RGB Bilder in das Netzwerk eingegeben.

Die Blinkererkennung ist im Vergleich zu der Bremslichterkennung komplizierter, weil eine Abhängigkeit von der Zeit in die Vorhersage mit einfließt. Zuerst werden per SIFT-flow (siehe Kapitel 4.2.4) die abgebildeten Autos übereinander gelegt und dann die Frameunterschiede (frame difference) berechnet. Aus den sich ergebenden Frames werden dann die Positionen der Blinker rausgeschnitten, sodass man zwei kleine Bilder hat auf denen nur die Blinker zu erkennen sind. Diese werden dann dem zweiten Modell übergeben (siehe Abb. 4.2). Dieses Paper legt den Grundstein für das nächste Paper, welches state-of-the-art[5] ist.

[3] CNN-LSTM sind speziell für Sequenz Vohersagen von räumlichen Inputs (Bilder, Videos) entworfen.
[4] Ein Convolutional Neural Network (CNN) ist ein künstliches neuronales Netz, das in der Bilderkennung und -verarbeitung verwendet wird und speziell für die Verarbeitung von Pixeldaten ausgelegt ist. (siehe 4.2.1)
[5] Mit dem Ausdruck state of the art bezeichnet man den aktuellen Entwicklungszustand einer Technologie oder eines Produkts.

4.1.2 An Attention-based Recurrent Convolutional Network for Vehicle Taillight Recognition

Der führende Autor von „An Attentionbased Recurrent Convolutional Network for Vehicle Taillight Recognition"([13]) hat auch an „Learning to tell brake and turn signals in videos using CNN-LSTM structure" ([10]) mitgeschrieben. Anstatt wie in [10] zwei seperate Modelle zu trainieren, konzentrieren sich die Autoren hier auf ein Modell, welches auch auf einem CNN-LSTM basiert. Im Gegensatz zu [10] werden hier die Blinkerpositionen nicht herausgeschnitten. Damit das Modell weiß, worauf es achten soll, wird Attention [24] verwendet. Genauer gesagt Spatial Attention in dem CNN Teil des Modells und temporal Attention in dem LSTM Teil (siehe Abb. 4.3).

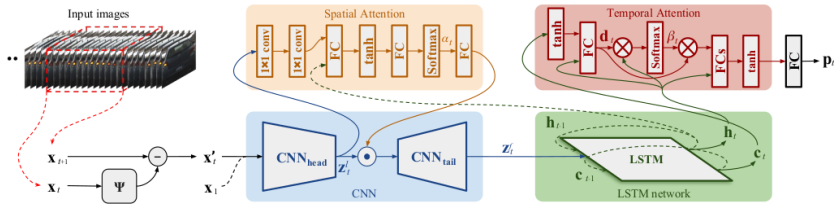

Abbildung 4.3: Modell-Architektur aus [13]

Der CNN Teil basiert auf ResNet50[6]. Das LSTM hat ein hidden-Layer der Größe 256. Die 16 Frames lange Bildsequenz wird auch hier per SIFT-flow übereinander gelegt und es wird der Frameunterschied berechnet, bevor die Daten an das Modell übergeben werden.

In der Tabelle 4.1 sind die Resultate der Tests von den verschiedenen Modellen aufgeführt. Im Vergleich ist zu erkennen, dass nach ausreichendem Testen das Attention-Modell besser abschneidet als das alte CNN-LSTM Modell. Somit ist dieses Modell das neue state-of-the-art für den „UCMerced's Vehicle Rear Signal" Datensatz [10].

Methode	OOO	BOO	OLO	BLO	OOR	BOR	OLR	BLR	Total
CNN-LSTM [10]	97.5	90.4	**95.8**	88.4	95.2	87.0	89.5	92.8	93.0
Attention (S)	96.8	92.9	95	89.2	95.9	87.8	94.5	**100**	93.9
Attention (T)	98.4	95.1	90.8	89.2	97.0	92.4	90.0	**100**	94.8
Attention (S+T)	**98.6**	**95.3**	90.8	93.3	97.0	**97.0**	97.7	98.6	**96.1**

Tabelle 4.1: Vergleich der genannten Modelle. Ergebnisse stammen aus [13]. (S)/(T) bedeutet, das nur spatial/temporal attention genutzt wurde und (S+T) bedeutet, dass beide attention Modelle genutzt wurden. [10].

[6] siehe 4.2.2

4.2 Definitionen

4.2.1 CNN

Convolutional Neural Network wurde von Yann LeCun zum ersten Mal in einem Paper genutzt und bildet einen Grundstein für die heutige Bilderkennung in künstlichen neuronalen Netzwerken [12]. Der Vorteil von CNNs gegenüber normalen MLPs ist, dass CNNs eine niedrigere Wahrscheinlichkeit zum Overfitting haben, weniger Speicher verbrauchen und besser Invarianzen erkennen bzw. ignorieren können. Zudem können CNNs räumliche Abhängigkeiten von Bildern erkennen. Die drei Hauptbestandteile eines CNN sind folgende (siehe Abb. 4.4):

Convolutional Layer: In diesem Teil wird ein Filter über das Bild gesetzt und somit die räumliche Größe verkleinert. Low Level Features (zum Beispiel Kanten oder Kleckse) werden somit erkannt. Wenn man dies mehrfach miteinander verbindet, können auch High-Level Features (wie zum Beispiel Lampen oder Räder) erkannt werden.

Pooling Layer: Dieses Layer ist dazu da, um die räumliche Größe zu verkleinern und somit die nötige Rechenleistung zu senken. Dazu lernt das Modell die dominanten, positions- und rotationsabhängigen Merkmale zu extrahieren.

Fully-connected Layer: In diesem Layer wird die Klasse bestimmt und gibt einen One-Hot Vektor aus.

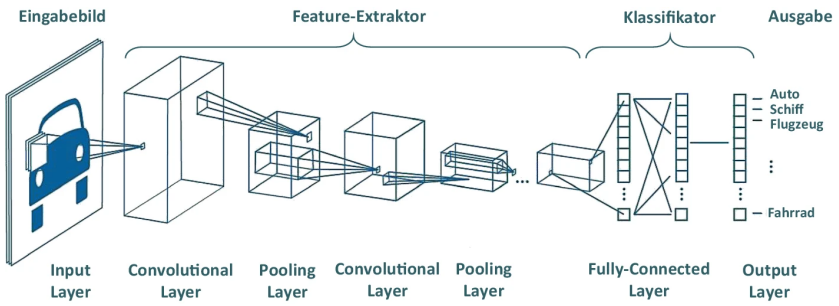

Abbildung 4.4: CNN Architektur [12]

4.2.2 ResNet

CNNs werden mit einer zunehmenden Tiefe (das heißt mehr Layer bzw. mehr Parameter) besser. Dies kommt daher, weil tiefere Netzwerke mehr Parameter haben, die angepasst werden können. Doch nach einer bestimmten Tiefe verbessert sich das Modell nicht mehr. Das liegt an dem sogenannten Vanishing Gradient Problem:

Wenn das Netzwerk zu tief ist, werden durch die Kettenregel so viele Ableitungen (welche kleiner als Eins sind) miteinander multipliziert, sodass die Loss-function auf Null fällt. Somit werden die Gewichte nicht mehr aktualisiert und das Netzwerk kann nicht mehr lernen. ResNet behebt dieses Problem mit Skip Connections (siehe Abb. 4.5).

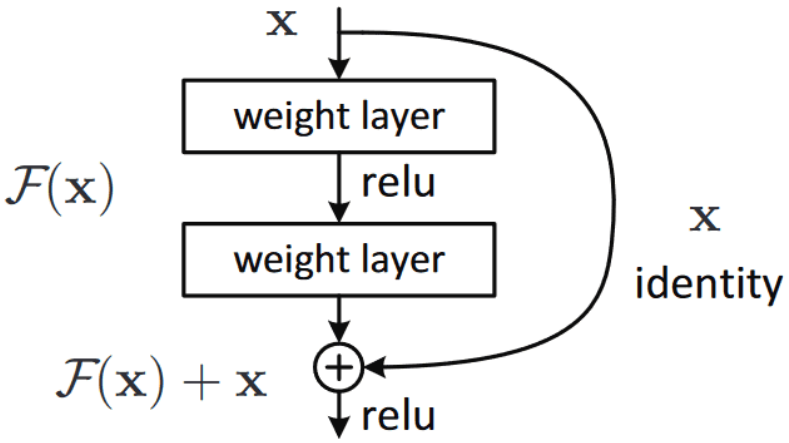

Abbildung 4.5: Skip Connection [7] (mehr dazu in Kapitel 5.3)

Durch das Überspringen von Layern bei der Berechnung werden die Ableitungen beibehalten, wodurch diese nicht zu klein werden.

4.2.3 LSTM

Recurrent Neural Networks (RNN) werden genutzt, um nacheinanderfolgende Sequenzen zu lernen (Video-, Texterkennung). Diese haben jedoch ein Problem: Wenn die Sequenz zu lang wird, werden die Informationen aus älteren Schritten vergessen, da man kein unendlich langes Netzwerk bauen kann, welches alle Schritte speichert. Um dieses Problem zu beheben, werden LSTMs[7] benutzt. Diese bestehen aus sogenannten Gates. Diese Gates lernen welche Informationen wichtig sind und mitgenommen werden bzw. welche unwichtig sind und verworfen werden. In der Abbildung 4.6 ist der Aufbau eines LSTM abgebildet.[8]

[7] Long short-term memory (LSTM) ist ein Verfahren und eine Technik, um die Leistungsfähigkeit rekurrenter neuronaler Netzwerke (RNN) und Künstlicher Intelligenzen zu verbessern.

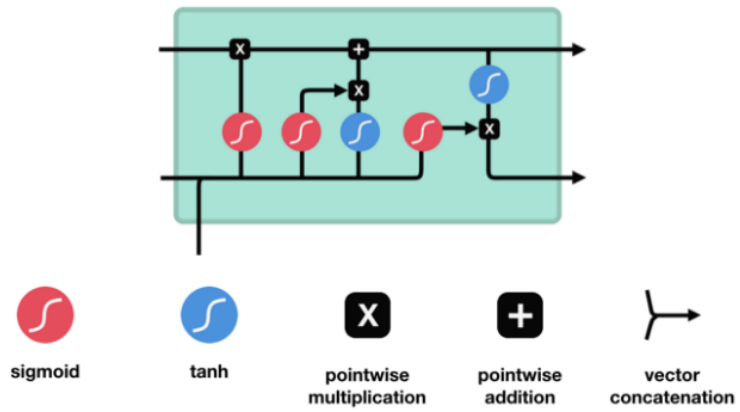

Abbildung 4.6: LSTM Aufbau [19]

4.2.4 SIFT-Flow

SIFT-Flow ist eine Methode für die Ausrichtung von Bildern, um die abgebildeten Objekten übereinander zu legen ([14]). In Abbildung 4.2.4 ist ein Beispiel abgebildet.

Abbildung 4.7: Links sieht man ein Bild von dem ersten Gebäude. Daneben sieht man ein Bild von einem anderen Gebäude. Mithilfe des SIFT-Flow kann man beide Bilder übereinander legen (warpen). Das Ergebnis sieht man am rechten Bild.

4.2.5 SORT

Dieser Algorithmus wird für das Tracking von vielen verschiedenen Objekten genutzt. Durch die Einfachheit der verwendeten Methoden, ist der Algorithmus 20x schneller als andere state-of-the-art Tracker und kann trotzdem mit deren Genauigkeit mithalten. Die Methode ist folgende: Für jedes erkannte Objekt wird eine Tracking-ID erstellt, welche mit einem Array aus Daten (Bounding Box) abgespeichert wird.

$$x = [u, v, s, r, \dot{u}, \dot{v}, \dot{s}]$$

Mit den gespeicherten Bounding Boxes zweier Frames kann man Fahrzeuge ganz einfach verfolgen. Man überprüft einfach ob zwei Objekte in den jeweiligen Frames das selbe Objekt sind. Für das bessere Verständnis soll das nachfolgende Beispiel dienen:

Objekt A wurde in Frame t und Objekt B in Frame t+1 erkannt. Die Bounding Box bzw. State von jedem erkannten Objekt (also auch A) wird bestimmt, indem man mit Hilfe des Kalman Filters[8] die zukünftige Position im Frame t+1 vorhersagt. Beide States (A in t+1 und B in t+1) werden nun miteinander verglichen. Wenn die berechnete IOU (intersection-over-union) größer als ein bestimmter Grenzwert ist, werden A und B als das selbe Objekt klassifiziert. Anschließend kann man mit den neu gewonnen Informationen aus B die State des Objektes mit dem Kalman Filter aktualisieren und die neue State des Objektes im Frame t+2 berechnen. ([2])

4.2.6 Detectron2

Detectron2 ist eine Software-Bibliothek von Facebook AI Research (FAIR) und bietet unter anderem state-of-the-art Objekterkennungsalgorithmen an. Detectron2 ist der Nachfolger von Detectron ([6]) und Mask R-CNN ([16]) und kommt mit einer Menge Verbesserungen. Zum einen lässt es sich schneller trainieren und die Nutzung bzw. Implementation in andere Projekte ist viel einfacher. Das liegt auch daran, dass Detectron2 in PyTorch[9] läuft ([9]).

[8] Das Kalman-Filter ist ein mathematisches Verfahren zur iterativen Schätzung von Parametern zur Beschreibung von Systemzuständen (in unserem Fall Position eines Fahrzeugs).

[9] PyTorch ist eine auf Maschinelles Lernen ausgerichtete Open-Source-Programmbibliothek für die Programmiersprache Python.

5 Implementation

In diesem Kapitel wird das gesamte Modell (Architektur, Training etc.) und dessen Einarbeitung in das von der AG gegebene ROS[1] Framework beschrieben. Programmiert wurde in Python und benutzt wurden folgende Bibliotheken:

- torchvision 0.10.0
- pytorch 1.9.0 py3.8_cuda10.2_cudnn7.6.5_0
- opencv-python 4.5.3.56
- pillow 8.3.1
- numpy 1.20.3
- cudatoolkit 10.2.89
- ROS Noetic

5.1 Datensatz

Für das Training wird der Datensatz aus „Learning to tell brake and turn signals in videos using CNN-LSTM structure“ ([10]) verwendet. Der Datensatz besteht aus einer Ansammlung von .png Dateien, welche mit Hilfe einer Ordnerstruktur sortiert und beschriftet sind. Es wird zwischen acht verschiedene Zuständen unterschieden. Jeder Zustand wird mit den vier Buchstaben „B“ (Bremsen), „L“ (linker Blinker), „R“ (rechter Blinker) und „O“ (OFF) kodiert. Dadruch werden die acht Zustände wie folgt kodiert (siehe Abb. 5.1):

- OOO : Keine Rücklichter leuchten auf
- BOO : Fahrzeug bremst
- OLO : Fahrzeug blinkt nach links
- BLO : Fahrzeug bremst und blinkt nach links
- OOR : Fahrzueg blinkt nach rechts
- BOR : Fahrzeug bremst und blinkt nach rechts
- OLR : Linker und rechter Blinker leuchten auf
- BLR : Alle Rücklichter leuchten auf

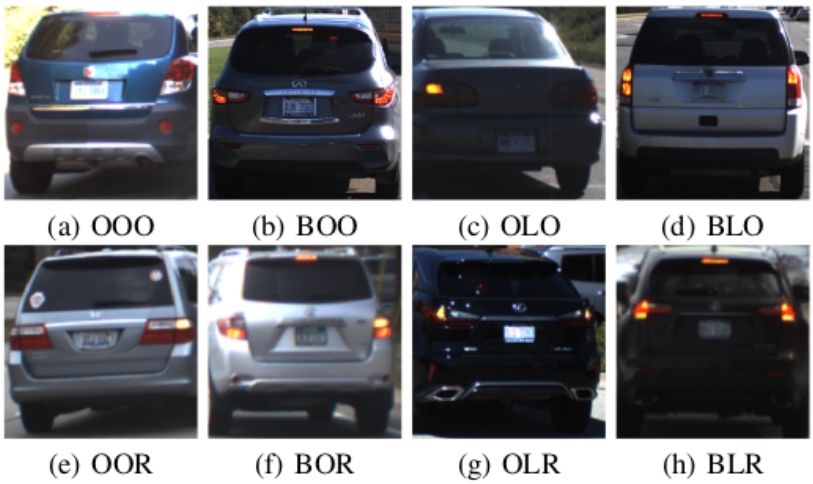

Abbildung 5.1: Beispiele der acht Rücklichterzustände des Datensatzes [10].

Klasse	OOO	BOO	OLO	BLO	OOR	BOR	OLR	BLR
Videos	188	211	78	63	58	33	9	9
Frames	21867	17874	6271	6380	4728	3527	1600	1390

Tabelle 5.1: Verteilung der Zustände im Datensatz [10].

Der genutzte Datensatz enthält 649 Video-Sequenzen und 63637 Frame, deren Verteilung in Tabelle 5.1 aufgeführt ist.

Dem Datensatz ([10]) ist nicht zu entnehmen, mit was für einer Framerate die Frames aufgenommen wurden. Es wird jedoch von 30 Frames pro Sekunde ausgegangen.

5.2 Preprocessing

Um die gegebenen Daten zu nutzen und eine sinnvolle Vorhersage zu bekommen, müssen die Daten zuvor bearbeitet werden. Einerseits muss dies mit den Trainings- und Testdaten geschehen, andererseits auch mit den Daten, die unser Modell anschließend in der Verwendung von dem ROS Framework bekommt.

Angefangen mit dem ersten Frame bilden die nächsten 16 Frames die erste Stichprobe. Die zweite Stichprobe wird mit dem Anfang des zweiten Frames und den nächsten darauffolgenden 16 Frames gebildet. Das geht so weiter, bis für alle Frames eine Stichprobe erstellt wurden.
Für das Lernen des Modells ist es sehr wichtig, dass diese Stichproben keine Lücken aufweisen und zusammengehörig sind.

[1] ROS stellt Bibliotheken und Werkzeuge zur Verfügung, welche bei der Entwickelung von Robotern helfen.

Innerhalb von 16 Frames kann man einen Blinker-Zyklus eindeutig erkennen. Da die benötigte Rechenleistung zum Modelldurchlauf für das spätere Verwenden des Modells in dem ROS Framework so niedrig wie möglich gehalten werden sollte, werden nur alle zwei Frames verwendet[2]. Somit kommt man auf acht Frames. Diese acht Frames sollten theoretisch auch zum Erkennen der Fahrzeugzustände eindeutig sein. In Kapitel 6 wird diese Annahme überprüft und explizit getestet. Unabhängig der Ergebnisfindung aus Kapitel 5 wird im weiteren für diese Arbeit acht Frames genutzt. Dadaurch werden die vielen Trainingsdurchläufe erheblich verschnellert.

Die Daten müssen in eine leicht zu lernende Form überführt werden, damit das Modell optimal lernen kann. Um die zu lernenden Merkmale deutlicher zu erkennen, werden dem Modell nicht einfach die rohen RGB-Daten übergeben, sondern die Frameunterschiede zwischen den Frames berechnet. Dadurch werden die Änderungen eindeutig (z.B. das Aufblinken eines Blinkers).
Da die Frames fahrende Autos abbilden und auch aus einem fahrenden Auto heraus fotografiert wurden, liegen die Positionen der Rücklichter nicht überein. Daher würde der Frameunterschied kein sinnvolles Ergebnis zurückgeben (da sich nicht nur die Blinker ändern). Aus diesem Grund werden die Frames mit Hilfe des SIFT-flow ([14]) übereinander gelegt und anschließend die Framunterschiede berechnet (siehe Abb. 5.2). Der resultieren Input für das Model ist demnach:

$$X = \{X_0, |W_1 - X_0|, |W_2 - X_1|, ...\}$$

Wobei X das Array der Input-Frames ist, X_1 das rohe RGB-Bild für den Zeitpunkt i ist und W_i das gewarpte Bild aus X_{i-1} und X_i ist.

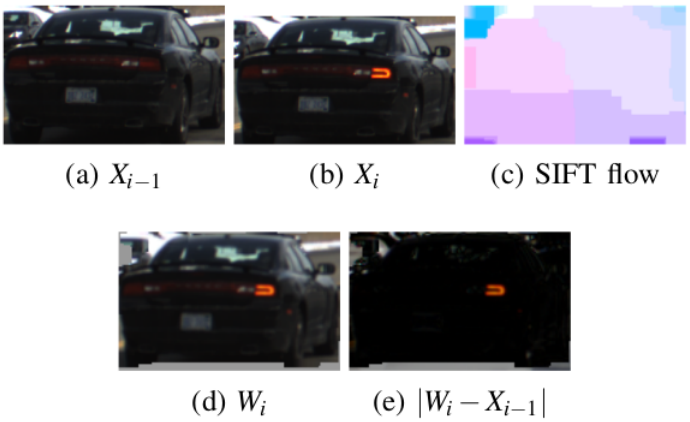

Abbildung 5.2: Frameunterschied zwischen zwei aufeinanderfolgenden Frames [10]. (a) Frame vom vorherigen Schritt. (b) Aktueller Frame. (c) SIFT flow zwischen (a) und (b). (d) Das Resultat, wenn man (b) mit dem SIFT flow aus (c) warped. (e) Betrag der Subtraktion aus (a) und (d).

[2] Siehe Kapitel 5.5 für mehr.

Anschließend werden die Input-Frames transformiert. Zum Einen wird die Größe auf 227 x 227 Pixel geändert und zum Anderen müssen die Bilder in Tensoren[3] transformiert und normalisiert werden, damit die vortrainierten Modelle[4] benutzt werden können. Dies hat den Vorteil, dass das Modell bestimmte Objekte erkennen kann, also ein Idee von High- bzw. Low-Level Features hat (Transfer-learning), wodurch Trainingszeit gespart wird.

Wenn man diese ganzen Preprocessesingschritte erst im Training durchführt, dauert das Training viel zu lange[5]. Da die Daten jedes Mal wenn sie mit dem Dataloader abgerufen werden neu berechnet werden müssen[6]. Dieser statische Prozess kann erheblich verkürzt werden, indem man aus dem gegebenen Datensatz schon einen ”preprocessed”Datensatz erstellt, welcher schon die gewünschte Input-Form aufweist (ohne Normalisation oder Colorbalancing siehe Kapitel 6). Das Training verkürzt sich somit auf eine Stunde pro Epoche (auf dem HPC).

Dieser neue Datensatz ist um den Faktor 16 größer als der Orginale. In „Learning to tell brake and turn signals in videos using CNN-LSTM structure“ ([13]) enthält der benutzte Datensatz 13273 Stichproben. Wie diese Zahl zustande kommt wird nicht erklärt und konnte daher auch nicht reproduziert werden. Der hier genutzte Datensatz besteht aus 53253 Stichproben.

Desweiteren muss der Datensatz [10] ausbalanciert werden[7]. Zum Beispiel ist im Gegensatz zu der Klasse „OOO“ die Anzahl der Frames bzw. Samples von Fahrzeugen welche die Klasse „BLR“ repräsentieren sehr gering. Das führt dazu, dass beim Training des Modells unbalanciert gelernt wird.
Die Erkennung von den Klassen mit geringeren Daten ist schwerer und wenn diese Daten zufällig in Test- und Trainigsdatensätze aufgeteilt werden, kann es dazu kommen, dass manche Klassen im Testdatensatz verschwinden. Daher haben die Autoren von [10] und [13] den Datensatz mit verschiedenen Methoden vermehrt. Diese Methode wurde in dieser Arbeit nicht benutzt, da die genutzte Hardware nicht performant ist, um auf einem derart vergrößerten Datensatz schnell zu trainieren. Für das Training auf dem normalen unveränderten Datensatz werden, je nach Experiment, 10 bis 20 Stunden benötigt.

Ausdiesem Grund wurde der Ansatz, der auch in [10] genutzt wurde, gewählt. Aus jeder Klasse werden genau 90% der Daten für das Training herausgefiltert und die restlichen Daten für das Testen genutzt. Somit wird verhindert, dass alle Frames einer Klasse im Testdatensatz verschwinden und nicht in das Training einfließen.

[3] Ein Tensor ist eine multilineare Abbildung, die eine bestimmte Anzahl von Vektoren auf einen Vektor abbildet.
[4] Damit ist das benutzte ResNet gemeint, welches auf dem Image-Net Datensatz vortrainiert wurde.
[5] auch auf dem HPC
[6] mehr als 10 Tage für 10 Epochen
[7] siehe Tabelle 5.1

5.3 Das Modell

Das Modell basiert auf der Grundlage der Paper „Learning to tell brake and turn signals in videos using CNN-LSTM structure“ ([10]) und „An Attentionbased Recurrent Convolutional Network for Vehicle Taillight Recognition“ ([13]). Da nicht nur die Bremslichter erkannt werden sollen, sondern auch die Blinkersignale, kann nicht einfach nur ein CNN für die Erkennung der räumlichen Eigenschaften benutzt werden.
Blinker leuchten auf, daher ist es schwer diese nur mit Hilfe eines Frames zu erkennen. Um Aktionen zu erkennen wird ein CNN-LSTM benutzt. Dies hat sich in beiden Papern bewährt.
Somit kann dem Modell eine zeitlich gebundene Sequenz als Input gegeben werden. Das CNN erkennt die räumlichen Eigenschaften (Position der Lichter, Farbe, etc.). Der Output des CNN wird an das LSTM weitergegeben, welches daraufhin die zeitlichen Informationen lernt (Aufblinken der Lichter).

Für das CNN wird das vortranierte ResNet50 [7] benutzt, welches eine Variante des ResNet Modells ist (Abb. 5.3) und das state-of-the-art für CNNs bildet.

ResNet50 nimmt standardmäßig die Inputgröße 224x224x3, doch kann man mit dem „resnet50“ aus der Torchvision-Bibliothek jede beliebige Inputgröße verwenden. Daher wird in dieser Arbeit eine Inputgröße von 227x227x3[8] genutzt.
Das Modell wird, wie jedes ResNet Modell, aus 5 Phasen gebildet, wobei die erste Phase immer gleich bleibt.
Zuerst wird eine convolution Operation mit einem 7x7 Kernel (64 Output Channels) und anschließend ein max-pooling mit einem 3x3 Kernel durchgeführt.

Stage 1 (siehe Abb. 5.3) besteht aus drei Residual Blöcken mit jeweils drei Layern. Die benutzten Kernels sind 1x1,64 3x3,64 und 1x1,256 (entnehme Abb.), wobei die Zahl vor dem Komma die Größe des Kernels bestimmt und die Zahl nach dem Komma die Output-Channel Anzahl (Filteranzahl) besagt.

Die gebogenen Pfeile stehen für die „identity connections“ und die gestrichelten Pfeile zeigen an, dass die Convolution in dem Residual Block mit einem stride von 2 berechnet wurde. Das heißt, dass die Anzahl der Channels gedoppelt wird und die anderen Dimensionen halbiert werden (Downsampling). Das ist immer dann nötig, wenn zur nächsten Stage gewechselt wird und somit eine Dimensionsverdoppelung stattfindet. Da bei einem Dimensionsunterschied zwischen dem Output der „identity connection„ und dem „normalen“ Output keine Addition durchgeführt werden kann.

Jeder der Residual Blöcke besteht aus 1x1, 3x3 und 1x1 Convolutions. Die 1x1 Convolutions sind dazu da, um die Dimensionen zu verkleinern und anschließend wiederherzustellen. Die 3x3 Convolutions werden also als Bottlenecks benutzt.

Anschließend besitzt das Modell ein Average Pooling Layer und ein darauf folgendes 1000 Neuronen großes Fully-connected Layer mit softmax.

[8] dieser Wert wurde auch in [10] verwendet

Der CNN Part ist damit komplett. An das Modell wird noch ein LSTM Netzwerk angefügt. Dieses besitzt ein Recurrent Layer mit der hidden size von 256. Zum Schluss werden zwei Fully-connected Layer (256,128 und 128,8) durchlaufen.

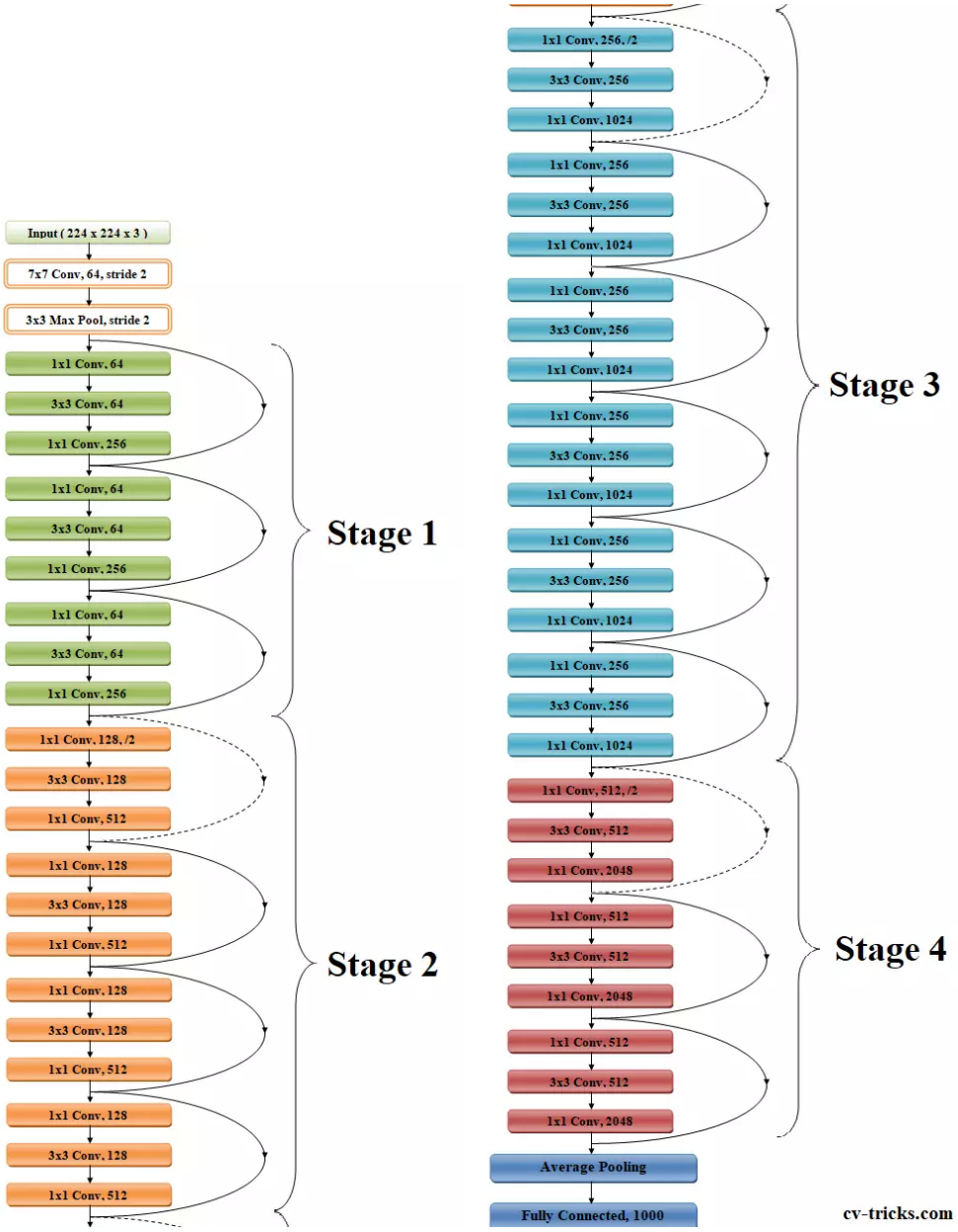

Abbildung 5.3: Aufbau von ResNet50 [21]

5.4 Training

Zum Trainieren des Modells wird wie in [13] eine Batch-Größe von 128 Samples und der SGD-Optimizer zum Lernen genutzt. Anders als in dem Paper wird für dieses Modell der CrossEntropyLoss als Loss-Function benutzt. Die Learning-Rate beträgt 0.1 und das verwendete Momentum beträgt 0.9. Mit diesen Parametern wird in Kapitel 6 experimentiert und die besten Werte herausgefiltert. Doch zum Anfang wurden diese Werte benutzt.

Um das Training zu verschnellern wird CUDA benutzt. Da das Training sehr viel Rechenleistung benötigt, hat diese Arbeit den HPC-Dienst der ZEDAT der Freie Universität Berlin in Anspruch genommen [1].

Um das Training noch weiter zu verschnellern werden zusätzlich für den Dataloader mehrere worker verwendet. Ein Worker ist ein Subprozess, mit dem das Training paralellisiert und somit noch performanter wird. Die optimale Anzahl von Workern ist

$$num\_GPUs \cdot 4$$

Da in dieser Arbeit auf weiteres Coding verzichtet wurde, um die Nutzung von mehreren GPUs zu erreichen, wird nur eine GPU benutzt (Hardware bedingt). Daher werden auch 4 Worker benutzt.

5.5 Implementation in das ROS Framework

Um mit dem fertig trainierten Modell eine sinnvolle Vorhersage zu machen, müssen die Videodaten, welche mit den im Auto installierten Kameras aufgenommen wurden, in die selbe Form transferiert werden wie die des Datensatzes. Das heißt, es kann nur eine Vorhersagen getroffen werden, wenn für ein Auto 16 Bilder vom Rücklicht vorhanden sind. Auch wenn man zum Schluss nur acht Frames benutzt, wird trotzdem eine Sequenz von 16 Frames benötigt.

Dies wäre der Normalfall, jedoch ist die Framerate[9] nicht die gleiche wie die aus den Datensatzaufnahmen[10]. Zwar wird keine explizite Angabe zu der Framrate gemacht doch kann man davon ausgehen, dass eine Standard Kamera genutzt wurde, welche mit 30 Frames pro Sekunde Bilder geschossen hat. Die ROS-Aufnahmen laufen aber mit ca. 10 Frames pro Sekunde. Um diesen Unterschied zu überbrücken werden nicht wie anfangs beschrieben 16 aufeinanderfolgende Bilder gespeichert (aus denen man jedes Zweite entfernt) sondern direkt acht aufeinanderfolgende Bilder gespeichert. Dadurch nähert sich die Framerate der Trainingsstichproben an die der ROS-Aufnahmen an.

[9] 10 Bilder pro Sekunde
[10] wahrscheinlich 30 Bilder pro Sekunde

Um acht Bilder pro Auto zu Speichern muss man zuerst die Autos erkennen. Dafür wird das Objekterkennungs-Modell von Facebook verwendet: Detectron2 [27]. Von ROS wird der derzeitge Frame gegeben, welcher an Detectron2 übergeben wird. Das Modell gibt uns alle erkannten Objekte in Form von Bounding-Boxes wieder. Es werden alle Autos herausgefiltert.

Mit den gegeben Koordinaten der einzelnen Bounding-Boxes können die Bildausschnitte der jeweiligen Autos herausgeschnitten werden.

Somit hat das Programm nun eine Menge Bilderdaten von verschiedenen Autos pro Frame. Doch es werden acht Frames pro Auto benötigt. Das heißt es muss ein Weg gefunden werden, mit dem man jedes der Frame-Ausschnitte einem Auto zuweisen kann. Für das gleichzeitige Tracking mehrerer Fahrzeuge müssen schon zwei aufeinanderfolgende Frames ausreichen. Dazu wird SORT [2] benutzt. Jetzt kann zu jedem Frame jeder Autoausschnitt mit der dazugehörigen Auto-ID in einem Dictionary abgespeichert werden (siehe Quelltext 5.1).

```
1  def update(tracker,imageScene):
2          cars = {} # dictionary of all cars and their saved
       Frames
3          """Get Image"""
4          for x1,y1,x2,y2,car_id in tracker:
5              x1,x2,y1,y2 = int(x1), int(x2), int(y1), int(y2)
6              crop_img = imageScene[y1:y2, x1:x2]
7
8              # append new Frame
9              if car_id in cars:
10                 cars[car_id].append(crop_img)
11             else:
12                 cars[car_id] = [crop_img]
```

Quelltext 5.1: Aktualisieren des Wörterbuchs mit den neuen Frames bzw. Auto IDs

Wenn für eines der Autos acht Frames gespeichert wurden, wandelt das Programm diese so um, sodass das erstellte Modell diese benutzen kann. Dafür werden die Frames erst einmal mit dem SIFT-Algorithmus übereinander gelegt und anschließend in einen Tensor transformiert. Jetzt wird das Modell benutzt um eine Vorhersage aufzustellen.

Für jedes Auto, welches in dem Frame entdeckt wurde, wird die Bounding-Box und dessen Auto-ID gezeichnet, auch wenn es keine Vorhersage gibt. Wenn es eine gibt wird auch die Vorhersage mit auf dem Outputbild angezeigt, damit man die Ergebnisse in dem Robot in rviz[11] mitverfolgen kann (Abb. 5.4).

[11] rviz ist ein 3D-Visualisierungstool für ROS-Anwendungen. Es bietet eine Ansicht des Robotermodells, erfasst Sensorinformationen von Robotersensoren und gibt die erfassten Daten wieder.

Abbildung 5.4: Outputbild aus rviz. An jeder Bounding-Box steht die ID und der Zustand.

Anschließend werden von allen Autos, für die in diesem Frame acht Bilder gespeichert wurden, die ersten Frames gelöscht. Es werden nicht mehr als die letzen acht Frames benötigt, daher ist es unnötig mehr abzuspeichern. Zudem werden noch alle Einträge zu den Autos, die nicht in diesem Frame vorkamen gelöscht. Es können keine zutreffenden Vorhersagen getroffen werden, wenn es zeitliche Lücken zwischen den Input-Frames gibt. Daher werden nur die Frames verwendet, welche zeitlich direkt nacheinander gespeichert wurden. Somit wird der Speicher nicht unnötig belastet.

Eine Herausforderung ist die Verarbeitung der Frames in Echtzeit. Dies ist mit der vorhandenen Hardware nicht möglich. Noch eine Herausforderung ist, dass in dem Zeitraum, in dem das Modell arbeitet, keine weiteren Frames analysiert werden können. Sobald das Programm für ein Auto acht Frames gespeichert hat, startet das Modell und es gehen alle nachkommenden Frames verloren. Dies ist durch mehrere Prozesse lösbar. In dieser Implementation wurde diese Herausforderung jedoch ganz einfach mit der Verlangsamung der Abspielzeit gelöst.

Ausschlaggebend für die Vorhersagen ist auch die Kamera, aus der man die Videodaten bezieht. Da die Kameras an unterschiedlichen Stellen am Fahrzeug montiert wurden, ist der Winkel aus denen die Autos aufgenommen wurden auch unterschiedlich. Um eine gute Vorhersage zu erzielen sollten sich die Input-Bilder aus dem MIG-Fahrzeug denen des Trainings-Datensatzes so stark wie möglich ähneln.

Dies trifft auf die Kamera zu, dessen Daten man aus *'/sensors/hella/image'*[12] beziehen kann.

[12] Sensor Pfad in dem ROS-Projekt

6 Experimente

Um das beste Modell zu generieren, müssen Experimente durchgeführt werden. In diesem Kapitel werden die Experimente vorgestellt. Die vorgestellten Experimente werden nicht nach der Reihe durchgeführt. Einige Experimente sind sinnvoller am Anfang zu beobachten, um aus den gewonnen Informationen die nächsten Experimente zu gestalten.

6.1 Parameter ändern

6.1.1 Learningrate

Da in [10] und [13] wenig Angaben zu den Details der Modelle gegeben werden, müssen die optimalen Parameter mittels Experimenten herausgefunden werden.

Als Erstes wird die Learning-Rate und das Momentum getestet, wobei die Epochenanzahl[1] gleich bleibt.

Dabei wurden folgende Paramter verwendet:

Test	learning rate	momentum
Test 1	0.001	0.9
Test 2	0.01	0.9
Test 3	0.1	0.9

Tabelle 6.1: Parameterexperiment

6.1.2 Epochenanzahl

Ein interessanter Zusammenhang ist die Auswirkung der Trainingsdauer auf die Qualität des Modells. Diese lässt sich mit der Beobachtung der Epochenanzahl im Vergleich zu dem Loss bzw. der Genauigkeit des Modells ermitteln. Wenn man die Epoche findet, ab welcher das Training das Modell nicht verbessert, kann man das Overfitting vorbeugen und auch Zeit sparen, wodurch sich das Training verkürzt.

[1] 10 Epochen

6.2 Input

6.2.1 Anzahl der Frames

Dieser Test bezieht sich auf die Anzahl der Frames, welche für eine Vorhersage verwendet werden. In Kapitel 5.2 wurde angesprochen, dass dieses Modell acht[2] Frames benutzt. In diesem Experiment wird verglichen ob es einen großen Unterschied zwischen acht und 16 Frames gibt (Die Autoren von [10] und [13] benutzen 16 Frames). Die Annahme zu Beginn der Arbeit war die, dass es keinen großen Unterschied macht, da die acht Frames den selben Zeitraum abdecken wie die 16. Durch die Senkung der Input-Frames wurde erhofft, dass eine Vorhersage schneller erstellt wirde als mit den ganzen 16 Frames. Auch diese Annahme wird geprüft indem man die benötigte Zeit für einen Modelldurchlauf miteinander vergleicht.

6.2.2 Größe der Bilder

Das zweite Experiment hierbei bezieht sich auf die Größe der einzelnen Frames. Diese sind in dieser Arbeit 227x227 Pixel groß. Dies wird auf 110x110 heruntergesetzt. Das ist schon eine sehr große Änderung und die Inputgröße nimmt insgesamt ab. Dadurch kann man auch schneller trainieren. Ob die Genauigkeit jedoch mithalten kann wird in diesem Experiment analysiert.

6.3 Colorbalancing

Eines der großen Probleme bei der Rücklichterkennung ist, dass die Umwelteinflüsse die Vorhersage und somit die Genauigkeit verschlechtern kann. Dagegen wird in diesem Experiment versucht mit Colorbalanicng entgegenzuwirken. Der genutzte Quelltext [11] wurde aus dem C++ Projekt [22] in Python importiert.

6.4 Normalisation

Wegen dem genutzten vortrainierten ResNet normalisiert man die Input-Bilder, jedoch ändern sich die Frames dadurch stark (siehe Abb.6.1). Wie man sieht erkennt man nach der Normalisation die Umrisse nicht mehr so gut. Wenn man sich die RGB Input-Frames ansieht erkennt man zusätzlich einen Farbtonunterschied (siehe Abb. 6.2). In diesem Experiment wird die Auswirkung dieses Verhaltens auf das Training bzw. auf die Genauigkeit des Modells analysiert.

[2] jeden zweiten Frame aus insgesamt 16 Frames

Abbildung 6.1: Links ist das nicht normalisierte Bild aus dem „preprocessed“ Datensatz und rechts ist das normalisierte Bild. Das Fehlen von Kanten und Merkmalen ist deutlisch erkennbar.

Abbildung 6.2: Links ist das nicht normalisierte RGB Bild aus dem „preprocessed“ Datensatz und rechts ist das normalisierte Bild. Das normalisierte Bild ist bläulich eingefärbt.

6.5 Modell

6.5.1 ResNet

In diesem Experiment wird der Unterschied beobachtet, wenn man statt dem definierten ResNet50, als Basis für den CNN Teil, das ResNet18 oder ResNet34 nimmt. Wenn der Genauigkeitsverlust nicht zu hoch ist und dafür der Durchlauf des Modells verschnellert werden kann, lohnt es sich vielleicht diesen Ressourcentausch einzugehen.

Diese Überlegung gilt auch für das ResNet101. Wenn die Genauigkeitserhöhung die Geschwindigkeitszunahme bzw. den Speicherzuwachs auswiegt, kann man das Modell austauschen.

7 Auswertung

Für die Auswertung des Modells werden die Genauigkeiten der Erkennung der einzelnen Klassen bzw. Zustände mit einander vergleichen. Dazu wird das jeweilige Modell über den gesamten Datensatz [10] laufen gelassen. Die Auswertung wird in Form von Tabellen dokumentiert. Anschließend werden die Auswertungen der Experimente aus Kapitel 6 in den Kontext der Arbeit gesetzt und diskutiert.

Alle Genauigkeitsauswertungen wurden wie folgt berechnet: Das Modell wurde über den gesamten Datensatz durchlaufen und die Anzahl der korrekt erkannten Klassen durch die Anzahl der gesamten Samples geteilt. Dadurch ergibt sich die Prozentzahl pro Klasse und dadurch wiederum die durchschnittliche Genauigkeit des Modells.

$$Genauigkeit = num\_richtigeVorhersagen/num\_Samples * 100$$

Wichtig zu erwähnen ist, dass die Genauigkeit nur auf die Daten des genutzten Datensatzes bezogen sind. Auf den Daten aus ROS ist diese Genauigkeit deutlich niedriger.

Die Auswertung im ROS-Framwork ist sehr viel schwieriger, da ein Datensatz aus den MIG-Videodaten erst noch erstellt werden müsste. Daher gibt es keine Zahlen mit denen man arbeiten und vergleichen kann. Dennoch werden die Modelle zur intuitiven Auswertung in der besprochenen Implementation durchlaufen und überprüft ob diese sinnvolle Ergebnisse zurückgeben.

7.1 Parameter

7.1.1 Learningrate

Test	OOO	BOO	OLO	BLO	OOR	BOR	OLR	BLR	Total
Test 1	**100**	0	0	0	0	0	0	0	27.4
Test 2	92	96	81	84	66	69	71	66	78
Test 3	83	**98**	**92**	**98**	**75**	**89**	**95**	**99**	**91**

Tabelle 7.1: Auswertung der verschiedenen Modelle

Das erste Modell gibt zu jedem Input den selben Output wieder. Das lässt darauf-schließen, dass etwas mit dem Training nicht in Ordnung ist. Die Learning-Rate von 0.001 ist zu klein. Das Modell lernt zu langsam. Ein besseres Ergebnis wurde mit der Learning-Rate von 0.01 erzeugt. Doch sieht man, dass das beste Ergebnis mit einer Learning-Rate von 0.1 erzeugt wurde. Es ist jedoch zu beachten, dass mit der Learning-Rate von 0.1 das Modell im Laufe der Epochen auch teilweise schlechter

wurde. Dies ist mit der Learning-Rate 0.01 nicht passiert. Das könnte darauf hindeuten, dass das Modell (Learning-Rate = 0.01) nicht zu Ende traniert hat. Daher wurden für die Tests 2 und 3 die Modelle weiter trainiert.

7.1.2 Epochenanzahl

Ausgewertet wurden die Tests „Test 2“ und „Test 3“. Da es interessant zu sehen ist wie sich die learning-rate auf die optimale Trainingsdauer auswirkt.

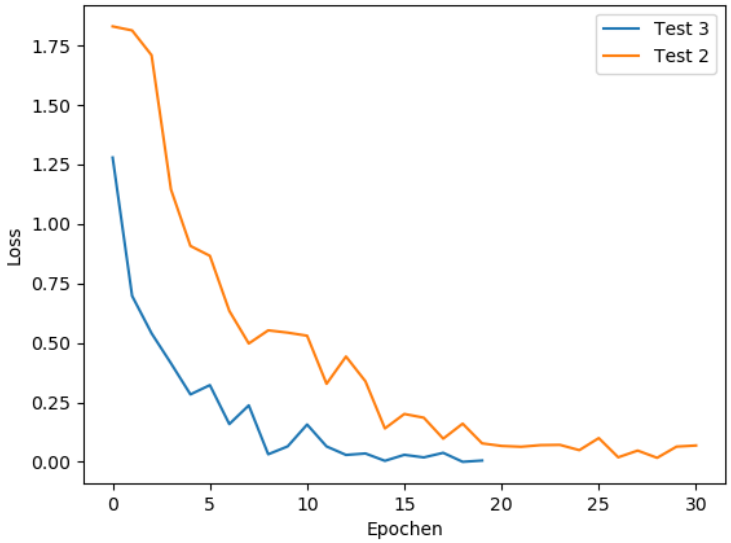

Abbildung 7.1: Zusammenhang zwischen Trainingsdauer und Loss [10].

Für Test 3, also das Modell mit der Learning-Rate 0.1, welche für die weiteren Modelle verwendet wurde, liegt die optimale Epochenanzahl bei ca. 15.

Für Test 2 liegt diese bei ca. 23 Epochen. Denn ab diesen Zeitpunkten lernt das Modell nicht mehr weiter. Dies ist durch den gleichbleibenden Loss-Wert zu erkennen.

7.2 Input

7.2.1 Anzahl der Frames

Das Modell hat mit 16 Input-Frames pro Stichprobe innerhalb 10 Epochen besser abgeschnitten. Für ein optimales Modell würde dieses Modell benutzt werden. Doch ist dieses Modell nicht so kompatibel mit dem ROS-Framework wie das Modell mit den acht Frames als Input. Das liegt an der Aufnahmegeschwindigkeit, welche zuvor besprochen wurde[1]. Das heißt die Inputform aus den ROS-Daten wäre nicht die Gleiche, wie wenn man 16 Frames nehmen würde. Hinzu kommt, dass ein Modelldurchlauf des 16-Frames-Modells doppelt so lange benötigt als das acht-Frames-Modell. Daher wird auf die eigentlich genauere 16 Frames Variante verzichtet. Dies ist vor allem deshalb möglich, weil die acht Frame Variante nicht sehr viel schlechter ist und mithalten kann.

Input-Frames	OOO	BOO	OLO	BLO	OOR	BOR	OLR	BLR	Total
8	83	**98**	92	**98**	75	**89**	95	**99**	91
16	**98**	**98**	**93**	95	**93**	86	**99**	98	**95**

Tabelle 7.2: Anzahl der Input-Frames Experiment-Auswertung

7.2.2 Größe der Bilder

Das Verkleinern der Input-Bilder hat das Modell nur verschlechtert. Die Auswertung ist der Tabelle 7.3 zu entnehmen. Durch das Verkleinern der Input-Bilder verlieren diese an Informationen und die Unterschiede zwischen den Klassen sind schlechter zu erkennen.

OOO	BOO	OLO	BLO	OOR	BOR	OLR	BLR	Total
91	91	84	93	83	75	88	83	86

Tabelle 7.3: Bildgröße Experiment-Auswertung

[1] siehe Kapitel 5.5

7.2.3 Colorbalancing

Das Colorbalancing hat nicht den erwarteten Effekt bewirkt. Das Modell wurde erheblich verschlechtert. Der Grund ist schwer zu bestimmen, doch liegt es wahrscheinlich daran, das durch das Colorbalancing die Unterschiede zwischen den einzelnen Rücklichtern schwerer zu erkennen sind.

OOO	BOO	OLO	BLO	OOR	BOR	OLR	BLR	Total
52	19	1	0	13	0	0	0	11

Tabelle 7.4: Colorbalancing-Experiment Auswertung

Hier lohnt sich ein Test im ROS-Framework nicht.

7.2.4 Normalisation

Beide Modelle (mit und ohne Normalisierung) werden nicht nur über den Datensatz getestet, sondern werden auch in der ROS Implementation ausgetestet. Das heißt für das Modell, welches mit den normalisierten Bildern trainiert wurde, werden die Input-Frames aus ROS auch normalisiert und für das Modell mit den nicht-normalisierten Trainingsdaten werden die Bilder nicht normalisiert.

Die Auswertung der Genauigkeit ist aus Tabelle 7.5 zu entnehmen.

Modell	OOO	BOO	OLO	BLO	OOR	BOR	OLR	BLR	Total
normalisiert	83	**98**	**92**	**98**	75	89	95	**99**	91
nicht normalisiert	**84**	**98**	84	**98**	**86**	**95**	**96**	98	**92**

Tabelle 7.5: Auswertung des Bildgrößen-Experimentes

Die Vermutung war korrekt. Das Normalisieren der Bilder verschlechtert die Performance. Auch wenn sich die Input-Bilder von denen des Image-Net[2] Datensatzes unterscheiden, führt das Normalisieren zu einem zu großen Informationsverlust.

Ein großer Unterschied ist bei der Nutzung des Modells in dem ROS-Framework erkennbar. Die Vorhersagen treffen öfter zu und weisen auf einen ausgeglicheneren Output-Vektor des Modells hin.

[2] Das ist der Datensatz, der für das Vortrainieren der ResNet Modell genutzt wurde. Die Bilder wurden hier normalisiert.

7.3 Modell

7.3.1 ResNet

Die Ergebnisse dieser Experimente sind sehr interessant.

ResNet	OOO	BOO	OLO	BLO	OOR	BOR	OLR	BLR	Total
18	97	89	84	96	77	77	57	74	81
34	**98**	97	83	90	**89**	**90**	85	97	**91**
50	83	**98**	**92**	**98**	75	89	**95**	**99**	**91**
101	97	90	43	75	79	67	36	91	73

Tabelle 7.6: ResNet-Experiment Auswertung

Die „peripheren“ ResNet Modelle schneiden am schlechtesten ab. Das ResNet101 ist sehr groß und hat zu viele Parameter die trainiert werden müssen. Man müsste noch viel länger trainieren um ansatzweise an die Genauigkeit der anderen Modelle heran zu kommen. Obwohl tiefere Modelle besser sind, verspricht sich bei dem ResNet101 kein erkennbarer Genauigkeitszuwachs. Das ResNet18 hingegen ist zu klein, es kann nur wenige Parameter trainieren und anpassen.

Interesannt sind die Modelle ResNet34 und ResNet50. Beide schneiden sehr gut ab jedoch lässt sich kein Unterschied erkennen. Das ResNet 34 schneidet besser mit den R (rechter Blinker) Klassen ab wo hingegen das ResNet 50 besser beim Erkennen der L (linker Blinker) Klassen ist. Der Grund dafür liegt wahrscheinlich beim Training. Jedes Training beinhaltet auch einwenig Glück:
Die Samples die in das Trainingsset kommen, welche Klassen in eine Batch zusammengemischt werden. Dies alles und noch mehr wird durch pseudorandom Verfahren bestimmt. Es ist auszuschließen dass dieses Verhalten etwas mit der Modell-Größe zu tun hat.
Diesem Experiment kann man entnehmen, dass das optimale Modell ein ResNet34 oder ein ResNet50 sein kann. Da für die ROS Implementation auch die Durchlaufzeit des Modells eine Rolle spielt, gilt somit für diese Arbeit das kleinere Resnet34 als optimal.

8 Fazit

In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst und ein optimales Modell erstellt. Anschließend werden Ideen für zukünftige Arbeiten vorgestellt, welche auf dieser Arbeit aufbauen könnten.

8.1 Diskussion der Resultate

Durch die gelernten Fakten, kann man nun das beste Modell herstellen. Dafür werden folgende Eigenschaften genutzt:

- learning-rate = 0.1
- 15 Epochen lang trainieren
- 8 Input-Frames
- Input-Bilder der Größe 227x227 Pixel verwenden
- kein Colorbalancing benutzen
- keine ResNet Normalisation benutzen
- ResNet 34 verwenden

In dieser Arbeit wurde ein Machine-Learning Modell erschaffen, welches auf einem bearbeiteten Datensatz trainiert wurde. Dazu wurden Experimente gestaltet, mit welchen man zu einem Endmodell schließen kann. Das daraus entstandene Modell weist eine Genauigkeit von 92% auf dem orginalen Datensatz [10] aus. Somit erzielt es eine Genauigkeit, welche nur 4% unter dem state-of-the-art[1] liegt, das in [13] erzeugt wurde.

Mit diesem Modell wurde im Rahmen dieser Arbeit eine Implementation für das ROS-Framework erstellt. Mit dieser lassen sich die Zustände der Autos aus dem erzeugten Video-Material bestimmen.

Da die Arbeit und deren Ergebnisse sehr umfangreich sind, gibt es viele Anhaltspunkte auf denen man neue Arbeiten basieren könnte (ein paar werden in Kapitel 8.2 diskutiert).

[1] 96% auf dem Datensatz ([10])

8.2 Future Work

Wie schon zuvor erwähnt, ist ein unbalancierter Datensatz ein Problem. In der Zukunft könnte man das Modell auf einem vermehrten Datensatz trainieren. Dies wurde schon von [10] und [13] gemacht. Es reicht, wenn man die Daten mit zufälligem cropping, horizontalem flipping, gamma correction, rotation und shifting anzureichern. Man muss nicht wie in [10] die Menge der Daten mit dreihundert multiplizieren. Es reicht auch aus die Daten der weniger vorkommenden Klassen zu vermehren, sodass man wenigstens einen wirklich balancierten Datensatz hat. Man könnte stattdessen auch einen weighted Dataloader implementieren, welcher die weniger vorkommenden Klassen mehrmals lädt.

Abgesehen vom Datensatz kann man auch noch das Training bzw. das Modell optimieren. Wenn man mehrere GPUs nutzt, kann man schneller trainieren. Dies kann man auf unterschiedlichen Wegen erreichen. Zum Beispiel gibt es die Möglichkeit Paralellisierung zu nutzen. Das Gute ist, dass es auch hier viele Wege und Ansätze gibt. Das Problem ist es jedoch die entsprechende Hardware für das Entwickeln zu besorgen. Zusätzlich kann man natürlich noch viel mehr Tests entwickeln und das Verhalten analysieren.

Die Implementierung der Attention-Mechanismen aus [13] wurde versucht zu rekonstruieren, doch für ein komplettes Training benötigt man zu viel Zeit und auch zu viele Ressourcen. Man hätte einen kleineren Datensatz zum Testen verwenden können, doch hätte man somit keinen sinnvollen Vergleich zwischen den Modellen erarbeiten können. Daher wurde in dieser Arbeit auf jegliche Attention in dem Modell[2] verzichtet (spatial, temporal).

In dieser Arbeit wurde sich hauptsächlich auf das Modell konzentriert. Daher ist die Implementation in ROS nicht ganz optimal und weist Verbesserungmöglichkeiten auf. Wie zum Beispiel, dass die Abspielrate der Frames nicht in Echtzeit ist. Es gibt Möglichkeiten dies zu erreichen. Das Programm muss nach jedem Frame kurz warten und eine Vorhersage treffen. Dabei werden keine neuen Eingaben mehr angenommen. Dies lässt sich mit einem zweiten Prozess umgehen, welcher einfach alle Frames akzeptiert und buffert. Somit verliert man keine Frames. Zudem kann man die Hardware und die Software anderweitig optimieren, damit man auf eine Echtzeitwiedergabe kommt. Für den praktischen Nutzen ist dies sehr wichtig.

Zudem lassen sich nur für die Autos, welche aus einem relativ kleinen Winkel aus aufgenommen wurden, gute Vorhersagen treffen. Dazu kommt, dass man das vorhandene Modell noch mit direkten Daten von den MIG Aufnahmen nachtrainieren müsste um bessere Vorhersagen zu machen.

Ein größeres Projekt wäre, dass man diese Implementation nicht nur in einer simulierten Autofahrt verwendet sondern in MIG einbaut. Damit MIG in Echtzeit die Zustände der Autos erkennen und vielleicht sogar darauf reagieren kann. Auch ist es möglich

[2] Der Quelltext zu dem Training ist jedoch trotzdem im GitLab Repo hochgeladen

diese Arbeit, welche Rücklichtzustände nur auf Grund von Video-Daten erkennt mit anderen Technologien wie zum Beispiel LiDAR zu verbinden.

9 Referenzen

[1] **Loris Bennett, Bernd Melchers und Boris Proppe**. *Curta: A General-purpose High-Performance Computer at ZEDAT, Freie Universität Berlin*. http://dx.doi.org/10.17169/refubium-26754. 2020.

[2] **Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos und Ben Upcroft**. „Simple online and realtime tracking“. In: *2016 IEEE International Conference on Image Processing (ICIP)*. 2016, S. 3464–3468. DOI: 10.1109/ICIP.2016.7533003.

[3] **BMVI**. *Gesetz zum autonomen Fahren tritt in Kraft*. URL: https://www.bmvi.de/SharedDocs/DE/Artikel/DG/gesetz-zum-autonomen-fahren.html (besucht am 27. 07. 2021).

[4] **Laurent Boucaud**. *CBAM: Convolutional Block Attention Module for CIFAR10 on ResNet backbone with Pytorch*. Okt. 2019. URL: https://github.com/elbuco1/CBAM.

[5] **H. Chen, Yi-Chien Wu und Chun-Chieh Hsu**. „Daytime Preceding Vehicle Brake Light Detection Using Monocular Vision“. In: *IEEE Sensors Journal* 16 (2016), S. 120–131.

[6] **Ross Girshick, Ilija Radosavovic, Georgia Gkioxari, Piotr Dollár und Kaiming He**. *Detectron*. https://github.com/facebookresearch/detectron. 2018.

[7] **Kaiming He, Xiangyu Zhang, Shaoqing Ren und Jian Sun**. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[8] **Sepp Hochreiter und Jürgen Schmidhuber**. „Long Short-Term Memory“. In: *Neural Computation* 9.8 (1997), S. 1735–1780.

[9] **Hiroto Honda**. *Digging into Detectron 2 — part 1*. URL: https://medium.com/@hirotoschwert/digging-into-detectron-2-47b2e794fabd (besucht am 05. 01. 2021).

[10] **Han-Kai Hsu, Yi-Hsuan Tsai, Xue Mei, Kuan-Hui Lee, Naoki Nagasaka, Danil V. Prokhorov und Ming-Hsuan Yang**. „Learning to tell brake and turn signals in videos using CNN-LSTM structure“. In: *IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017.

[11] **David Young-Chan Kay**. *Simple color balance algorithm using Python 2.7.8 and OpenCV 2.4.10.* https://gist.github.com/DavidYKay/9dad6c4ab0d8d7dbf3dc. Accessed: [02.11.2021]. 2017.

[12] **Yann Lecun und Yoshua Bengio**. „Convolutional Networks for Images, Speech and Time Series“. In: *The Handbook of Brain Theory and Neural Networks*. Hrsg. von Michael A. Arbib. The MIT Press, 1995, S. 255–258.

[13] **Kuan-Hui Lee, Takaaki Tagawa, Jia-En M. Pan, Adrien Gaidon und Bertrand Douillard**. *An Attention-based Recurrent Convolutional Network for Vehicle Taillight Recognition*. 2019. arXiv: 1906.03683 [cs.CV].

[14] **Ce Liu, Jenny Yuen und Antonio Torralba**. „SIFT Flow: Dense Correspondence across Scenes and Its Applications.“ In: *IEEE Trans. Pattern Anal. Mach. Intell.* 33.5 (2011), S. 978–994. URL: http://dblp.uni-trier.de/db/journals/pami/pami33.html#LiuYT11.

[15] **Wei Liu, Hong Bao, Jun Zhang und Cheng Xu**. „Vision-Based Method for Forward Vehicle Brake Lights Recognition“. In: *International Journal of Signal Processing, Image Processing and Pattern Recognition* 8 (Juni 2015), S. 167–180. DOI: 10.14257/ijsip.2015.8.6.18.

[16] **Francisco Massa und Ross Girshick**. *maskrcnn-benchmark: Fast, modular reference implementation of Instance Segmentation and Object Detection algorithms in PyTorch*. https://github.com/facebookresearch/maskrcnn-benchmark. Accessed: [02.11.2021. 2018.

[17] **Dario Nava, G. Panzani und S. Savaresi**. „A Collision Warning Oriented Brake Lights Detection and Classification Algorithm Based on a Mono Camera Sensor“. In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)* (2019), S. 319–324.

[18] **Ronan O'Malley, Martin Glavin und Edward Jones**. „Vehicle Detection at Night Based on Tail-Light Detection“. In: 2008.

[19] **Michael Phi**. *Illustrated Guide to LSTM's and GRU's: A step by step explanation*. URL: https://towardsdatascience.com/illustrated-guide-to-lstms-and-grus-a-step-by-step-explanation-44e9eb85bf21 (besucht am 19. 10. 2021).

[20] **Pablo Ruiz**. *Understanding and visualizing ResNets*. URL: https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8 (besucht am 19. 10. 2021).

[21] **Ankit Sachan**. *Detailed Guide to Understand and Implement ResNets*. URL: https://cv-tricks.com/keras/understand-implement-resnets/ (besucht am 16. 10. 2021).

[22] **Roy Shilkrot**. *Simplest Color Balance with OpenCV [w/code]*. URL: https://www.morethantechnical.com/2015/01/14/simplest-color-balance-with-opencv-wcode/ (besucht am 02. 11. 2021).

[23] **AutoNOMOS team**. *MadeInGermany*. URL: https://autonomos.inf.fu-berlin.de/vehicles/made-in-germany/ (besucht am 26. 10. 2021).

[24] **Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser und Illia Polosukhin**. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[25] **Thomas Kroher (ADAC-Redakteure) Wolfgang Rudschies**. *Autonomes Fahren: So fahren wir in Zukunft*. URL: https://www.adac.de/rund-ums-fahrzeug/ausstattung-technik-zubehoer/autonomes-fahren/technik-vernetzung/aktuelle-technik/ (besucht am 08. 09. 2021).

[26] **Sanghyun Woo, Jongchan Park, Joon-Young Lee und In So Kweon**. *CBAM: Convolutional Block Attention Module*. 2018. arXiv: 1807.06521 [cs.CV].

[27] **Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo und Ross Girshick**. *Detectron2*. https://github.com/facebookresearch/detectron2. 2019.