

Erkennen von Fahrradfahrern auf Kamerabildern mit einem künstlichen neuronalen Netzwerk

Alexander Wyron Wachtberger

15. Mai 2019
Version: Abgabe



Fachbereich VI
Informatik und Medien
Technische Informatik - Embedded Systems

Masterarbeit
zur Erlangung des akademischen Grades
Master of Engineering (M.Eng.)

Erkennen von Fahrradfahrern auf Kamerabildern mit einem künstlichen neuronalen Netzwerk

Alexander Wyron Wachtberger

Gutachter Herr Prof. Dr. Biessmann
 Fachbereich VI - Informatik und Medien
 Beuth Hochschule für Technik

Betreuer Herr Prof. Dr. Macos
 Fachbereich VI - Informatik und Medien
 Beuth Hochschule für Technik

15. Mai 2019

Alexander Wyron Wachtberger

Erkennen von Fahrradfahrern auf Kamerabildern mit einem künstlichen neuronalen Netzwerk

Masterarbeit, 15. Mai 2019

Gutachter: Herr Prof. Dr. Biessmann

Betreuer: Herr Prof. Dr. Macos

Beuth Hochschule für Technik

Technische Informatik - Embedded Systems

Informatik und Medien

Fachbereich VI

Luxemburger Straße

13353 Berlin

Danksagung

Mein Dank geht in erster Linie an meine Freundin Katrin Köhntopp. Insbesondere in den letzten Zügen hast du mir geholfen ruhig zu bleiben und einen klaren Kopf zu bewahren. Außerdem hast du dich darum gekümmert, dass ich wenn ich schon nicht genug schlafe zumindest gut und genug esse. Danke!

Mein Dank geht auch an meinen Firmenbetreuer Lars Schürmann, ohne ihn wäre das Thema nicht entstanden und er hat mir stets genügend Freiraum und Feedback gegeben, um ans Ziel zu kommen.

Abschließend möchte ich all den Leuten danken, die mir Ideen und Hinweise gegeben haben, Korrektur gelesen und Feedback gegeben haben.

Zusammenfassung

Diese Arbeit befasst sich mit künstlichen neuronalen Netzwerken, genauer gesagt mit Objekt Detektoren. Im Rahmen dieser Arbeit wird einem Objekt Detektor per Transfer Learning beigebracht, zuvor unbekannte Objekte zu erkennen. Die Objekte stellen im Rahmen dieser Arbeit Fahrradfahrer dar. Für die Umsetzung wird zunächst ein Datensatz zusammengestellt, sowie die richtigen Trainingsparameter identifiziert. Der Objekt Detektor wird auf Genauigkeit geprüft und im eingebetteten System auf Geschwindigkeit getestet. Der Test im eingebetteten System wird u.a. auf dem NVIDIA Jetson AGX Xavier durchgeführt. Die Testergebnisse werden evaluiert und die Frage beantwortet, wie zuverlässig und schnell Fahrradfahrer auf Kamerabildern mit einem künstlichen neuronalen Netzwerk detektiert werden können. Die Arbeit schließt ab mit der Identifikation von Schwächen im System sowie Handlungsansätzen für Folgearbeiten.

Abstract

This thesis deals with artificial neural networks, or more specifically object detectors. By transfer learning, an object detector is taught to recognize previously unknown objects. For the purpose of this work, the objects depict cyclists. First, a data set is compiled and the correct training parameters are found. The object detector is checked for accuracy and tested for speed in an embedded system. Amongst others, the test in the embedded system will be performed on the NVIDIA Jetson AGX Xavier. The test results are evaluated and the question of how reliable and fast cyclists can be detected on camera images with an artificial neural network is answered. Finally, limitations of this work are identified and recommendations for future work is given.

Inhaltsverzeichnis

Abstract	vii
Abbildungsverzeichnis	xi
Tabellenverzeichnis	xiii
Quellcodeverzeichnis	xv
Abkürzungsverzeichnis	xvii
1 Einleitung	1
1.1 Fragestellung	3
1.2 Aufbau der Arbeit	3
2 Fachliche Grundlagen	5
2.1 Maschinelles Lernen	6
2.1.1 Überwachtes Lernen	6
2.1.2 Gradientenverfahren	8
2.1.3 Das Training	9
2.2 Künstliche neuronale Netzwerke	11
2.2.1 Struktur künstlicher neuronaler Netzwerke	12
2.2.2 Fehlerrückführung	15
2.3 Faltende neuronale Netzwerke	15
2.4 Objekt Detektoren	17
3 Aufgabenstellung	19
3.1 Ziel der Arbeit	19
3.2 Abgrenzung	20
3.3 Anforderungsanalyse mit Anwendungsfällen	21
3.3.1 Akteure	21
3.3.2 Szenarien	21
3.3.3 Anwendungsfälle	24
3.3.4 Use Case Diagramm	28
3.4 Anforderungen	29
3.4.1 Vorgehensweise	29
3.4.2 Anforderungskatalog	30

4 Lösungsansatz	33
4.1 Das verwendete künstliche neuronale Netzwerk	34
4.1.1 Darknet Framework	37
4.2 Datenbasis	38
4.2.1 Tsinghua-Daimler Cyclist Benchmark	38
4.2.2 Selbst aufgenommene Videos und YouTube Videos	39
4.2.3 Datensatz Bereinigung	40
4.3 Training	41
4.3.1 Trainingsdaten	41
4.3.2 Trainingsverlauf	47
4.3.3 Trainingsende	48
4.4 Hardware	49
5 Systementwurf	51
5.1 Systemarchitektur	51
5.1.1 Gesamtsystem	51
5.1.2 Softwarekomponenten	53
5.2 Dynamisches Verhalten	57
6 Evaluierung	63
6.1 Geschwindigkeitstest	63
6.2 Genauigkeitstest	66
6.2.1 Fehleranalyse	68
6.3 Auswertung	71
6.4 Limitationen und Handlungsempfehlungen	74
7 Zusammenfassung	79
Literaturverzeichnis	81
Webseiten	85
A Tabellen	87
B Quellcode Auszüge	91
C Sonstiges	115
C.1 Inhalt der DVD	115
C.2 Installationsanleitung für das FFDS	116

Abbildungsverzeichnis

2.1	Abbildung einer Funktion mit mehreren lokalen Minima	8
2.2	Gegenüberstellung eines biologischen Neurons zu einem künstlichen Neuron	11
2.3	Eingabeschicht mit zwei Werten und Ausgabeschicht mit einem Neuron	13
2.4	XOR-Funktion mit einem künstlichen neuronalen Netzwerk	14
2.5	Unterschiedlich komplexe Merkmale in einem Bild	16
2.6	Gegenüberstellung von Begrenzungsrahmen und Segmentation	17
3.1	Anwendungsfälle für das Fahrradfahrer Detektionssystem	28
4.1	YOLOv3 Netzwerk Architektur	36
4.2	Beispielbild eines nicht markierten Fahrradfahrers	39
4.3	Durchschnittlicher Fehler über die Anzahl der Iterationen	48
4.4	Beispielausgabe der berechneten Genauigkeitswerte	49
5.1	VeloxCar mit eingebettetem Fahrradfahrer Detektionssystem	52
5.2	Softwarekomponenten des Fahrradfahrer Detektionssystems	53
5.3	Die Klassen und Methoden des Fahrradfahrer Detektionssystem	55
5.4	Vollständiger Programmablauf des Fahrradfahrer Detektionssystem	58
5.5	Sequenz vom Laden des Netzwerks bis zur Veröffentlichung	61
6.1	Falsch berechnete Begrenzungsrahmen aufgeteilt nach Fehlerkategorien	69
6.2	Nicht erkannte Fahrradfahrer aufgeteilt nach Fehlerkategorien	70
6.3	Beispielbilder falscher Detektionen	76
6.4	Beispielbilder fehlender Detektionen	77

Tabellenverzeichnis

2.1	Wahrheitstabelle für die logische XOR-Funktion mit zwei Variablen	13
2.2	Ausgabewerte der Knoten des XOR-Netzwerks	15
3.1	Szenario 1: Start und Stopp des FFDS	22
3.2	Szenario 2: Detektionsablauf	23
3.3	Szenario 3: Erhalten von Ergebnisdaten	23
3.4	Anwendungsfall: FFDS starten	25
3.5	Anwendungsfall: Fahrradfahrer detektieren	26
3.6	Anwendungsfall: Detektionen beobachten	27
4.1	Vergleich der Genauigkeit und Schnelligkeit aktueller Objekt Detektoren	34
4.2	Datensatz-Statistik des Tsinghua-Daimler Cyclist Benchmark	39
4.3	Anzahl der automatisch generierten Daten	40
4.4	Auflistung und Erläuterung der Trainingsvariablen in Darknet	43
4.5	Verwendete Werte der Trainingsvariablen	46
6.1	Ergebnisse der Geschwindigkeitstests für YOLOv3	65
6.2	Ergebnisse der Geschwindigkeitstest für das YOLOv3-tiny Netzwerk	65
6.3	Auszug der Ergebnisse des Genauigkeitstests für YOLOv3	67
6.4	Auszug der Ergebnisse des Genauigkeitstests für YOLOv3-tiny	68
6.5	Alternative Berechnung der Genauigkeit	72
A.1	Die verwendeten Systeme der Geschwindigkeitstests	87
A.2	Ergebnisse der Geschwindigkeitstests für YOLOv3	88
A.3	Ergebnisse der Geschwindigkeitstests für YOLOv3-tiny	89

Quellcodeverzeichnis

B.1	Java Funktion zum Parsen von Markierungsdateien aus dem Tsinghua-Daimler-Datensatz	91
B.2	C Funktion zur Datensatzerstellung mit Darknet	94
B.3	Bash Skript zur Überprüfung des aktuellen Fehlers	96
B.4	Shell Skript zur Überprüfung des Trainings	97
B.5	Python Skript zum Plotten des Fehlers	98
B.6	Java Programm zum Auswerten der Trainings- und Testergebnisse	100
B.7	Implementation der Schnittstelle Image_Provider als ROS Node	106
B.8	Implementation der Schnittstelle Publish_Detections als ROS Node	108
B.9	Implementation des FFDS in C	109
B.10	C Funktion zum Speichern von Bildern mit Begrenzungsrahmen	111
B.11	Auszug aus einer Konfigurationsdatei für Darknet	113

Abkürzungsverzeichnis

KNN Künstliches neuronales Netzwerk

SGV Stochastisches Gradientenverfahren

DNN Deep Neural Network

CNN Convolutional Neural Network

SMART Specific Measurable Achievable Reasonable Time Bound

FFDS Fahrradfahrer Detektionssystem

ADAS Advanced Driver Assistance Systems

FPN Feature Pyramid Network

CUDA Compute Unified Device Architecture

cuDNN CUDA Deep NeuralNetwork

ap Average precision

IoU Intersection over Union

ROS Robot Operating System

NMS Non Maximum Suppression

FPS Frames per Second

TP True Positive

TN True Negative

FP False Positive

FN False Negative

PPV Positive Predictive Value

TPR True positive rate

Einleitung

Automobilhersteller prognostizieren die ersten fahrerlosen Fahrzeuge für das Jahr 2030. Fahrerlos bedeutet, dass das Fahrzeug ein Ziel erhält, autonom den Weg dorthin findet und schließlich auch parkt. Solche Fahrzeuge werden keine Lenkräder oder Gaspedale besitzen und menschliche Insassen sind nur Passagiere. Damit ein Auto ohne Fahrer vom Startpunkt bis zum Zielpunkt gelangt, benötigt es Wissen. Angefangen bei „wo befindet sich die Straße bzw. die korrekte Fahrbahn?“, über „welche Verkehrsregeln gelten aktuell?“ bis hin zu „wann bremse ich wie stark ab?“ und „wo ist ein Parkplatz?“. Um die genannten Fragen zu beantworten, können verschiedenste Methoden und Technologien gewählt werden. Die Verwendung einer Kamera, welche fortlaufend Bilder der aktuellen Szene aufnimmt, ist eine Möglichkeit, Informationen zu sammeln. Im Anschluss müssen diese Informationen eingeordnet und ausgewertet werden, um komplexe Verkehrsszenarien autonom lösen zu können. Verkehrsszenarien sind allerdings so vielfältig, dass sich nicht für jedes Szenario eine Lösung bereitstellen lässt. Vielmehr müssen aus dem Kontext heraus Entscheidungen getroffen werden – so wie es auch der Mensch im Verkehr tut. Dies ist nur auf Basis vollständiger Umgebungsdaten möglich um darauf beruhend ein Verständnis über das aktuelle Verkehrsszenario zu erhalten, wie Haist beschreibt [Hai16]. Ein wesentlicher Teil davon liegt in der Erkennung und Klassifizierung von Objekten im aktuellen Verkehrsszenario.

Der Verband der Automobilindustrie hat die Automatisierungsgrade des automatisierten Fahrens in sechs Stufen klassifiziert [Aut15]. Seit Jahren findet die Stufe eins Gebrauch, das assistierte Fahren. Sehr wenige Serienautos bieten teilautomatisiertes Fahren an, bei denen der Fahrer das System dauerhaft überwachen muss. Stufe drei beschreibt das hochautomatisierte Fahren, bei dem der Fahrer das System nicht mehr dauerhaft überwachen, aber gegebenenfalls die Steuerung übernehmen muss. Hochautomatisiertes Fahren ist kein Zukunftsszenario mehr, sondern bereits Realität: Insbesondere der Automobilhersteller Audi AG bietet mit seinem Audi A8 ein gutes Beispiel – laut Hersteller das erste Serienauto auf der dritten Automatisierungsstufe, allerdings gilt der dringende Hinweis, das autonome Fahrsystem nur auf Autobahnen und baulich abgetrennten Straßen zu verwenden. Das System nennt sich AI Staupilot und ist derzeit in Europa noch nicht gesetzlich zugelassen. Ein weiterer Hersteller, welcher sich dem Thema automatisiertes Fahren angenommen hat, ist Tesla. Tesla bietet ein Fahrerassistenzsystem namens Autopilot, allerdings handelt es sich nur

um ein System der Stufe zwei, welches ständig überwacht werden muss. Weder Fahrradfahrer noch Fahrräder werden von diesem System klassifiziert oder erkannt. Weiter gibt es vereinzelt zugelassene autonome Fahrzeuge anderer Hersteller, wie das Googlecar oder von der Firma Uber nachgerüstete Serienautos. Grundsätzlich ist hochautomatisiertes Fahren also ein sehr präsentes und aktuelles Thema in der Forschung unter Automobilherstellern, gleichermaßen etabliert sich das Thema auch immer mehr in der Gesellschaft. Ein wichtiger Schwerpunkt in der aktuellen Automatisierungs-Forschung liegt auf dem Erkennen und Klassifizieren aller Objekte in Verkehrsszenarien. Dazu zählen neben Tieren (Wild, Hunde, Hasen, Vögel, usw.), Hindernissen (Bäume, Absperrungen, Pfeiler, Fahrbahnbegrenzung, usw.) und sonstigen Objekte (Plastiktüten, Heuballen, vom Laster fallende Gegenstände, usw.) insbesondere weitere Verkehrsteilnehmer wie Fahrradfahrer.

Im Jahr 2016 wurden vom ADAC Notbremsassistenten führender Automobilhersteller getestet [Rat16]. Das Ergebnis in Bezug auf das Erkennen von Fahrradfahrern ist leider immer noch mangelhaft: die meisten Fahrradfahrer werden vom Notbremsassistenten der jeweiligen Automobilhersteller überhaupt nicht erkannt und ohne Geschwindigkeitsreduktion überfahren. Der Audi A4 hat mit einem Erfüllungsgrad von 50% noch die beste Bewertung bei diesem Testszenario. Der Volvo V60 fällt bei dem Testszenario komplett durch, obwohl er nach Herstellerangaben einen Notbremsassistenten mit automatischer Fußgänger- und Fahrradfahrer-Erkennung besitzt. Für die Nutzung außerhalb von Autobahnen ist hochautomatisiertes Fahren aktuell also noch nicht geeignet. Dabei gewinnt das Thema Fahrradfahrererkennung immer mehr an Relevanz: die Anzahl der schwerverletzten Unfallbeteiligten Fahrradfahrer in Berlin stieg von 480 in 2001 auf 681 in 2017 - ein Anstieg um 141%. Bei jedem 4. Verkehrstoten im Rahmen eines Verkehrsunfalls handelt es sich um einen Radfahrer [Ber17]. Radfahrer haben in Berlin über 80% ihrer Unfälle mit Kfz [Fah18]. Mit einer zuverlässigen Fahrradfahrererkennung in Serienautos können diese Zahlen erheblich reduziert werden.

Die Klassifizierung und Lokalisierung von Objekten mittels handgeschriebener Algorithmen des maschinellen Sehens wurde im Jahr 2012 von Deep Learning Algorithmen abgelöst als, Geoffrey Hinton et al. ein künstliches neuronales Netzwerk (KNN) trainierten und damit deutlich bessere Fehlerraten erzielten [KSH12]. Das KNN wurde später als AlexNet bekannt; und obwohl dieses mittlerweile nicht mehr aktuell ist, wird es als wesentliche Grundlage für aktuelle Forschungen verwendet. Die Theorie für die verwendeten Deep Learning Algorithmen gab es bereits 1986 [RHW86]. Deep Learning ist ein Bereich des maschinellen Lernens. Dabei wird eine Reihe von Techniken zur automatischen Erkennung und Wiederverwendung von Merkmalen verwendet und mit einem KNN umgesetzt. Diese Technik erzielt zurzeit in vielen Bereichen Durchbrüche, weil Probleme für die ein KNN trainiert wurde, genauer und schneller gelöst werden als es Menschen vermögen. Anwendungsfälle

variieren von Branche zu Branche, von der Erkennung verschiedener Krankheiten in der Medizintechnik, künstliche Go-Spieler oder das Erkennen, zu welcher Baumart ein Stück Baumrinde auf einem Foto gehört. Auch in der Automobilindustrie ist Deep Learning ein wichtiges Mittel, um verschiedene Probleme zu lösen.

1.1 Fragestellung

Im Rahmen dieser Arbeit wird der Fragestellung nachgegangen, wie zuverlässig und schnell Fahrradfahrer auf Kamerabildern mit einem künstlichen neuronalen Netzwerk detektiert werden können - insbesondere in eingebetteten Systemen und in Hinblick auf unterschiedliche, reale Verkehrsszenarien.

1.2 Aufbau der Arbeit

Um einen Lösungsansatz zu erarbeiten, werden zunächst die wichtigsten Grundlagen des maschinellen Lernens, künstlicher neuronaler Netzwerke und Detektoren für Bilder in Kapitel 2 erläutert. In Kapitel 3 wird das Ziel dieser Arbeit im Detail beschrieben und eine Anforderungsanalyse mit Anwendungsfällen durchgeführt. Kapitel 4 beschreibt die Auswahl des Lösungswegs und des künstlichen neuronalen Netzwerks, das Zusammenstellen des verwendeten Datensatzes, das Training des Netzwerks und die verwendete Soft- und Hardware. Die Systemarchitektur und das dynamische Verhalten werden in Kapitel 5 beschrieben. Im Kapitel 6 wird das implementierte System getestet, die Tests evaluiert, Limitationen aufgeführt und ein Fazit gezogen. Abschließend wird die Arbeit in Kapitel 7 zusammengefasst.

Fachliche Grundlagen

Dieses Kapitel dient der Einführung der theoretischen Grundlagen dieser Arbeit. Für genauere und tiefere Einblicke in den Bereich des maschinellen Lernens, sowie der künstlichen neuronalen Netzwerke (KNN), wird das Buch „Machine learning in action“ von Peter Harrington [Har12] und folgende Kurse von Coursera.org empfohlen: „Maschinelles Lernen“ der Stanford University und „Die fünf Spezialisierungen für Deep Learning“ von deeplearning.ai. Diese Kurse werden von Andrew Ng gehalten, Professor für maschinelles Lernen an der University of Cambridge und weltweit führender Experte für künstliche Intelligenz.

Dieses Kapitel zielt darauf ab, kompakt und verständlich zu sein. Deswegen werden mathematische Vorgehensweisen genannt, aber auf das Abbilden und die Erläuterung der mathematischen Formeln wird verzichtet. Diese werden in den genannten Quellen ausführlich behandelt und tragen dem Verständnis nicht zusätzlich bei. Grundlage dieser Arbeit ist der Bereich der künstlichen Intelligenz. Das maschinelle Lernen ist ein Teil davon. Künstliche neuronale Netzwerke werden für das maschinelle Lernen verwendet. Dementsprechend wird zunächst der Teilbereich des maschinellen Lernens erläutert. Anschließend wird sich den künstlichen neuronalen Netzwerken gewidmet. Beruhend auf dem Wissen über KNN werden die faltenden neuronale Netzwerke erklärt. Abschließend werden Detektoren eingeführt, welche auf Bildern Objekte klassifizieren und lokalisieren.

2.1 Maschinelles Lernen

Zunächst wird sich der Frage gewidmet, was genau Bedeutung und Ziel des maschinellen Lernens ist, um anschließend zu verstehen, wieso maschinelles Lernen bei der Detektion von Fahrradfahrern auf digitalen Bildern Verwendung findet. Der Grundgedanke des maschinellen Lernens wird in der Definition von Tom Mitchell treffend beschrieben:

“ Definition: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

— **Tom Mitchell**
[Mit97, S. 2]

Ein Beispiel zur Verdeutlichung des Zitats ist das Klassifizieren von E-Mails als Spam. Die Erfahrung E wäre die Anzahl der vom Benutzer als Spam klassifizierten E-Mails. Die Aufgabe T wäre das Klassifizieren einer E-Mail. Die Performance P wäre die relative Anzahl korrekt klassifizierter E-Mails.

Der Algorithmus eines Programms ist dafür zuständig, aus den Eingabeparametern korrekte Ausgaben zu erzeugen. Diese Algorithmen werden im Allgemeinen von Programmierern explizit erstellt. Im Gegensatz dazu wird ein Programm mit maschinellen Lernalgorithmen nicht explizit programmiert, sondern muss erst lernen richtige Ausgaben zu erzeugen. Das Lernen geschieht beim Trainieren mit Daten. Grundsätzlich lässt sich beim Training unterscheiden zwischen überwachtem und unüberwachtem Lernen. Diese Unterscheidung bezieht sich in erster Linie auf die zum Trainieren verwendeten Daten. Beim unüberwachten Lernen wird mit Eingabedaten ohne erwartete Ausgabedaten trainiert. Der Algorithmus soll in den Eingabedaten Muster erkennen und ein Modell erzeugen, um die Eingabedaten zu beschreiben. Im Rahmen dieser Arbeit wird das überwachte Lernen verwendet und im Folgenden weiter erläutert.

2.1.1 Überwachtes Lernen

Das überwachte Lernen lässt den Lernalgorithmus mit Eingabe- und Ausgabedaten trainieren. Die Ausgabedaten müssen für diese Art des Lernens korrekt und bekannt sein. Der Algorithmus versucht, eine Hypothese zu finden, die möglichst zielsichere Voraussagen trifft. Unter dem Begriff Hypothese ist eine Abbildung zu verstehen, die

jedem Eingabewert den vermuteten Ausgabewert zuordnet, vgl Mohir et al. [MRT18]. Die Hypothese enthält Parameter, die durch das Training eingestellt werden. Diese Parameter werden Gewichte genannt.

Das überwachte Lernen wird weiter unterteilt in Regressions- und Klassifikationsprobleme. Ein Regressionsproblem entsteht, wenn die Ausgabedaten beliebige quantitative Werte eines vorgegebenen Wertebereiches annehmen können. Liegen die Ergebnisse in diskreter Form vor bzw. sind die Werte qualitativ, spricht man von einem Klassifikationsproblem, vgl. James et al. [Jam+13, S. 28]. Ein Beispiel für ein Regressionsproblem ist die Vorhersage der Preisentwicklung von Aktien. Ein Beispiel eines Klassifikationsproblems ist die Bestimmung, ob eine E-Mail Spam ist. Im Rahmen dieser Arbeit wird sich mit einem Klassifikationsproblem auseinandersetzen, weil Teile eines Bildes als Fahrradfahrer klassifiziert werden müssen.

Klassifikationsprobleme können auf unterschiedliche Weise gelöst werden: Rosenblatt (vgl. [Ros58]) schlägt den Perzeptron Algorithmus vor und Anderson (vgl. [And07, S. 163 ff.]) beschreibt unterschiedliche Algorithmen zur Lösung. Darunter zählen logistische Regression, Support Vector Machine, K-nearest Neighbour und andere Methoden. Nicht zuletzt werden auch KNN genannt, um Klassifikationen anhand von Eingabedaten vorzunehmen. Ein Algorithmus, der die Eingabedaten in mehrere Klassen klassifizieren muss, hat entsprechend mehr Ausgabedaten, i.d.R ein Ausgabewert je Klasse, in der die Wahrscheinlichkeit für die Klassenzugehörigkeit abgebildet wird.

Beim überwachten Lernen werden die berechneten Ausgabedaten mit den korrekten Ausgabedaten verglichen. Dies geschieht mit der Berechnung einer Kostenfunktion bzw. des Verlusts (englisch: Cost Function or Loss). Dazu wird der euklidische Abstand ermittelt, zwischen den mit der Hypothese berechneten und den korrekten Ausgabedaten. Der euklidische Abstand wird auch als Fehler bezeichnet. Das Ziel ist diesen Fehler zu minimieren, damit die berechneten Ausgabedaten den korrekten Ausgabedaten entsprechen. Zum besseren Verständnis dient folgende Metapher: Man befindet sich auf einem Berg und möchte das Tal erreichen. Die einzigen verfügbaren Informationen sind die Höhen (die berechneten Fehler). Durch die Verbildlichung wird auch klar, dass die Wahl des Startpunktes entscheidend ist. Jeder Abhang des Berges kann entweder zu einem tiefer oder höher gelegenem Tal führen. In Abbildung 2.1 ist zu sehen, dass je nach Startpunkt ein anderes Minimum erreicht wird. Der Startpunkt wird durch die nicht trainierten Gewichte der Hypothese festgelegt, diese sind gleichmäßig, aber nicht symmetrisch zu gestalten, vgl. Bishop [Bis+95, S. 353]. Abgesehen von der Wahl des Startpunktes läuft das Minimieren des Fehlers auf ein Gradientenverfahren hinaus.

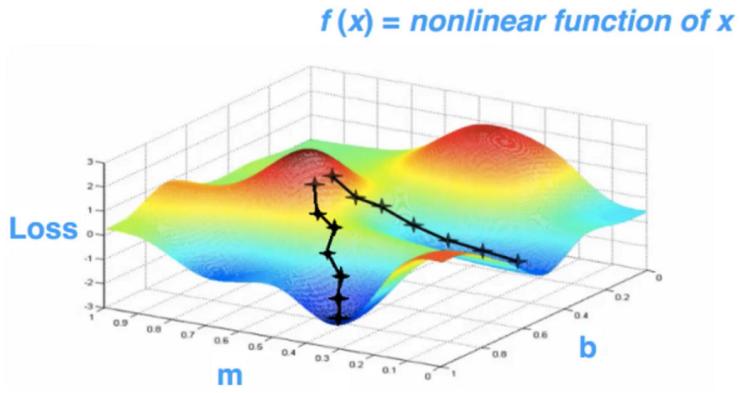


Abb. 2.1.: Abbildung einer Funktion mit mehreren lokalen Minima. Für zwei unterschiedliche Startpunkte werden verschiedene Minima erreicht.
Quelle: <https://medium.com/analytics-vidhya/snapshot-ensembles-leveraging-ensembling-in-neural-networks-a0d512cf2941>

2.1.2 Gradientenverfahren

Das Gradientenverfahren verwendet die Ableitung der Kostenfunktion, um die Abweichung jedes Gewichtes zu korrigieren. Um bei der Metapher der Berg und Tal Landschaft zu bleiben, weist ein Gradient in Richtung bergabwärts. Diese Richtung ist erstens nicht zwingend die Richtung des lokalen Minimums, und zweitens ist nicht bekannt, wie weit ein Minimum entfernt ist. Die Lösung dafür ist, lediglich kleine Schritte zu gehen und anschließend erneut zu prüfen, welche die korrekte Richtung ist. Die Größe der Schritte, die bergab gegangen wird, beschreibt die Lernrate. Die Lernrate ist eine Trainingsvariable und wird beim Korrigieren der Gewichte verwendet, vgl. Zell [Zel94, S. 106].

Das Gradientenverfahren verwendet in seiner natürlichen Form den gesamten Datensatz. Je nach Größe der Eingabedaten, des gesamten Datensatzes und der Komplexität der Hypothese kann das Berechnen des Gradienten für den gesamten Datensatz sehr rechenintensiv und damit auch zeitintensiv sein. Das stochastische Gradientenverfahren (SGV) ist eine Erweiterung des Gradientenverfahrens und verwendet nur ein Datenpaar, um den Gradienten zu berechnen. Ein Kompromiss ist, mehrere Datenpaare des Datensatzes zu verwenden, um einen besseren Gradienten bei weniger Rechenaufwand zu erhalten. Dieses Verfahren ist im Englischen als Mini-batch gradient descent bekannt. Verschiedene Erweiterungen und Varianten des SGV werden aktuell verwendet. Aktuelle Frameworks (z.B. TensorFlow) erlauben die Wahl des Gradientenverfahrens. Momentum ist eine einfache und effiziente Erweiterung des SGV, welche seit einigen Dekaden erfolgreich im Sektor des maschinellen Lernens verwendet wird, vgl. Zeiler [Zei12, S. 2]. Momentum ist einstellbar und deswegen auch eine Trainingsvariable. Im folgenden Unterkapitel werden wichtige Punkte für das Training erklärt.

2.1.3 Das Training

Das Training zielt darauf ab, der Hypothese die zu Grunde liegende Struktur der Eingabedaten beizubringen, um bei Eingabedaten mit denen nicht trainiert wurde valide Ausgabedaten zu erhalten. Beim überwachten Lernen werden Datenpaare bereitgestellt, welche die Eingabedaten und die erwarteten Ausgabedaten beinhalten. Der Lernalgorithmus verwendet die Eingabedaten, um Ausgabedaten zu berechnen. Mit der Kostenfunktion wird der Fehler zwischen berechneten und erwarteten Ausgabedaten berechnet. Mit einem Gradientenverfahren wird der Fehler der Kostenfunktion minimiert, indem die Hypothese angepasst wird.

Für das Training wird der vorhandene Datensatz in Trainingsdaten und Testdaten aufgespalten, damit ein Datensatz zum Testen vorhanden ist, der dem Algorithmus unbekannt ist. Den Datensatz so aufzuteilen wird als Holdout Verfahren bezeichnet, vgl. Kuhlmann [Kuh09, S. 538]. Beim Training wird zwischen einer Iteration bzw. einem Batch und einer Epoche unterschieden. Eine Iteration ist durchlaufen, sobald mit der Anzahl der Datenpaare trainiert wurde, die für die Berechnung des Gradienten verwendet werden. Eine Epoche bedeutet, dass alle Trainingsdaten einmal verwendet wurden. Während des Trainings sollte sich der Fehler stets minimieren. Je nach Gradientenverfahren darf allerdings eine Schwankung des Fehlers auftreten. Das Training ist beendet, wenn sich der Fehler nicht mehr verkleinert. Anschließend wird die Hypothese mit Daten getestet, welche während des Trainings nicht verwendet wurden. Mithilfe der Kostenfunktion wird der Fehler ermittelt. Drei Fälle können eintreten:

- Eine Unteranpassung (engl. Underfitting) ist das Resultat einer hohen Verzerrung auch genannt Bias. Eine Unteranpassung liegt vor, wenn die Hypothese bei Trainingsdaten und bei Testdaten schlechte Ausgabedaten liefert. Das Hauptproblem in diesem Fall ist, dass die grundlegende Struktur der Eingabedaten nicht korrekt erfasst wird, weil die Hypothese zu unflexibel ist.
- Eine hohe Varianz führt zu einer Überanpassung (engl. Overfitting). Die Überanpassung zeigt sich durch schlechte Ausgabedaten bei Testdaten im Gegensatz zu deutlich korrekteren Ausgabedaten bei Trainingsdaten. In diesem Fall erfasst die Hypothese das Rauschen in den Trainingsdaten, weil die Hypothese zu komplex für die Eingabedaten ist und bzw. oder nicht genug Trainingsdaten vorliegen, aus der die grundlegende Struktur hervorgeht.
- Ist ein guter Bias-Varianz Kompromiss vorhanden, ist das Ergebnis eine geringe Fehlerrate bei Test- und Trainingsdaten.

Möglichkeiten, um der Unteranpassung entgegenzuwirken, sind das Erhöhen der Komplexität der Hypothese durch zusätzliche Verwendung der Eingabedaten oder die Verwendung weiterer Eingabedaten. Die Überanpassung kann durch das Erhöhen der Trainingsdaten, eine einfachere Hypothese oder eine Regulierung verhindert werden. Die Regulierung ist eine Methode, welche durch einen zusätzlichen Term in der Kostenfunktion angewendet wird. Ein zusätzlicher Vorteil der Regulierung ist, dass die Gewichte klein bleiben, dadurch alle Gewichte berücksichtigt werden und einzelne Gewichte das Ergebnis nicht maßgeblich bestimmen. Während des Trainings können die aktuellsten Gewichte mit den Testdaten verwendet werden, um den aktuellen Fehler beim Verwenden der Testdaten zu messen. Wenn sich dieser Fehler erhöht, dann ist das ein Anzeichen von Overfitting. Zusätzlich handelt es sich hier um einen Punkt, an dem das Training frühzeitig abgebrochen werden sollte (engl.: early stopping), vgl. Prechelt [Pre98, S. 2].

Um den bestmöglichen Kompromiss zwischen Bias und Varianz zu erhalten, empfiehlt Bishop (vgl. [Bis+95, S.372]) verschiedene Hypothesen zu trainieren und mit einem Kreuzvalidierungsverfahren die beste Hypothese zu bestimmen. Andrew Ng empfiehlt das Aufteilen der vorhandenen Daten in Trainings-, Validierungs- und Testdaten. Die Validierungsdaten sollen verwendet werden, um verschiedene Hypothesen zu trainieren und der Vergleich soll dann mit dem Fehler der Testdaten erfolgen. Dementsprechend sind die Validierungsdaten nur ein relativ kleiner möglichst vielfältiger Teil der Trainingsdaten. Die Validierungsdaten können nach der Auswahl einer geeigneten Hypothese für das Training verwendet werden.

Eine grobe Übersicht der im Rahmen dieser Arbeit wichtigsten Punkte des maschinellen Lernens wurden erörtert. Um Klassifikationsprobleme zu lösen kann ein KNN verwendet werden. Im Rahmen dieser Arbeit sollen auf Bildern Objekte klassifiziert werden, deswegen wird im folgenden Kapitel der Aufbau und die Funktionsweise von KNN erklärt.

2.2 Künstliche neuronale Netzwerke

„Ein menschliches Gehirn hat 100 Milliarden Neuronen und 100 Billionen Synapsen und arbeitet mit 20 Watt (genug, um eine Glühbirne zu betreiben) - im Vergleich dazu hat das größte künstliche neuronale Netzwerk 10 Millionen Neuronen mit 1 Milliarde Verbindungen auf 16 000 CPUs und verbraucht 3 Millionen Watt“

— Autor unbekannt
übersetzt aus dem Englischen
Quelle: [Unb18]

Künstliche neuronale Netzwerke bilden die Funktionsweise von biologischen Neuronen im Gehirn vereinfacht ab. Neuronen im menschlichen Gehirn bestehen aus vielen verschiedenen Teilen. Die linke Seite der Abbildung 2.2 zeigt dies in einer bereits stark vereinfachten Darstellung. Zwischen den Enden des Axon und den Dendriten eines anderen Neurons bilden sich im Gehirn die Synapsen, welche in Abbildung 2.2 für das biologische Neuron nicht aufgeführt sind. Auf der rechten Seite in Abbildung 2.2 ist ein künstliches Neuron dargestellt. Ein künstliches Neuron wird auch Knoten genannt. Der Prozess der Verarbeitung von Signalen in einem

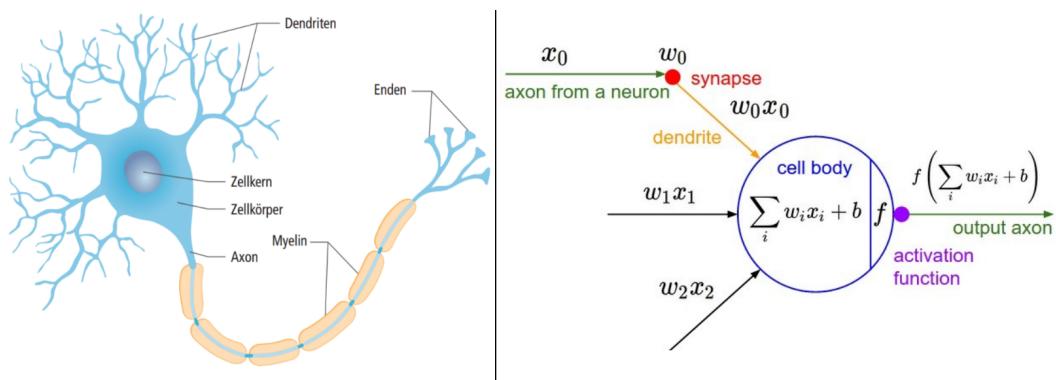


Abb. 2.2.: Gegenüberstellung einer vereinfachten Darstellung eines biologischen Neurons im menschlichen Gehirn (links) und eines künstlichen Neurons (rechts).
Quellen: links [Spi18], rechts https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/more_images/neuron_model.jpeg

Knoten lässt sich folgendermaßen beschreiben: Ein Dendrite erhält einen Wert aus den Eingabedaten. Jeder Eingabewert wird mit einem Gewicht (die Parameter der Hypothese) multipliziert. Im Zellkörper werden die gewichteten Eingabedaten aufsummiert und ein Schwellwert, auch Bias genannt, addiert (Körperfunktion). Mittels einer Aktivierungsfunktion entsteht der Ausgabewert. Der Ausgabewert liegt an

den Enden des Axon an. Die Gewichte lassen sich als Synapsen auffassen, weil sie die Verbindung zwischen zwei Knoten abbilden bzw. auch die Stärke der Verbindung zwischen ebendieser darstellen; die Gewichte können auch negative Zahlen aufweisen.

Aktivierungsfunktion

Die Aktivierungsfunktion berechnet die Ausgabe des künstlichen Neurons. Die Aktivierungsfunktion ist i.d.R. eine monoton wachsende Funktion. Der aus den Eingangs値en mittels der Körperfunktion berechnete Wert, wird der Aktivierungsfunktion übergeben. Die Wahl der Aktivierungsfunktion beeinflusst die Genauigkeit von Klassifikationen, wie Harmon und Klabjan beim vergleichen verschiedene Aktivierungsfunktionen zeigen, vgl. [HK17, S. 5]. Die Aktivierungsfunktion ist zum Beispiel ein Tangens Hyperbolicus, wenn der Ausgabewert auf maximal 1 beschränkt werden soll. Die Aktivierungsfunktion kann aber auch linear sein und zum Beispiel das Ergebnis der Körperfunktion wiedergeben. Wenn Die Aktivierungsfunktion binär ist, also nur 0 oder 1 ausgegeben wird, dann wird häufig der Name Perzepron anstatt Neuron bzw. neuronales Netzwerk verwendet. Das Perzepron wurde 1958 von Rosenblatt vorgestellt, vgl. [Ros58]. Nach der Einführung eines einzelnen Knotens wird im Folgenden erläutert wie Knoten miteinander agieren und ein Netzwerk bilden.

2.2.1 Struktur künstlicher neuronaler Netzwerke

Ein künstliches neuronales Netzwerk besteht aus miteinander verbundenen Knoten, diese sind in Schichten organisiert. Jede Schicht besteht aus einem oder mehreren Knoten. Die erste Schicht wird die Eingabeschicht genannt und besteht aus den Eingabedaten. Jeder Wert der Eingabedaten wird von einem Knoten repräsentiert. In der Eingabeschicht werden i.d.R. keine Berechnungen durchgeführt. Dementsprechend gilt für die Knoten der Eingabeschicht, das Ausgabewert dem Eingabewert gleicht. Die Ausgabedaten einer Schicht werden dann zu den Eingabedaten der nachfolgenden Schicht. Schichten haben unterschiedliche Typen. Der Typ bestimmt die Anzahl der Knoten, welche Knoten welche Eingabedaten erhalten und die Berechnung der Eingabedaten.

Die Struktur eines KNN setzt sich mindestens aus einer Eingabe- und einer Ausgabeschicht zusammen. In diesem Unterkapitel wird gezeigt, dass diese zwei Schichten nicht ausreichen, um komplexere Funktionen abzubilden. Komplexere Funktionen können mit der Einführung von weiteren Schichten zwischen Eingabe- und Ausgabeschicht berechnet werden. Diese weiteren Schichten werden versteckte Schichten

genannt. Sobald mindestens eine versteckte Schicht vorhanden ist, ist das Netzwerk ein tiefes neuronales Netzwerk (engl.: Deep Neural Network (DNN)).

Anhand eines simplen Netzwerks wird gezeigt, dass die XOR-Funktion nicht berechenbar ist. Das Netzwerk besteht nur aus Eingabe- und Ausgabeschicht. Anschließend wird dem Netzwerk eine versteckte Schicht hinzugefügt, wodurch die XOR-Funktion nun berechenbar ist. Die Wahrheitstabelle für die logische XOR-Funktion mit zwei Variablen ist in Tabelle 2.1 abgebildet. In Abbildung 2.3 ist ein KNN zu sehen, wel-

Input1	Input2	Ouput
0	0	0
0	1	1
1	0	1
1	1	0

Tab. 2.1.: Wahrheitstabelle für die logische XOR-Funktion mit zwei Variablen

ches aus einer Eingabeschicht und einer Ausgabeschicht besteht. Die Eingabeschicht besteht aus den beiden Inputs **Eingabe1** und **Eingabe2**. Die Ausgabeschicht besteht aus einem künstlichen Neuron **Output**. Als Aktivierungsfunktion wird hier eine bi-

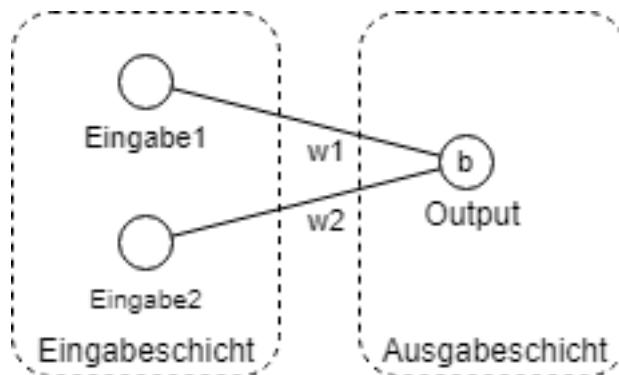


Abb. 2.3.: Darstellung einer Eingabeschicht mit zwei Werten und einer Ausgabeschicht mit einem Neuron. An den beiden Linien sind die Gewichte w_1 und w_2 gekennzeichnet. Im Neuron wird der Bias b aufgeführt.
Quelle: Eigene Darstellung

näre Aktivierungsfunktion gewählt, für die gilt, alles über 0 ergibt eine 1, alles unter oder gleich 0 ergibt eine 0:

$$af(x) = x > 0 ? 1 : 0$$

Durch die Verwendung eines künstlichen Neurons ergibt sich die Hypothese:

$$output = af(w_1 * Eingabe1 + w_2 * Eingabe2 + b)$$

Stellt man das lineare Gleichungssystem auf, ergibt sich hierfür keine Lösung, da die letzte Zeile den beiden mittleren widerspricht:

$$\begin{aligned} 0 &= w_1 * 0 + w_2 * 0 + b \Rightarrow b = 0 \\ 1 &= w_1 * 0 + w_2 * 1 + b \Rightarrow w_2 = 1 \\ 1 &= w_1 * 1 + w_2 * 0 + b \Rightarrow w_1 = 1 \\ 0 &= w_1 * 1 + w_2 * 1 + b \Rightarrow w_1 = -w_2 \end{aligned}$$

Die Lösung hierfür ist eine weitere versteckte Schicht einzufügen. Diese neue Schicht besteht aus zwei künstlichen Neuronen, wie in Abbildung 2.4 zu sehen ist. Die

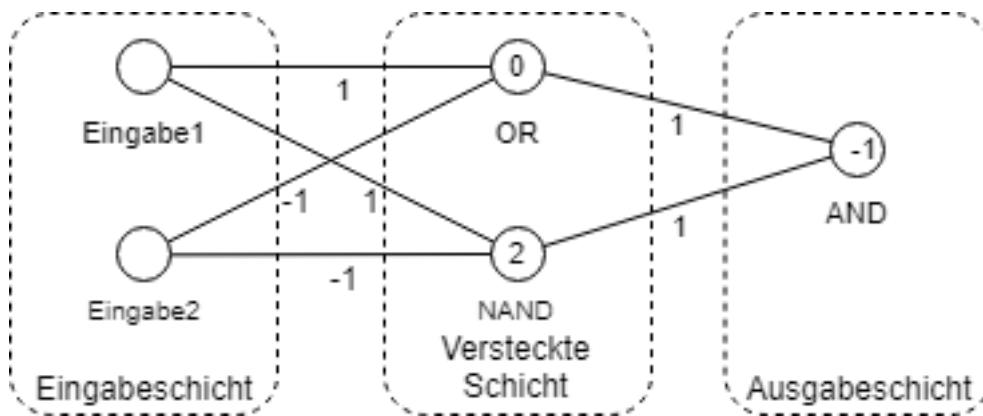


Abb. 2.4.: Realisierung der logischen XOR-Funktion für zwei Eingangswerte in einem neuronalen Netzwerk mit einer versteckten Schicht. Die Zahlen unter den Linien stellen jeweils die Werte der Gewichte dar. Die Zahlen in den Knoten stellen die Werte des jeweiligen Bias dar.

Quelle: Eigene Darstellung

Hypothese wird in diesem Fall zu:

$$\begin{aligned} \text{output} &= af(af(1 * \text{Eingabe1} + 1 * \text{Eingabe2} + 0) * 1 + \\ &\quad af(-1 * \text{Eingabe1} + -1 * \text{Eingabe2} + 2) * 1 + (-1)) \end{aligned}$$

In Abbildung 2.4 ist eine mögliche Lösung für die Gewichte und die Bias b zu sehen, um die XOR-Funktion zu erhalten. Gewichte und Bias können Gleitkommazahlen sein. Die Bias sind auch trainierbare Gewichte. Praktisch wird der Bias umgesetzt, indem das Bias-Gewicht mit einem nicht vorhandenen Eingabewert, der immer gleich 1 ist, multipliziert wird. Diese drei künstlichen Neuronen sind jetzt in der Lage die XOR-Funktion abzubilden, indem ein Neuron der versteckten Schicht die OR-Funktion und ein Neuron die NAND-Funktion abbildet und das Neuron in der Ausgabeschicht eine AND-Funktion. In Tabelle 2.2 sind die Ausgabewerte der einzelnen Knoten entsprechend der Eingabeparameter abgebildet.

Eingabe1	Eingabe2	OR	NAND	AND (XOR)
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

Tab. 2.2.: Ausgabewerte der drei Knoten des XOR-Netzwerks für alle vier möglichen Kombinationen der Eingabeparameter

In diesem Kapitel wurde gezeigt, dass ein KNN einen Algorithmus nur mit Hilfe von zuvor trainierten Gewichten berechnet. Die Berechnung wird im Englischen forward pass genannt. Das Gegenstück dazu ist die Fehlerrückführung, welches im Englischen der backward pass oder auch die backpropagation of Error ist.

2.2.2 Fehlerrückführung

Mit der Fehlerrückführung werden die Gewichte in einem KNN trainiert. Die Formel für die Fehlerrückführung wird durch Differentiation hergeleitet, wobei sich die partielle Ableitung der Fehlerfunktion durch Verwendung der Kettenregel ergibt. Wie beim maschinellen Lernen wird auch hier ein Gradientenverfahren angewendet, um den Fehler des Ausgabewertes zu minimieren. Der Unterschied ist, dass in diesem Fall für jedes künstliche Neuron ein Gradient anhand seiner Ausgabe erstellt wird.

2.3 Faltende neuronale Netzwerke

Für die Bildverarbeitung essenziell sind sogenannte faltende neuronale Netzwerke (engl.: Convolutional Neural Networks (CNN)). Insbesondere für die Klassifikation von Bildern werden CNN verwendet. In der digitalen Bildverarbeitung wird die diskrete Faltung verwendet, um z.B. Kantenbilder zu erstellen, Merkmale zu extrahieren oder Merkmale auf einem Bild wiederzufinden. Diesen Ansatz verwenden CNN. Eine diskrete Faltung geschieht mit einem Faltungskern, dieser wird hier auch Filter oder im Englischen Feature Detector genannt. Ein Faltungskern ist immer quadratisch. Bei einer Größe von drei entspricht das einem Faltungskern von 3x3. Diese neun Werte des Faltungskerns sind trainierbare Gewichte. Eine diskrete Faltung wird auf einer zwei dimensionalen Matrix angewendet, wobei jeder Wert in der Matrix mitberechnet wird. Das Ergebnis einer diskreten Faltung ist wiederum eine zwei dimensionale Matrix. Digitale Bilder bestehen aus einer zwei dimensionalen Matrix je Farbkanal, also pro Pixel und Farbe ein Wert. Dementsprechend haben Graustufenbilder einen Farbkanal und farbige Bilder z.B. je einen Farbkanal für Rot, Grün und Blau. Bei mehreren Farbkanälen wird jeweils jeder Filter angewendet.

Eine Schicht, welche diskrete Faltung anwendet, heißt Faltungsschicht (engl.: Convolutional Layer). Die Ausgabedaten einer Faltungsschicht sind Feature Maps oder auch Activation Maps. Diese Feature Maps zeigen, an welcher Stelle ein Merkmal gefunden wurde bzw. wie hoch die Aktivierung für ein bestimmtes Merkmal an einer bestimmten Stelle ist. Die Größe einer Feature Map wird von drei Parametern bestimmt:

- Die Tiefe entspricht der Anzahl der Filter, da jeder Filter eine diskrete Faltung durchführt und damit eine zweidimensionale Matrix ausgibt.
- Die Schrittweite gibt an, wie viele Pixel bzw. Werte während der Faltung übersprungen werden. Durch Schrittweiten größer als eins kann die Ausgangsmatrix größer sein als die Eingangsmatrix.
- Zero-Padding bestimmt, ob die Ränder mit Nullen aufgefüllt werden sollen, um auch die äußersten Pixel zu verrechnen.

Wie zuvor gezeigt, werden versteckte Schichten eingeführt, um komplexere Funktionen zu ermöglichen. Genauso verhalten sich Faltungsschichten: werden mehrere Schichten miteinander kombiniert, können dadurch komplexere Merkmale extrahiert bzw. wiedererkannt werden. Durch die Verwendung mehrerer Filter werden zunächst Kanten, Ecken, Rundungen u.ä. erkannt. Diese Merkmale werden dann zu komplexeren Merkmalen zusammengesetzt, wie Abbildung 2.5 zeigt.

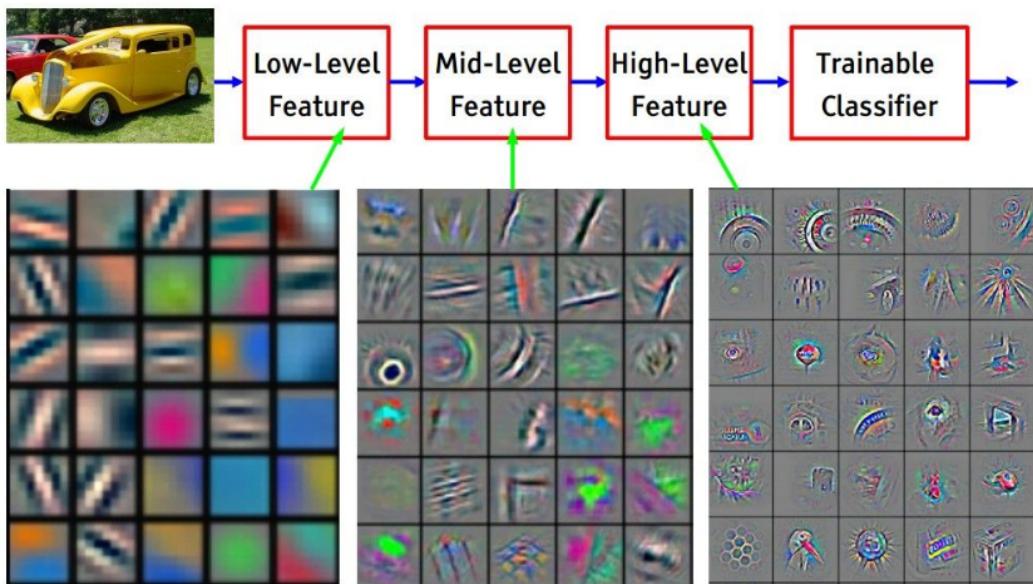


Abb. 2.5.: Unterschiedlich komplexe Merkmale in einem Bild, während des Verlaufs einer Klassifizierung durch ein faltendes neuronales Netzwerk.
Quelle: https://cdn-images-1.medium.com/max/1600/0*Hv4jsLa_iyo8UR6S.png

Die ersten Architekturen (z.B. LeNet 5 oder YOLOv1) verwenden nach einer Faltungsschicht eine Schicht, die Max-Pooling durchführt. Max-Pooling bestimmt aus einem bestimmten Bereich den maximalen Wert und wird zum Verkleinern der Feature Map verwendet. Allerdings lässt sich das auch alternativ mit einer Faltungsschicht durchführen. Anhand von Feature Maps wird schließlich eine Klassifikation des Bildes oder von Bildbereichen durchgeführt. Die Klassifikation kann z.B. durch vollständig vernetzte Schichten oder durch eine weitere Faltungsschicht geschehen. Ein Problem bei tiefen Netzwerken ist eine Sättigung der Lernrate bzw. ein zu geringer Abbau dieser. Die Lösung dafür sind Rückstandsschichten (engl.: Residual Block) die Informationen von vorangegangenen Schichten zur aktuellen hinzufügen, vgl. He et al. [He+16].

2.4 Objekt Detektoren

Die Objekt Detektion befasst sich zusätzlich zum Klassifizieren von Objekten auf Bildern mit der Lokalisierung der Objekte auf dem Bild. Zurzeit werden zwei Ansätze verfolgt. Der erste Ansatz verwendet Segmentierung, das bedeutet, dass nur der gesamte Bereich des Objekts markiert wird, wie in Abbildung 2.6 auf der rechten Seite zu sehen ist. Der zweite Ansatz verwendet Begrenzungsrahmen (engl.: Bounding

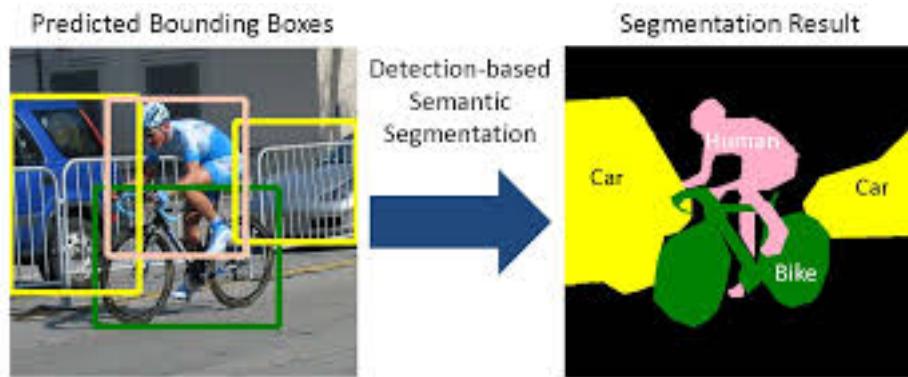


Abb. 2.6.: Gegenüberstellung von zwei Methoden der Lokalisierung von Objekten auf Bildern. Links werden Begrenzungsrahmen zur Lokalisierung von Objekten verwendet. Rechts sind für das gleiche Bild nur die klassifizierten Segmente zu sehen.
Quelle: [Xia+13, S. 1]

Box). Hier befindet sich das Objekt vollständig innerhalb des Begrenzungsrahmen und berührt diesen an jeder Seite. Für das Training von Objekt Detektoren werden Datenpaare gebraucht, die als Eingabedaten Bilder und als Ausgabedaten Markierungen haben. Markierungen beschreiben zum einen die Lokalisierung als Segmente oder mit Begrenzungsrahmen. Zum anderen die Klassifikation, mindestens eine Klasse je Lokalisierung. Bilder werden auf die Größe skaliert, die von der ersten versteckten Schicht erwartet wird.

Dieses Kapitel gibt einen groben Überblick über die im Rahmen dieser Arbeit verwendeten Theorien und Methoden, so dass im Folgenden insbesondere auf das hier vorgestellte Vokabular zurückgegriffen werden kann.

Aufgabenstellung

In diesem Kapitel wird der Anforderungskatalog erstellt. Das Ziel wird allgemein erläutert und Teilziele werden formuliert. Anschließend wird der Geltungsbereich dieser Arbeit eingegrenzt. Die Anforderungsanalyse mit Anwendungsfällen wird hauptsächlich verwendet, um die Anforderungen zu ermitteln. Abschließend finden sich die Anforderungen unterteilt in funktionale und nicht-funktionale Anforderungen. Das Kapitel wird mit einem Anwendungsfall geschlossen.

3.1 Ziel der Arbeit

Die aufgeführten Ziele sind nach der SMART-Methode erstellt (Specific Measurable Achievable Reasonable Time Bound). Im Rahmen dieser Arbeit sind die bezifferten Ziele aufgrund der beschränkten Zeit und den vorhandenen Ressourcen so angepasst, dass sie realistisch und insbesondere erreichbar sind. Das Hauptziel wird zunächst grob genannt und dann konkret in zwei Teilziele für das KNN und das eingebettete System gegliedert.

Das Ziel dieser Arbeit ist die Fertigstellung eines eingebetteten Systems mit einer digitalen Kamera und einer Software, die mit einem KNN auf Bildern Fahrradfahrer detektiert. Die Ergebnisdaten sind die Begrenzungsrahmen aus den Detektionen. Die Begrenzungsrahmen werden mit Koordinaten, Breite und Höhe erfasst. Die Ergebnisdaten werden über einen Feldbus nach dem Beobachter Muster veröffentlicht. Angemeldete Beobachter sind Systeme, wie zum Beispiel Fahrerassistenzsysteme oder Überwachungssysteme, die sich für die Ergebnisdaten interessieren. Das System wird im Verlauf dieser Arbeit als Fahrradfahrer Detektionssystem (FFDS) betitelt. Der Fokus beim Detektieren liegt primär auf dem Erkennen von Fahrradfahrern, die tatsächliche Position auf dem Bild ist sekundär, wird jedoch ebenfalls ausgewertet. Die beschränkten Ressourcen, die ein eingebettetes System bietet, stellen eine Herausforderung dar. Erwartet wird, dass ein Kompromiss zwischen Schnelligkeit und Genauigkeit gefunden werden muss.

Ein zuverlässiges Erkennen von Fahrradfahrern bedeutet im Straßenverkehr und insbesondere in Hinsicht auf autonom agierende Fahrzeuge, dass die Software Fahrradfahrer auch bei schlechten Bedingungen erkennt, z.B. bei Gegenlicht oder in

einem schlechtem Winkel. Die Genauigkeit ergibt sich aus Trefferrate und Präzision. Die Trefferrate gibt den Prozentsatz an, wie viele Fahrradfahrer von allen tatsächlich vorhandenen Fahrradfahrern erkannt werden. Veranschlagt wird eine Trefferrate von 95% - nicht erkannte Fahrradfahrer lassen sich z. B. einer sehr weiten Entfernung oder einem fast vollständig verdeckten Fahrrad zurechnen. Die Präzision gibt an, ob eine Detektion tatsächlich ein Fahrradfahrer ist. Das Erkennen von Fahrradfahrern, obwohl keines vorhanden ist muss gering sein, deswegen soll die Präzision 95% betragen. Ein beispielhaftes Problem, das entstehen kann, wenn fälschlicherweise ein Fahrrad auf Kollisionskurs vor dem Auto erkannt wird ist, dass ein nachgelagertes System wie z.B. ein Notbremsassistent aktiviert wird und unvermittelt eine Vollbremsung durchführt. Das kann unangenehm bis gefährlich für die Insassen sein, zu Auffahrunfällen führen und weitere Verkehrsteilnehmer in Gefahr bringen. Bei einer Präzision von 95% sind das bei angenommenen 10 Bildern pro Sekunde ein Fehler in fünf Sekunden - dies wird als annehmbar hingenommen. Die Überschneidung der Position eines Fahrradfahrers auf einem Bild mit der tatsächlichen Position wird wie erwähnt zweitrangig betrachtet und soll mindestens 75% betragen.

3.2 Abgrenzung

Im Rahmen dieser Arbeit wird sich auf das Detektieren von Fahrradfahrern fokussiert. Das impliziert insbesondere die nachfolgenden Abgrenzungen:

- Ein detekterter Fahrradfahrer wird nicht verfolgt, d.h. jedes Bild wird für sich geprüft. Vom Versuch, einen erkannten Fahrradfahrer auf einem folgenden Bild wiederzufinden, wird Abstand genommen.
- Plausibilitätsprüfungen werden nicht unternommen. Als Beispiel ließe sich nennen, wenn ein Fahrradfahrer mitten im Bild auf offener Straße detektiert wird und im darauffolgenden Bild wird dieser nicht wieder detektiert – hier wird keine Annahme getroffen.
- Die Entfernung zu einem detektierten Fahrradfahrer wird nicht berechnet. Diese Berechnung wird Systemen überlassen, die mit geeigneten Sensoren valide Entfernungsdaten ermitteln können.
- Mit dem Argument der beschränkten Sensorik wird ebenfalls davon Abstand genommen, detektierte Fahrradfahrer im Raum zu lokalisieren.
- Fahrräder gibt es in vielen verschiedenen Arten und Formen. Im Rahmen dieser Arbeit wird sich lediglich auf Fahrräder für eine Person mit zwei Rädern in

den gängigen Größen fokussiert. Betrachtet werden primär Mountainbikes, Stadträder, Rennräder, Laufräder, Kinderräder, Trekkingräder, Tourenräder, Crossrad, Cruiser, BMX und ähnliche. Im Rahmen dieser Arbeit werden Tandems, Liegeräder, Einräder, Dreiräder, Vierräder usw. allerdings nicht explizit von der Systemerkennung ausgeschlossen.

Weiterhin wird nur eine Kamera verwendet, welche direkt mit dem eingebetteten System verbunden wird. Der in dieser Arbeit entwickelte Algorithmus kann auf allen Kamerabildern angewendet werden, um Informationen zu erhalten, ob und in welchem Bereich sich ein Fahrradfahrer befindet und z.B. gebremst werden muss, das Tempo erhöht werden kann oder ob abgebogen werden darf.

Die Software wird im Rahmen dieser Arbeit nicht als baremetal Anwendung auf der Hardware implementiert. Der Hauptgrund dafür ist, dass die verwendeten Nvidia GPU Treiber nicht als Quellcode verfügbar sind. Die Binärdateien der Treiber sind abhängig von der Betriebssystem spezifischen Schnittstelle für die GPU. Zusätzlich gilt die Annahme, dass eine baremetal Anwendung keinen signifikanten Gewinn der Performance mit sich bringt, weil die Anwendung zum Großteil auf der GPU läuft.

3.3 Anforderungsanalyse mit Anwendungsfällen

Für die Anforderungsanalyse werden Anwendungsfälle verwendet. Als erstes werden die Akteure identifiziert. Anschließend werden Szenarien mit den Akteuren definiert. Durch Verallgemeinerung der Szenarien werden die Anwendungsfälle bestimmt. Zusätzlich wird ein Use Case Diagramm zur Übersicht erstellt.

3.3.1 Akteure

Das Fahrradfahrer Detektionssystem unterstützt den Autofahrer nur indirekt. Viel mehr interagiert das Fahrradfahrer Detektionssystem (FFDS) jedoch mit weiteren Fahrerassistenzsystemen, zu Englisch Advanced Driver Assistance Systems (ADAS). Über diese erhält der Autofahrer Warnungen, Anweisungen oder Hilfestellungen. Deswegen ist der Autofahrer kein Akteur, aber das Fahrzeug bzw. die ADAS sind Akteure.

3.3.2 Szenarien

Drei Szenarien werden festgehalten, in dem das System verwendet wird. Bei dem ersten Szenario in Tabelle 3.1 handelt es sich um ein VW Golf, welcher das Detek-

tionssystem integriert. Sobald der Motor vom Golf startet, beginnt das FFDS zu arbeiten. Das Verarbeiten von Bildern und das Detektieren von Fahrradfahrern wird in einem zweiten Szenario in Tabelle 3.2 beschrieben. Im dritten Szenario meldet sich ein Notbremsassistent am Feldbus an(im Szenario ein CAN-Bus), um Daten zu erhalten, wenn Fahrradfahrer auf der Frontkamera detektiert werden, siehe Tabelle 3.3.

Szenario	Start und Stopp des FFDS
Akteur-Instanz	VW Golf mit integriertem FFDS

Ereignisfluss

1. Der Motor des VW Golf wird gestartet. Beim Start des Motors wird das FFDS mit Strom versorgt und gestartet.
2. Der Algorithmus wird vorbereitet. Dazu wird das KNN und die Gewichte in den Arbeitsspeicher der GPU geladen.
3. Die Verbindung zur Frontkamera wird hergestellt.
4. Die Kommunikation über den CAN-Bus wird initialisiert.
5. Die Detektionsroutine startet, siehe Szenario 2: Detektionsablauf.
6. Beim Ausschalten des Motors wird das FFDS ausgeschaltet.

Tab. 3.1.: Szenario 1: Start und Stopp des FFDS

Szenario	Detektionsablauf (mit und ohne Detektionen)
Akteur-Instanz	-
Ereignisfluss	
	<ol style="list-style-type: none"> 1. Nachdem das KNN mit Gewichten in den Arbeitsspeicher der GPU geladen wurde und die Verbindung zur Frontkamera eingerichtet ist, startet die Schleife der Detektionsroutine. 2. Das FFDS lädt das nächste Bild. 3. Die Daten des Bildes werden zur Verarbeitung in den Arbeitsspeicher der GPU geladen. 4. Die GPU berechnet die Detektionen von Fahrradfahrern auf dem Bild. 5. <ol style="list-style-type: none"> a) Das FFDS veröffentlicht über den CAN-Bus mit der ID „DE7EC7“ die Ergebnisdaten aller detektierten Fahrradfahrer des zuletzt berechneten Bildes. Mit dieser CAN-Bus ID werden die Ergebnisdaten detektiert Fahrradfahrer auf der Frontkamera übertragen. b) Für den Fall, dass keine Detektionen vorliegen, wird nichts veröffentlicht. 6. Bei Punkt 2 wird fortgefahrt.

Tab. 3.2.: Szenario 2: Detektionsablauf

Szenario	Erhalten von Ergebnisdaten
Akteur-Instanz	ADAS für Notbremse
Ereignisfluss	
	<ol style="list-style-type: none"> 1. Das ADAS wird aktiviert, wenn sich das Fahrzeug vorwärts bewegt. 2. Das ADAS meldet sich als Beobachter für die CAN-Bus ID „DE7EC7“ an. Mit dieser CAN-Bus ID werden die Ergebnisdaten von auf der Frontkamera detektierten Fahrradfahrer übertragen. 3. Nach der Anmeldung erhält das ADAS die Ergebnisdaten aller detektierten Fahrradfahrer je Bild und nur wenn Detektionen auftreten. 4. Das ADAS kann die Ergebnisdaten anschließend verarbeiten.

Tab. 3.3.: Szenario 3: Erhalten von Ergebnisdaten

3.3.3 Anwendungsfälle

Die aus den Szenarien zusammengefassten und verallgemeinerten Anwendungsfälle sind erstens „Anwendungsfall: FFDS starten“, zweitens „Anwendungsfall: Fahrradfahrer detektieren“ und drittens „Anwendungsfall: Detektionen beobachten“. Weitere denkbare Anwendungsfälle, welche im Rahmen dieser Arbeit nicht ausführlich beschrieben werden, sind unter anderem:

- Das Ermitteln von maximalen Aufkommen von Fahrradfahrern.
- Der Einsatz in Zonen, wo keine Fahrradfahrer erlaubt sind, wie z.B. Einkaufszentren, Fußgängerwege/-tunnel/-brücken, Bahnsteige und generell Gebäude.
- Das Zählen von Fahrradfahrern, was die eindeutige Wiedererkennung eines Fahrradfahrers voraussetzt, um diese nicht doppelt zu zählen. Dazu sei erwähnt, dass bei einer zeitunkritischen Anwendung problemlos zusätzliche Bildvergleiche und -verarbeitungen vorgenommen werden können.

Nachfolgend sind die drei Anwendungsfälle ausführlich in Tabelle 3.4, Tabelle 3.5 und Tabelle 3.6 beschrieben.

Anwendungsfall	FFDS starten
Akteur-Instanz	Fahrzeug mit integriertem FFDS
Ereignisfluss	<ol style="list-style-type: none"> 1. Der Motor des Fahrzeugs wird gestartet. Beim Start des Motors wird das FFDS mit Strom versorgt und gestartet. 2. Der Algorithmus wird vorbereitet. Dazu wird das KNN und die Gewichte in den Arbeitsspeicher der GPU(s) geladen. 3. Die Verbindungen zu den Kameras werden initiiert. 4. Die Verbindung zum Feldbus wird vorbereitet. 5. Die Detektionsroutine startet, siehe „Anwendungsfall: Fahrradfahrer detektieren“.
Anfangsbedingungen	<ul style="list-style-type: none"> • Eine Stromzufuhr muss vorhanden sein. • Der Algorithmus muss sich im Speicher befinden. • Die Verbindungen zu den Kameras müssen bestehen.
Abschlussbedingungen	<ul style="list-style-type: none"> • Jede verfügbare GPU hat den Algorithmus zum Berechnen von Bildern vorbereitet. • Bilder können von den Kameras erhalten werden. • Ergebnisse können über den Feldbus veröffentlicht werden.
Qualitätsanforderungen	-

Tab. 3.4.: Anwendungsfall: FFDS starten

Anwendungsfall	Fahrradfahrer detektieren
Akteur-Instanz	-
Ereignisfluss	<ol style="list-style-type: none"> 1. Das erste Bild wird in den Arbeitsspeicher geladen. 2. Das FFDS verarbeitet das zuvor geladene Bild und lädt parallel dazu das folgende Bild. 3. Die Daten des Bildes werden zur Verarbeitung in den Arbeitsspeicher einer GPU geladen. 4. Die GPU berechnet die Ergebnisdaten von Fahrradfahrern auf dem Bild. 5. Bei Punkt 2 wird fortgefahren und parallel dazu werden Daten wie folgt veröffentlicht: <ul style="list-style-type: none"> a) Das FFDS veröffentlicht, über einen Feldbus die Ergebnisdaten aller detektierten Fahrradfahrer des zuletzt berechneten Bildes. Bei der Veröffentlichung wird ein eindeutiger Identifikator verwendet, um die Ergebnisdaten einer Kamera zuzuordnen. b) Für den Fall, dass keine Detektionen vorliegen, wird nichts veröffentlicht.
Anfangsbedingungen	<ul style="list-style-type: none"> • Das FFDS ist gestartet. • Das FFDS hat den Algorithmus geladen. • Das FFDS kann Bilder der Kameras erhalten. • Das FFDS kann Daten über den Feldbus veröffentlichen.
Abschlussbedingungen	<ul style="list-style-type: none"> • Ergebnisdaten wurden über einen Feldbus mit korrektem Identifikator veröffentlicht, so dass angemeldete Beobachter diese Daten erhalten können.
Qualitätsanforderungen	<ul style="list-style-type: none"> • Für den Fall mehrerer Kameras sollten alle Kameras gleichviel GPU Zeit erhalten. • Für den Fall mehrerer GPUs sollten alle GPUs gleich stark ausgelastet sein. • In jedem Fall muss der Identifikator für die entsprechende Kamera eindeutig und korrekt sein. • Der Feldbus muss ein Beobachter Entwurfsmuster unterstützen.

Tab. 3.5.: Anwendungsfall: Fahrradfahrer detektieren

Anwendungsfall	Detektionen beobachten
Akteur-Instanz	ADAS
Ereignisfluss	<ol style="list-style-type: none"> 1. Ein ADAS meldet sich als Beobachter am Feldbus für einen Identifikator an. Der Identifikator ist zuständig für die eindeutige Zuordnung von Ergebnisdaten und Kamera. 2. Nach der Anmeldung erhält das ADAS die Ergebnisdaten aller detektierten Fahrradfahrer je Bild, sobald Detektionen ermittelt werden. 3. Das ADAS kann die Ergebnisdaten anschließend verarbeiten.
Anfangsbedingungen	<ul style="list-style-type: none"> • Das ADAS ist aktiv und benötigt Ergebnisdaten zu detektierten Fahrradfahrern auf einer oder mehreren Kameras. • Das FFDS und das ADAS haben Zugang zum gleichen Feldbus.
Abschlussbedingungen	<ul style="list-style-type: none"> • Das ADAS erhält Ergebnisdaten zum angemeldeten Identifikator, sobald neue Daten vorliegen.
Qualitätsanforderungen	<ul style="list-style-type: none"> • In jedem Fall muss der Identifikator für die entsprechende Kamera eindeutig und korrekt sein. • Der Bus muss ein Beobachter Entwurfsmuster unterstützen.

Tab. 3.6.: Anwendungsfall: Detektionen beobachten

3.3.4 Use Case Diagramm

In Abbildung 3.1 sind in einem UML Use Case Diagramm die Anwendungsfälle abgebildet, die das System für ein Fahrzeug, ADAS oder Überwachungssystem zur Verfügung stellt. Dabei handelt es sich um den Anwendungsfall „FFDS starten“, welches vom Fahrzeug, auf dem sich das FFDS befindet, ausgeführt wird. Das Starten des FFDS führt automatisch zum Beginn des Anwendungsfalls „Fahrradfahrer detektieren“. Der dritte aufgeführte Anwendungsfall „Detektionen beobachten“ wird von Fahrerassistenz- oder Überwachungssystemen ausgeführt.

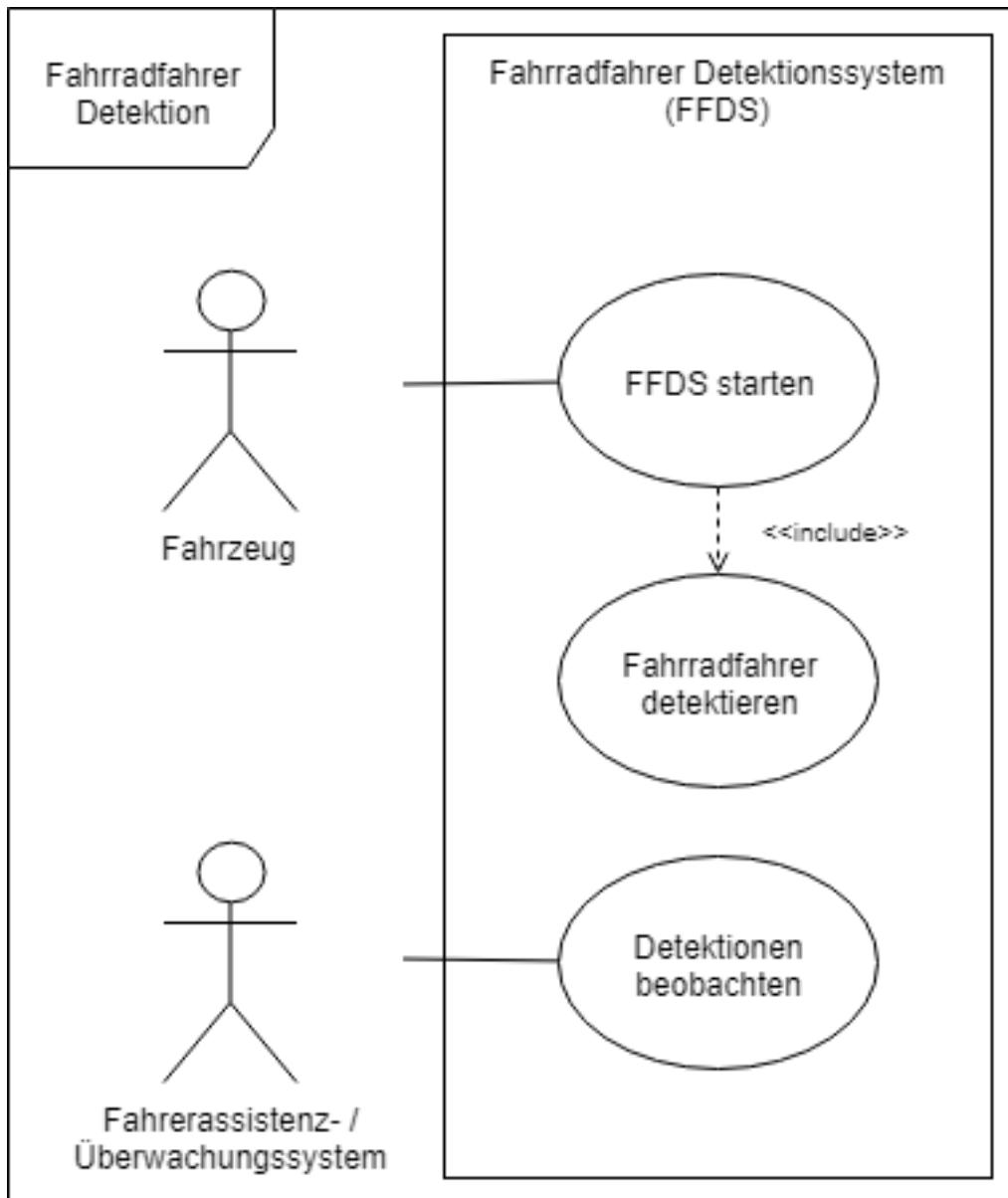


Abb. 3.1.: Anwendungsfälle des Fahrradfahrer Detektionssystem (FFDS). Ein Fahrzeug kann das FFDS starten, woraufhin Fahrradfahrer detektiert werden. Ein Fahrerassistenz- oder Überwachungssystem kann die Detektionen beobachten.

Quelle: Eigene Darstellung

3.4 Anforderungen

„Eine Bedingung oder Fähigkeit, die von einem Benutzer (Person oder System) zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird.“

— Klaus Pohl & Chris Rupp
[PR15]

Dieses Unterkapitel erläutert die Vorgehensweise der Erstellung und Modifizierung des Anforderungskatalogs. Anschließend wird der Anforderungskatalog aufgeführt.

3.4.1 Vorgehensweise

Um die Anforderungen an das System zur Detektion von Fahrradfahrern zu ermitteln, wurden zusätzlich zu der Anforderungsanalyse mit Anwendungsfällen weitere Methoden verwendet: Interviews mit dem Firmenbetreuer, die Inventurmethode bzw. dem perspektiv-basierten Lesen, dem Perspektivenwechsel und Brainstorming. Die gesammelten Anforderungen an das System werden kategorisiert und auf folgende Eigenschaften überprüft: Vollständigkeit, Korrektheit, Klassifizierbarkeit (rechtliche Klarheit), Konsistenz, Prüfbarkeit, Eindeutigkeit, Verständlichkeit, Gültigkeit und Aktualität, Realisierbarkeit, Notwendigkeit, Verfolgbarkeit. Die gesammelten und geprüften Anforderungen werden durchnummeriert in diesem Kapitel beschrieben. Die Anforderungen haben eine relativ grobe Granularität, da diese Arbeit keinen Projektplan darstellt. Die Anforderungen werden zu einem späteren Zeitpunkt überprüft und falls nötig geändert. Auf ein spezielles Änderungsverfahren wird im Rahmen dieser Arbeit verzichtet. Die im Folgenden niedergeschriebenen Anforderungen sind jeweils final.

3.4.2 Anforderungskatalog

Im Anforderungskatalog sind lediglich die Anforderungen nach funktionalen und nicht-funktionalen Anforderungen aufgeteilt und durchnummieriert. Die Anforderungsschablone nach Chris Rupp [Rup06, S. 227 f.] wird zur Formulierung von Anforderungen verwendet:

[Wann? Unter welcher Bedingung] [muss, sollte, wird]
das System [-, Wem? die Möglichkeit bieten, Fähig sein]
[Objekt und Ergänzung des Objekts] [Prozesswort].

Die funktionalen Anforderungen sind:

- Vorbereitung:
 1. Vor Inbetriebnahme muss das System einen Mikrocontroller enthalten.
 2. Vor Inbetriebnahme muss das System eine Kamera enthalten.
 3. Vor Inbetriebnahme muss das System das KNN enthalten.
 4. Vor Inbetriebnahme muss das System die Software im Mikrocontroller enthalten.
 5. Vor Inbetriebnahme muss das System die Gewichte für das KNN enthalten.
 6. Vor Inbetriebnahme muss das System den Mikrocontroller mit der Kamera verbinden.
- Systemstart:
 7. Sobald Strom am Mikrocontroller anliegt, muss das System starten (Systemstart).
 8. Beim Systemstart muss das System die Kamera mit Strom versorgen.
 9. Beim Systemstart muss das System die Treiber der Kamera laden.
 10. Beim Systemstart muss das System die Software starten (Softwareinitialisierung).
 11. Bei der Softwareinitialisierung muss das System eine Verbindung zur Kamera aufbauen, um Bilder abrufen zu können.

12. Bei der Softwareinitialisierung muss das System die Verbindung zum Feldbus initialisieren, um die Ergebnisse senden zu können.
13. Bei der Softwareinitialisierung und nachdem die Verbindung zur Kamera aufgebaut ist, muss das System das erste Bild der Kamera abrufen.
14. Bei der Softwareinitialisierung muss das System das KNN in den Arbeitsspeicher (der GPU) laden.
15. Bei der Softwareinitialisierung und nachdem das KNN geladen wurde muss das System die Gewichte für das KNN in den Arbeitsspeicher (der GPU) laden.
16. Nach der Softwareinitialisierung ist das System bereit, den Algorithmus zu verwenden, also das Detektieren von Fahrradfahrern auf Bildern.
 - Betriebsmodus
17. Nach dem die Softwareinitialisierung abgeschlossen ist, muss das System den Betriebsmodus starten.
18. Im Betriebsmodus und parallel zum Verarbeiten der Bilddaten muss das System fortwährend Bilder von der Kamera abrufen.
19. Im Betriebsmodus und parallel zum Abrufen der Bilder von der Kamera muss das System die Bilddaten in den Arbeitsspeicher (der GPU) laden.
20. Nach dem Laden der Bilddaten muss das System den Algorithmus verwenden, um Fahrradfahrer auf dem Bild zu detektieren.
21. Nachdem der Algorithmus ein Bild berechnet hat, muss das System die Ergebnisdaten aller detektierten Fahrradfahrer über einen Feldbus veröffentlichen.
22. Die Veröffentlichung muss mit einem eindeutigen Identifikator geschehen, welcher die Kamera und die Ergebnisdaten zuordnet.

Die nicht funktionale Anforderungen sind:

24. Nachdem das KNN trainiert wurde, muss die Software das KNN mit mindestens 10 000 Bildern testen, auf denen Fahrradfahrer abgebildet sind (Softwaretest).
25. Bei dem Softwaretest soll das KNN eine Trefferrate von 95% erreichen.
26. Bei dem Softwaretest sollen die vom KNN detektierten Positionen von Fahrradfahrern auf Bildern zu mindestens 75% mit der tatsächlichen Position übereinstimmen.
27. Bei dem Softwaretest soll das KNN eine Präzision von 95% aufweisen.
28. Das System soll mindestens 10 Bilder pro Sekunde analysieren können, dies gilt nur für vorhandene GPU Unterstützung.
29. Nach Abschluss des Softwaretests soll das System im eingebetteten System getestet werden, wobei hauptsächlich die Bilder pro Sekunde gemessen werden und nur stichprobenartig, ob ein Fahrradfahrer erkannt wurde.

Lösungsansatz

In diesem Kapitel wird zunächst allgemein beschrieben wie die Vorgehensweise ist, um das Fahrradfahrer Detektionssystem herzustellen. Anschließend werden in den Unterkapiteln die Methoden und Technologien verdeutlicht.

Der Kern des FFDS ist ein künstliches neuronales Netzwerk, welches in ein eingebettetes System integriert wird. Im Rahmen dieser Arbeit wird keinen neuen KNN Architekturen nachgegangen. Ein Objekt Detektor, welcher den Anforderungen aus Kapitel 3 entspricht, wird als KNN ausgewählt. Dieses KNN muss modifiziert und trainiert werden, damit Fahrradfahrer detektiert werden können. Der hierfür gewählte Ansatz nennt sich Transfer Learning, vgl. Pan und Yang [PY10]. Dazu werden die Schichten des Netzwerkes, welche zur Klassifizierung und Lokalisierung zuständig sind, neu trainiert. Die vorigen Schichten, welche für die Merkmalsextraktion zuständig sind, werden mit den vortrainierten Gewichten übernommen. Das ausgewählte Netzwerk muss mit entsprechenden Daten trainiert werden, um Fahrradfahrer auf Bildern detektieren zu können. Für das Training des Netzwerkes wird ein Datensatz mit Datenpaaren von Bildern mit Fahrradfahrern und zugehörigen Markierungen erstellt. Aus dem komplettem Datensatz werden Validierungsdaten ausgewählt und dazu verwendet, passende Trainingsvariablen zu finden. Nach Trainingsende werden die Gewichte, die verwendeten Daten und die Trainingsvariablen gespeichert. Das KNN wird dann auf Mikrocontrollern integriert und erhält Bilddaten von einer Kamera. Das FFDS wird schließlich auf dem VeloxCar integriert, dort werden auch Bilder einer Kamera empfangen und verarbeitet. Zusätzlich sendet das FFDS die Ausgabedaten an einen Feldbus im Mikrocontroller, damit die Daten von anderen Systemen abrufbar sind.

4.1 Das verwendete künstliche neuronale Netzwerk

Die Auswahl des künstlichen neuronalen Netzwerks für den Zweck dieser Arbeit wird maßgeblich durch Redmon und Farhadi [RF18, S. 4] beeinflusst. Redmon und Farhadi vergleichen mehrere Detektoren, welche dem aktuellen Stand der Technik entsprechen. Im März 2018 wurde der Objekt Detektor YOLOv3 von Redmon und Farhadi veröffentlicht, vgl. [RF18]. YOLOv3 wir von einigen sogenannten Merkmal Pyramiden Netzwerken (engl.: Feature Pyramid Network (FPN)) zumindest in Sachen Genauigkeit übertrffen. YOLOv3 ist bei 1,2% schlechterer Genauigkeit jedoch mehr als dreimal so schnell im Vergleich zu FPN, siehe Tabelle 4.1. Der

Netzwerk	Genauigkeit in %	Zeit pro Bild in ms
FPN	59,1	172
YOLOv3	57,9	51
YOLOv3-tiny	33,1%	5

Tab. 4.1.: Gegenüberstellung der Genauigkeit in Prozent und der Schnelligkeit in ms von Objekt Detektoren. Verglichen werden ein Objekt Detektor mit Merkmals Pyramiden (FPN), YOLOv3 und YOLOv3-tiny. Der FPN hat die höchste Genauigkeit und YOLOv3-tiny ist der schnellste Objekt Detektor.

Quellen: [Red18] und [RF18, S. 4]

geringe Unterschied der Genauigkeit sowie die Anforderung, 10 Bilder pro Sekunde zu verarbeiten, sind ausschlaggebend für die Auswahl von YOLOv3 als KNN für diese Arbeit. Der Zeitaspekt in Kombination mit eingeschränkten System Ressourcen im eingebetteten System führt dazu, dass zusätzlich das YOLOv3-tiny Netzwerk verwendet wird. Die Präzision von YOLOv3-tiny ist deutlich schlechter, aber das Netzwerk ist zehnmal schneller, siehe Tabelle 4.1. Zusätzlich ist YOLOv3-tiny deutlich kleiner in Bezug auf Netzwerktiefe (YOLOv3-tiny 24 Schichten zu YOLOv3 107 Schichten) und Gewichte (YOLOv3-tiny 34 MB zu YOLOv3 250 MB), vgl. [Red18].

YOLO ist ein Akronym und steht für „You Only Look Once“. Der Objekt Detektor ist ein tiefes CNN, da fast ausschließlich Faltungsschichten verwendet werden. Die hier genutzte dritte Version verwendet zusätzlich Rückstandsschichten; Schichten, die die Ausgabe zweier Schichten verketten und Schichten, die hochskalieren. Während andere Architekturen zufällige Bildbereiche wählen, um Objekte zu erkennen, teilt YOLO das Bild in gleich große Teile auf. Die Bilder werden vor der Bearbeitung des Netzwerks auf eine bestimmbare Auflösung skaliert. Damit das Seitenverhältnis der Bilder erhalten bleibt, werden die Bilddaten ggf. mit schwarzen Pixeln (Nullen) aufgefüllt.

YOLOv3 verwendet state-of-the-art Technologien, unter anderem:

- Die Eliminierung von nicht maximalen Ergebnissen (engl.: Non Maximum Supression) wird verwendet. Das erhöht die Präzision um 2-3%, vgl. Redmon et al. [Red+16, S. 4].
- Die Normierung der Eingabedaten (engl.: Batch Normalization). Die Normierung erhöht die Konvergenzgeschwindigkeit während des Trainings, erhöht die durchschnittliche Präzision um 2% und dient der Regulierung. Letzteres wirkt der Überanpassung entgegen, vgl. Redmon und Farhadi [RF16, S. 2].
- Die Verwendung von Ankerrahmen (engl.: anchor boxes). Die Begrenzungsrahmen werden relativ zu den Ankerrahmen bestimmt, vgl. Redmon und Farhadi [RF16, S. 2].
- Die Durchführung des Trainings in verschiedenen Skalierungen. Durch diese Methode kann nach dem Training die Auflösung gewechselt werden, jedoch in Relation zur Performance, vgl. Redmon und Farhadi [RF16, S. 3].
- Die Verwendung des FPN Ansatz für die Detektion in verschiedenen Skalierungen, vgl. Redmon und Farhadi [RF18, S. 2].
- Die Vermehrung der Daten (engl.: Data Augmentation) verändert Bilder in Farbe und Helligkeit. Die Vermehrung der Daten wird in Kombination mit zufälligen Bildausschnitten verwendet, damit das Training weniger anfällig gegenüber Überanpassung ist und eine höhere Trainingsdatenmenge vorhanden ist, vgl. Redmon und Farhadi [RF18, S. 3].

Der Ablauf des gesamten YOLOv3 Netzwerks ist in Abbildung 4.1 zu sehen. In Abbildung 4.1 ist links oben das eingegebene Bild dargestellt. Das Bild wird vor der Eingabe in das Netzwerk auf die vom Netzwerk erwartete Größe skaliert. Dabei wird das Seitenverhältnis beibehalten und ggf. die Ränder mit schwarzen Pixeln aufgefüllt - das Verfahren wird Letterbox genannt. Zuerst werden für die Merkmalsextraktion ausschließlich Faltungsschichten und Rückstandsschichten durchlaufen, diese sind in der Abbildung als Residual Block dargestellt. Die ersten 75 Schichten sind für die Merkmalsextraktion zuständig. Anschließend wird die Detektion großer Objekte (Scale 1) vorgenommen. Das Bild ist zu diesem Zeitpunkt in 32 x 32 Pixel große Bereiche eingeteilt, als Beispiel wäre ein 608 x 608 Pixel großes Bild in 19 x 19 Bildbereiche á 32 x 32 Pixeln aufgeteilt. Die Detektion großer Objekte ist in der 83. Schicht abgeschlossen. Die 83. Schicht ist eine yolo-Schicht und enthält Detektionen. Detektionen enthalten für jeden Bildbereich drei Begrenzungsrahmen. Zu einem Begrenzungsrahmen gehören die Höhe, Breite, X-, Y-Koordinate und Wahrscheinlichkeit.

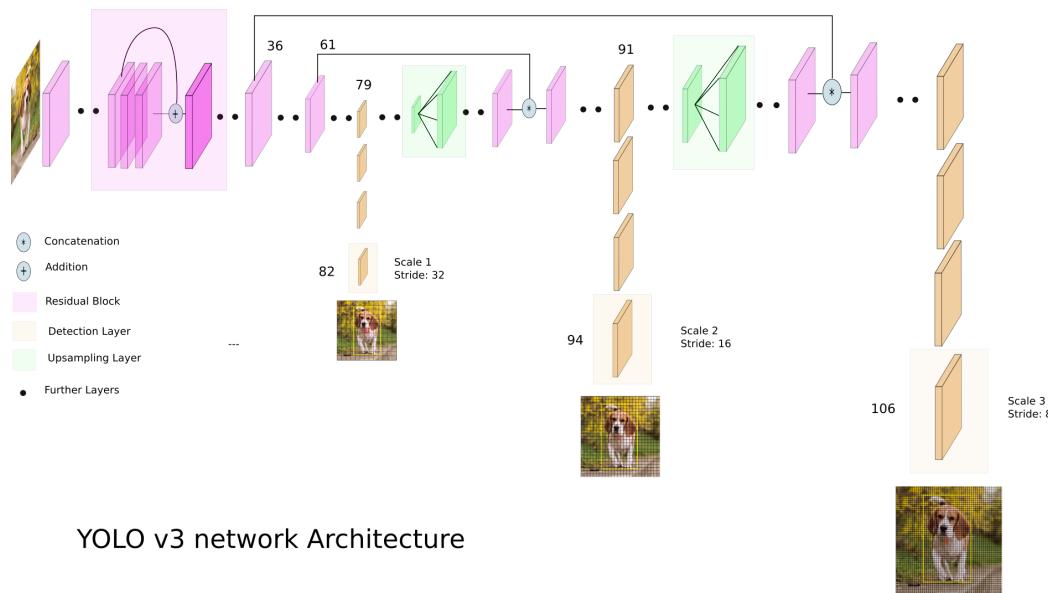


Abb. 4.1.: Architektur des Objekt Detektors YOLOv3. Die unterschiedlichen Aufgaben der Faltungsschichten sind farblich gekennzeichnet. Gut erkennbar sind die Schichten, die für die Merkmals Pyramiden verwendet werden und die drei Ausgabeschichten. Die Nummerierung der Schichten beginnt mit 0.

Quelle: <https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b>

keiten. Die Wahrscheinlichkeiten sind zum einen ein genereller Wert, der angibt ob sich tatsächlich ein Objekt in diesem Begrenzungsrahmen befindet, also Objekthaf tigkeit (engl.: objectness). Zum anderen wird für jede Klasse eine Wahrscheinlichkeit berechnet. Für unser Beispielbild ergibt sich für die Fahrradfahrerdetektion:

Bildbereiche*Begrenzungsrahmen*(Höhe, Breite, Koordinaten, Objekthaftigkeit,
Fahrradfahrerklasse)= $19 * 19 * 3 * 6$

Wichtig anzumerken sei hier, dass die Nummerierung der Schichten auf dem Bild bei 0 anfängt, deswegen sind die im Text genannten Zahlen jeweils eins höher. Um kleinere Objekte zu erkennen wird im Folgenden die Methode der Merkmals Pyramiden durchgeführt. Dazu werden die extrahierten Merkmale tieferer Schichten verwendet, um auf einer höheren Auflösung Objekte zu erkennen. Dazu werden die tieferen Schichten mit einer früheren Schicht, auf der die Auflösung höher ist, verkettet. Die 80. Schicht wird erst hochskaliert, damit sie mit der 62. Schicht verkettet werden kann. Diese Verkettung durchläuft sieben Faltungsschichten, um die Detektion mittlerer Objekte (Scale 2) zu berechnen. Die 95. Schicht ist wieder eine Ausgabeschicht und eine yolo-Schicht, diesmal mit Bildbereichen der Größe 16 x 16 Pixel. Zuletzt wird die 92. Schicht hochskaliert und mit der 37. verkettet. Dann werden wieder sieben Faltungsschichten durchlaufen, um kleinen Objekte (Scale 3) zu detektieren. Diese yolo-Schicht ist die 107. und die letzte Ausgabeschicht. Das Bild wird diesmal in 8 x 8 Pixel große Bereiche eingeteilt.

Da der Transfer Learning Ansatz verfolgt wird, werden die Gewichte für die Schichten benötigt, die für die Merkmalsextraktion zuständig und bereits trainiert sind. Die Merkmalsextraktionsschichten sind bei YOLOv3 die ersten 74 Schichten und bei YOLOv3-tiny die ersten 15 Schichten. Die Gewichte der Merkmalsextraktionschichten für YOLOv3 werden separat angeboten, als Datei mit dem Namen Darknet53.conv.74. Die Gewichte für die Merkmalsextraktionsschichten werden für das YOLOv3-tiny aus den für den COCO Datensatz fertig trainierten Gewichten extrahiert. Dazu wird das nachfolgend vorgestellte Framework Darknet verwendet.

4.1.1 Darknet Framework

YOLOv3 wurde ursprünglich mit dem quelloffenen Framework Darknet von Joseph Redmon entwickelt. Darknet stellt Bibliotheken zur Verfügung, um ohne viel Aufwand ein KNN zu definieren, trainieren und zu testen. Darknet wurde im Jahr 2013 veröffentlicht, vgl. Redmon [Red16]. Darknet ist in C, C++ und CUDA geschrieben und bietet die Möglichkeit die Bibliotheken CUDA, CUDNN und OpenCV zu verwenden. Im Rahmen dieser Arbeit wird von all diesen Optionen Gebrauch gemacht. CUDA ist eine NVIDIA Architektur für parallele Berechnungen und ebenso eine Bibliothek, um diese Architektur zu nutzen. Die NVIDIA CUDA Deep Neural Network-Bibliothek (cuDNN) ist eine Bibliothek mit GPU Unterstützung für Grundelemente von tiefen neuronalen Netzwerken. Mit diesen Bibliotheken kann in den Arbeitsspeicher der GPU ein- und ausgelesen werden und die GPU für Berechnungen verwendet werden. Im Vergleich zu Keras - ein anderes Framework für KNN - scheint die GPU besser verwendet zu werden, wobei diese Aussage nur auf Beobachtung mit dem Tool nvidia-smi beruht: 1-30% GPU Auslastung mit Keras im Gegensatz zu >90% GPU Auslastung mit Darknet. Ein weiterer Vorteil von Darknet ist, dass es quelloffen ist. Dadurch können beliebige Schichten, Algorithmen oder Anwendungen hinzugefügt werden. Zusätzlich ist C eine Programmiersprache, welche die Portabilität erhöht. Aus den genannten Gründen wird das Darknet Framework im Rahmen dieser Arbeit verwendet. Für die Extrahierung der vortrainierten Gewichte steht folgende Methode in Darknet zur Verfügung:

```
darknet partial YOL0v3-tiny.cfg yolov3-tiny.weights Ausgabedatei 15
```

Der Aufruf ist wie folgt aufgebaut. Mit darknet wird das Programm gestartet, auf einem Windows Rechner wäre es entsprechend darknet.exe. Die Methode zum Extrahieren der Gewichte wird mit partial aufgerufen. In der Datei YOL0v3-tiny .cfg ist die Struktur des KNN beschrieben. In der Datei yolov3-tiny.weights befinden sich die vortrainierten Gewichte. Der Parameter Ausgabedatei beschreibt, den Pfad wohin die Gewichte als binär Datei gespeichert werden. Die Zahl am Ende beschreibt, bis zu welcher Schicht die Gewichte extrahiert werden sollen.

4.2 Datenbasis

Als Basis für das Training des KNN dient ein Datensatz, der sich aus mehreren Quellen zusammensetzt: YouTube Videos, selbst aufgenommene Videos und Daten aus dem Tsinghua-Daimler Cyclist Benchmark von Xiaofei Li et al. [Li+16]. Als wichtige Kriterien für die Auswahl der Daten dienen zum einen das notwendige Vorhandensein von Markierungen für alle erkennbaren Fahrradfahrer. Zum anderen dürfen ausschließlich Fahrradfahrer markiert sein. Die Markierungen sind in separaten Dateien abgelegt und müssen eindeutig einem Bild zuordenbar sein. Diese Kriterien sind ausschlaggebend, um die Ziele der hohen Präzision und der niedrigen Fehlerquote zu erreichen. Aus den oben genannten Quellen ergibt sich ein Datensatz mit 30 773 Bildern, wobei 67% (20 651 Bilder) für das Training und die restlichen 33% (10 122 Bilder) zum Testen verwendet werden. Für das Validierungsset wurden 999 zufällig ausgewählte Bilder verwendet, jeweils 333 aus jeder der drei Quellen.

4.2.1 Tsinghua-Daimler Cyclist Benchmark

Daimler Benz veröffentlichte im Jahr 2016 in Zusammenarbeit mit der Universität Tsinghua in Beijing einen Datensatz, vgl. [Li+16]. Mit einer Frontkamera wurden aus einem fahrenden Auto Bilder aufgenommen. Die Aufnahmen wurden von November 2014 bis März 2015 erstellt. Anschließend wurden auf jedem Bild Fahrradfahrer mit Begrenzungsrahmen markiert. Diese Daten bestehen aus insgesamt über 30 000 Bildern, wie in Tabelle 4.2 zu sehen ist. Die Bilder haben eine Auflösung von 2048 x 1024 Pixeln. Das Set „Ohne“ beschreibt Daten, auf denen kein Fahrradfahrer zu sehen ist (Anm.: Das heruntergeladene Ohne-Set besteht nur aus 994 Bildern). Für das verwendete überwachte Lernen sind nur markierte Bilder zulässig. Der Datensatz wird einer manuellen Qualitätsprüfung unterzogen. Während der Überprüfung der markierten Bilder fällt auf, dass nicht alle Fahrradfahrer markiert sind, wie z.B. in der Abbildung 4.2 zu sehen ist. Alle Bilder, welche die Auswahlkriterien erfüllen, werden in den Datensatz übernommen, diese sind in der Tabelle 4.2 in der Zeile „Verwendete Bilder“ aufgelistet. Die Validierungs- und Testdaten zählen zu den 10 122 Testdaten. Die Trainingsdaten und „Ohne-Daten“ werden für das Training verwendet.

Die Markierungsdateien liegen in diesem Datensatz im JSON Format vor. Mit einem selbst geschriebenen Java Programm werden die Informationen der Begrenzungsrahmen in das von Darknet benötigte Format überführt, siehe im Anhang Listing B.1.



Abb. 4.2.: Beispielbild aus dem Tsinghua-Daimler Cyclist Benchmark, auf dem im linken Bildbereich eindeutig ein Fahrradfahrer zu sehen ist. Für diesen Fahrradfahrer ist jedoch kein Begrenzungsrahmen vorhanden (rote Markierung). Auf der rechten Seite des Bildes ist ein korrekt markierter Fahrradfahrer zu sehen (grüne Markierung).

Quelle: Adaptiert von [Li+16]

	Anzahl Daten				
	Training	Validierung	Test	Ohne	Insgesamt
Alle Bilder	9741	5095	14570	1000	30406
Markierte Bilder	9741	1019	2914	1000	14674
Verwendete Bilder	6657	1019	2914	994	11584
Fahrradfahrer	16202	1314	4657	0	22173

Tab. 4.2.: Datensatz Statistik des Tsinghua-Daimler Cyclist Benchmark. Die Statistik wurde übersetzt und erweitert. Abgebildet sind die Anzahl der Bilder in den einzelnen Datensätze und die Anzahl der Verwendeten.

Quelle: Adaptiert von [Li+16]

4.2.2 Selbst aufgenommene Videos und YouTube Videos

Im Rahmen dieser Arbeit werden zusätzlich Videos genutzt, die selbst aufgenommen wurden, sowie passende YouTube Videos. Selbst gefilmt wurde mit Hilfe eines Smartphones, welches im Frontbereich eines Fahrrads befestigt wurde. Aufgenommen wurde der Verkehr der Berliner Innenstadt und Fahrten durch den Tiergarten. Die Aufnahmen fanden im Herbst 2018 statt. Die Auflösung der Kamera ist 720 x 1280 Pixel. Bei den YouTube Videos handelt es sich um zwei Videos, bei denen Fahrradfahrer in unterschiedlichen Verkehrssituationen zu sehen sind. Die Videos heißen „HERE ARE 45 EXAMPLES OF WHY CYCLISTS ARE DISLIKED“ und „HERE ARE 40 EXAMPLES OF WHY CYCLISTS ARE DISLIKED“. Beide Videos sind nicht mehr auf YouTube verfügbar, aber werden der Arbeit beigelegt. Die Auflösung der Videobilder ist 1280 x 720 Pixel.

Die Videos werden verwendet, um Datenpaare automatisch zu generieren. Erzeugt werden einzelne Bilder, auf denen Fahrradfahrer erkannt werden, sowie die zugehörigen Informationen der Begrenzungsrahmen in Markierungsdateien. Der Algorithmus zur automatischen Generierung der Daten aus Videos verwendet YOLOv3 mit vortrainierten Gewichten und Darknet. Mit YOLOv3 werden Personen und Fahrräder detektiert. Anschließend wird ein selbst geschriebener Algorithmus verwendet, um Personen und Fahrräder zu Fahrradfahrern zusammenzufassen. Der Algorithmus stellt lediglich fest, ob sich die Person im Bild höher befindet als das Fahrrad, und welche Personen sich mit welchen Fahrrädern am meisten überschneiden, siehe im Anhang Listing B.2. Mit dieser Logik werden die meisten Personen ausgeschlossen, die keine Fahrradfahrer sind. Dieser Algorithmus wird in Darknet in C implementiert und kann auch mit anderen Detektoren verwendet werden.

Die erzeugten Daten erweisen sich teilweise als problematisch. Zum einen, weil einige Fahrradfahrer nicht markiert sind, da sie nicht detektiert werden. Zum anderen, weil Menschen, die hinter Fahrrädern stehen, als Fahrradfahrer erkannt werden. In Tabelle 4.3 ist zu sehen, wie viele Bilder mit Fahrradfahrern erfasst werden und auf wie vielen Bildern alle Fahrradfahrer korrekt markiert sind.

	Smartphone	YouTube
Alle Bilder	33 983	6 301
Verwendete Bilder	15 563 (45,7%)	3 626 (57,5%)

Tab. 4.3.: Anzahl an automatisch generierten Daten und der Anzahl der genutzten Daten nach der Aussortierung.

4.2.3 Datensatz Bereinigung

Die Bereinigung der Daten erfolgt manuell. Jedes Bild wird auf die folgenden Auswahlkriterien überprüft:

1. Jeder Fahrradfahrer muss markiert sein. Eine zulässige Ausnahme tritt auf, wenn sich zwei Fahrradfahrer komplett überschneiden und nur ein Begrenzungsrahmen vorhanden ist.
2. Nur Fahrradfahrer dürfen als Fahrradfahrer markiert sein, also z.B. keine Mopeds oder Menschen, die ein Fahrrad schieben.
3. Der Begrenzungsrahmen muss größtenteils korrekt sein. Da die Anforderung dafür niedrig ist und der Aufwand für eine exakte Ermittlung sehr hoch wäre, wird hier subjektiv ausgewählt.

Für die Bereinigung wird eine Funktion in C geschrieben, um nur die korrekten und vollständigen Daten für den Datensatz zu verwenden. Die Funktion wird in C mit OpenCV geschrieben und erhält als Parameter das Verzeichnis, in dem sich die zu prüfenden Daten befinden, sowie ein Offset. Letzteres dient dazu nach einer Unterbrechung die Bearbeitung ab einer gewissen Stelle fortsetzen zu können. Jeweils ein Bild und die dazugehörige Markierungsdatei werden eingelesen. Das Bild wird inklusive der Begrenzungsrahmen aus der Markierungsdatei angezeigt. Mittels Tastatureingaben kann durch die Bilder eines Verzeichnisses navigiert und entschieden werden, ob die Qualität der Markierungen den Auswahlkriterien entspricht. Die Dateinamen der Bilder werden entsprechend der Tastatureingabe in der Datei gute.txt oder schlechte.txt gespeichert. Die Funktion wurde folgendermaßen verwendet: Während der Bereinigung der Daten werden alle Bilder, bei denen alle Auswahlkriterien zutreffen, als gut markiert. Bei einer versehentlichen fehlerhaften Eingabe wird das Bild als schlecht markiert. Die Differenz der beiden Dateien ergibt dann die Dateinamen aller Bilder, deren Daten den Auswahlkriterien entsprechen.

4.3 Training

Um ein Training durchzuführen, werden zunächst Trainingsdaten vorbereitet, daraufhin wird das KNN trainiert und schließlich findet eine Auswertung des Trainings statt. Dementsprechend ist dieses Kapitel in drei Teile gegliedert: im ersten Schritt werden die für das Training notwendigen Daten genauer inspiziert. Im Anschluss wird beschrieben, wie das Training gestartet und der Trainingsverlauf beobachtet wird. Abschließend werden das Ende des Trainings und die Schritte in der Nachbearbeitung erläutert.

4.3.1 Trainingsdaten

Der erste wesentliche Schritt eines Trainings liegt in der Vorbereitung der Trainingsdaten. Dieses Unterkapitel befasst sich damit, wie die Dateien und Datensätze für das Framework Darknet aufgebaut sind. Zum Trainieren mit Transfer Learning wird die Architektur des Objekt Detektors, die Trainingsvariablen, ein Datensatz und vortrainierte Gewichte benötigt. Die Architektur und die Trainingsvariablen befinden sich bei Darknet zusammen in einer Konfigurationsdatei, welche nachfolgend beschrieben wird. Anschließend wird der Aufbau des Datensatzes erläutert und die Datensatzdatei erklärt. Die Gewichte befinden sich in einer binären Datei und stehen bereits zur Verfügung, wie in Kapitel 4.1 gezeigt. Darin befinden sich lediglich die Gewichte für den Objekt Detektor.

Konfigurationsdatei

Die Konfigurationsdatei dient der Beschreibung des verwendeten künstlichen Netzwerks. In der Konfigurationsdatei befinden sich allgemeine Netzwerkgrößen, Trainingsvariablen und die Struktur des Netzwerks. Die meisten Größen erhalten von Darknet einen Standardwert, wenn sie nicht explizit angegeben werden. Ein Auszug der verwendeten Konfigurationsdatei `ffds.cfg` ist im Anhang als Listing B.11 aufgeführt. Die vollständige Datei wird der Arbeit beigefügt und befindet sich zusätzlich in einem Git Repository im gitLab der Beuth-Hochschule unter <https://gitlab.beuth-hochschule.de/s72834/ffds>. Die Datei ist in Abschnitte eingeteilt, die mit eckigen Klammern eingeleitet werden. Der erste Abschnitt beginnt mit `[net]` und enthält allgemeine Netzwerkgrößen und Trainingsvariablen. Die Tabelle 4.4 enthält eine Auflistung aller vom Fahrradfahrer Detektor verwendeten Trainingsvariablen mit einer kurzen Beschreibung wofür sie verantwortlich sind. Dabei ist auffällig, dass ein Teil für die Anpassung der Gewichte verantwortlich ist und ein anderer für die Vermehrung der Daten. Die Vermehrung von Daten ist eine Methode, um den vorhandenen Datensatz so zu verändern, dass die Daten zwar immer noch sehr ähnlich, aber nicht mehr gleich sind. Fahrradfahrer sind sowohl auf den ursprünglichen Bildern erkennbar als auch auf den in Blickwinkel, Farnton, Helligkeit und Sättigung geänderten Bildern. Die Vermehrung von Daten ist ein weiteres Mittel, um gegen Überanpassung anzugehen, wie Perez und Wang beschreiben, vgl. [WP17, S.2]. Drei allgemeine Variablen werden für das Netzwerk bestimmt: Die Anzahl der Farbkanäle, sowie die Höhe und die Breite, auf die die Bilder skaliert werden. Höhe und Breite sollte identisch und durch 32 teilbar sein, weil das Bild in 32 x 32 Pixel große Bildbereiche aufgeteilt wird.

Variable	Beschreibung
batch	Das Training mit Mini-Batches wird unterstützt und die Größe eines Batch kann angegeben werden. Das Netzwerk verarbeitet immer diese Anzahl an Bildern für die Berechnung des Fehlers bzw. der Anpassung der Gewichte, deswegen sollte diese Zahl für den Betrieb auf eins reduziert werden.
subdivisions	Die Batch Größe wird durch die Zahl der subdivisions geteilt. Das Ergebnis muss eine ganze Zahl sein und stellt die Anzahl der Bilder dar, die gleichzeitig verarbeitet werden. Die Anzahl hängt von der skalierten Größe der Bilder und dem Arbeitsspeicher der GPU ab.
momentum	Momentum wird verwendet, um die „Richtung“ der letzten Anpassung beim Aktualisieren der Gewichte miteinzubeziehen.
decay	Die Gewichtszerfall Variable namens decay reguliert die Gewichte in Relation zu ihrer Größe zusätzlich zur Lernrate.
learning_rate	Learning_rate, Die Lernrate, gibt an, wie stark die Gewichte nach einer Iteration angepasst werden.
policy	Policy beschreibt die Art und Weise, in der die Lernrate angepasst wird. Eine Möglichkeit ist „steps“, welche die beiden nachfolgenden Variablen steps und scales benötigt.
steps	Für policy=steps, wird die Lernrate nach einer bestimmten Anzahl von Iterationen angepasst. Die Variable steps enthält eine kommaseparierten Liste mit je einer Anzahl von Iterationen.
scales	Scales enthält für jede Zahl in steps in einer kommaseparierten Liste eine Gleitkommazahl. Die Zahlen sind der Faktor, mit dem die Lernrate multipliziert wird.
angle	Die maximale Gradzahl angle bestimmt wie weit Bilder zufällig nach links oder rechts gedreht werden dürfen. Angle wird für die Vermehrung von Daten verwendet.
saturation	Saturation gibt den maximalen Wert für die Änderung der Sättigung eines Bildes an. Saturation wird für die Vermehrung von Daten verwendet.
exposure	Exposure gibt den maximalen Wert für die Änderung der Helligkeit eines Bildes an. Exposure wird für die Vermehrung von Daten verwendet.
hue	Hue gibt den maximalen Wert für die Änderung des Farbtons eines Bildes an. Hue wird für die Vermehrung von Daten verwendet.

Tab. 4.4.: Auflistung und Erläuterung der Trainingsvariablen in Darknet

Nachdem der allgemeine Netzwerkabschnitt der Konfigurationsdatei beschrieben wurde, folgt die Beschreibung der einzelnen Schichten des Netzwerks. Darknet bietet eine Vielzahl von Schichten an und kann einfach um eigene Schichten erweitert werden. Im Rahmen dieser Arbeit ist das Einführen einer neuen Schicht nicht nötig. Erläutert werden nur die Schichten, die von YOLOv3 verwendet werden. Jede Schicht kann eine individuelle Aktivierungsfunktion enthalten. In der Variable activation wird bestimmt, welche Aktivierungsfunktion verwendet werden soll. Im verwendeten Netzwerk werden nur zwei Aktivierungsfunktionen genutzt:

linear ($f(x) = x$) und leaky ($f(x) = x$, wenn $x > 0$ ansonsten $f(x) = x/10$).

Das verwendete Netzwerk enthält nur vier unterschiedliche Schichten: route, shortcut, convolutional und yolo. Die route-Schicht verkettet die Ausgabe von mehreren Schichten miteinander, wobei die Größen der Schichten übereinstimmen müssen. Dieser Schicht wird über die Variable layers mitgeteilt welche Schichten miteinander verkettet werden sollen. Wenn nur eine Schicht angegeben wird, erhält die folgende Schicht die Ausgabedaten der angegebenen Schicht. Sie besitzt keine weitere Variable. Entsprechend auch keine Aktivierungsfunktion. Die shortcut-Schicht entspricht in Darknet der Rückstandsschicht. Die Rückstandsschicht addiert zu ihren Eingabedaten die Ausgabedaten einer weiteren Schicht. Die shortcut-Schicht besitzt die Variable für die Aktivierungsfunktion und die Variable from. Letztere gibt die Schicht an, von der die Ausgabedaten addiert werden sollen. Die Funktionsweise einer Faltungsschicht (convolutional) wurde bereits in Kapitel 2.3 beschrieben. Die Variablen für eine Faltungsschicht sind die bekannten: Anzahl der Filter, Schrittweite, Größe des Faltungskerns und Zero-Padding. Zusätzlich kann in jeder Faltungsschicht die Normierung der Ausgabedaten verwendet werden, diese Technik wurde im Jahr 2015 von Ioffe und Szegedy vorgestellt, vgl. [IS15]. YOLOv3 verwendet die Normierung der Eingabedaten in jeder Faltungsschicht.

Die yolo-schicht ist zuständig, um die vorhergesagten Begrenzungsrahmen und Klassen zu überprüfen und die wahrscheinlichsten auszuwählen. Zusätzlich wird der Begrenzungsrahmen in Relation zu den Ankerrahmen gesetzt. Im verwendeten Netzwerk werden drei yolo-Schichten für drei unterschiedliche Größen verwendet, um insbesondere kleinere Objekte zu detektieren. Nachfolgend werden die Variablen der yolo-Schicht erklärt und die eingestellten Werte genannt. Die yolo-Schicht enthält zwei Trainingsvariablen: erstens jitter, um die Bilder beim Training zufällig zu verschieben; zweitens random zum Bestimmen, ob die Bilder unterschiedlich skaliert werden sollen. YOLOv3 verwendet ein jitter von 0,3 und die Skalierung auf unterschiedliche Größen. In der yolo-Schicht wird definiert, wie viele Klassen detektiert werden sollen. Die Anzahl der Klassen steht in der Variable classes, für das FFDS ist das nur eine. Zwei Variablen (ignore_tresh und truth_tresh) werden zur Überprüfung verwendet, ob ein berechneter Begrenzungsrahmen ei-

nem korrektem Begrenzungsrahmen um ein gewissen Grad überschneidet und ggf. ignoriert oder neu berechnet. Die Variablen nehmen die Werte 0,5 und 1 an. Die Variable `num` gibt die Anzahl aller vorhandenen Ankerrahmen an. Die Breite und Höhe der Ankerrahmen wird durch eine kommaseparierte Liste in der Variablen `anchors` angegeben. Die letzte Variable `mask` enthält eine kommaseparierte Liste mit Indizes. Mit den Indizes wird bestimmt, welche Ankerrahmen aus der `anchors` Liste in dieser Schicht verwendet werden.

Ankerrahmen wurden jeweils für die beiden Netzwerke und die Trainingsdaten neu berechnet, wie von Redmon und Farhadi [RF18, S. 1 f.] angegeben. Für die Berechnung wurde AlexeyAB's Fork¹ von Darknet verwendet, weil dort eine entsprechende Methode vorhanden ist:

```
darknet detector calc_anchors cyclist.data -num_of_clusters 9 -width  
608 -height 608
```

Die Funktion wird mit `darknet detector calc_anchors` aufgerufen. Darauf folgt der Pfad zu der Datensatzdatei. Die Variable `num_of_clusters` ist die Anzahl der Ankerrahmen, die ausgegeben wird. Die beiden Variablen `width` und `height` sollen den in der Konfigurationsdatei angegeben Werten für `width` und `height` entsprechen.

Verschiedene Trainingsvariablen werden mit den Validierungsdaten über 500 Epochen trainiert. Anschließend wird die optimalste Einstellung der Trainingsvariablen für das Training verwendet, welche nun beschrieben werden. Trainiert wird mit einer Batchgröße von 96 Bildern. Obwohl der Wert relativ groß ist, ist die Konvergenzgeschwindigkeit akzeptabel. Zusätzlich wirkt eine hohe Batchgröße dem Varianzproblem entgegen, wie Mu Li et al. schreiben [Li+14]. Mit einer Unterteilung der Batchgröße auf 16 Teile wird der Arbeitsspeicher der GPUs optimal genutzt, da bei der höchsten Auflösung von 608 x 608 Pixeln pro Bild maximal sechs Bilder in den Arbeitsspeicher der GPUs passen. Die Lernrate wird mit 0,001 initialisiert und nach einer bestimmten Anzahl von Iterationen heruntergesetzt. Bei 50 000 Iterationen wird die Lernrate das erste Mal halbiert. Bei 75 000 wird die Lernrate wieder halbiert und bei 90 000 um den Faktor zehn verkleinert. Weitere Variablen werden in Tabelle 4.5 aufgelistet.

¹Quellcode unter <https://github.com/AlexeyAB/darknet/>

Variable	Wert
momentum	0,9
decay	0,0005
angle	20
saturation	1,5
exposure	1,5
hue	0,1

Tab. 4.5.: Die eingestellten Werte für die Trainingsvariablen. Mit diesen Werten werden die beiden Netzwerke YOLOv3 und YOLOv3-tiny trainiert, um Fahrradfahrer zu detektieren.

Datensatzdatei

Darknet verwendet zum Trainieren und zum Testen eine Datensatzdatei, in welcher mehrere Informationen enthalten sind:

- Die Anzahl der Klassen.
- Der Pfad zu einer Datei, in der in jeder Zeile ein Klassenname steht. Die Anzahl der Namen sollte gleich mit der zuvor angegebenen Anzahl der Klassen sein.
- Zwei Pfade zu Dateien, die pro Zeile einen absoluten Pfad zu einem Ordner mit Bilddateien enthalten.
 - Einen Pfad für die Trainingsbilder.
 - Einen Pfad für die Testbilder.
- Der Pfad, an dem die trainierten Gewichte gespeichert werden.

Die Bilddateien liegen im JPEG oder PNG Format vor. Die Markierungsdateien müssen den gleichen Namen wie die Bilddateien haben, die Dateiendung lautet `.txt`. Falls die Bilddateien in einem Verzeichnis mit dem Namen „images“ liegen, müssen die Markierungsdateien in einem parallelen Verzeichnis namens „labels“ liegen, ansonsten in demselben Verzeichnis. Das benötigte Format der Markierungsdatei ist pro Zeile eine Detektion. Dazu gehört als erstes die Klasse als Nummer, diese entspricht der Zeile in der Datei mit den Klassennamen. Danach wird der Begrenzungsrahmen mit den Koordinaten seines Mittelpunkts, der Breite und der Höhe als Gleitkommazahl erwartet, wobei alle Zahlen relativ zur Bildgröße sind.

4.3.2 Trainingsverlauf

Die Ausgabe des Trainings mit Darknet wird in eine Datei umgeleitet. Der Aufruf zum Trainieren ohne Umleitung ist:

```
darknet detector train cyclist.data train.cfg Darknet.conv.53 -gpus 0,1
```

Mit `darknet detector train` wird die Funktion zum Trainieren aufgerufen. Anschließend folgen die Pfade für die Datensatzdatei, die Konfigurationsdatei und die Gewichtsdatei. Hier wird der Parameter `gpus` verwendet, mit dem angegeben werden kann, welche GPUs verwendet werden sollen.

Während des Trainings wird überprüft, ob der Fehler weiterhin sinkt oder steigt. Außerdem wird auf „Early Stopping“-Kriterien geachtet, wobei das Training ggf. abgebrochen wird. Um die Änderung des Fehlers zu beobachten, werden selbst geschriebene Shell, Batch und Python Skripte miteinander kombiniert. Vom Windows Rechner wird ein Batch Skript gestartet, welches sich mit dem Trainingscomputer per SSH verbindet und dort ein Shell Skript ausführt, siehe im Anhang Listing B.3. Das Shell Skript prüft, ob trainiert wird und speichert entsprechend eine 0 oder eine 1 in die Datei mit dem Namen „`laeufs`“. Außerdem werden aus der Datei, in der die Ausgabe des Trainings steht, alle Fehlerwerte ermittelt. Die Fehlerwerte werden in der Datei „`loss.txt`“ gespeichert. Damit ist das Shell Skript abgeschlossen, siehe im Anhang Listing B.4. Das Batch Skript kopiert beide Dateien in den lokalen Speicher. Wenn das Training noch läuft, dann wird der Verlauf des Fehlers des Trainings mit einem Python Skript in ein Diagramm übertragen, siehe im Anhang Listing B.5. Sollte das Training nicht mehr laufen, wird das Lied Kalimba von Mr. Scruff abgespielt. Die Abbildung 4.3 zeigt den Verlauf des durchschnittlichen Fehlers in Prozent über die Anzahl der Iterationen eines Trainings. Bei 50 000 Iterationen wird die Lernrate das erste Mal halbiert, welches sich sehr deutlich in der Fehlerkurve in Abbildung 4.3 niederschlägt.

Die Batch Größe für eine Iteration beträgt bei diesem Training 96 Bilder und die maximale Anzahl der Iterationen beträgt 100 000. Der Verlauf des Fehlers ist wie erwartet annähernd eine exponentielle Kurve. Das Minimum wird nach ungefähr 92 000 Iterationen erreicht. Das entspricht einem Training von 440 Epochen. Durch längeres Training in Kombination mit einer kleineren Lernrate sinkt der Fehler nicht weiter und die mAP ändert sich nicht. Der Fehler für die Test- und Trainingsdaten ist ungefähr gleich hoch. Das sind Indizien für ein Varianzproblem.

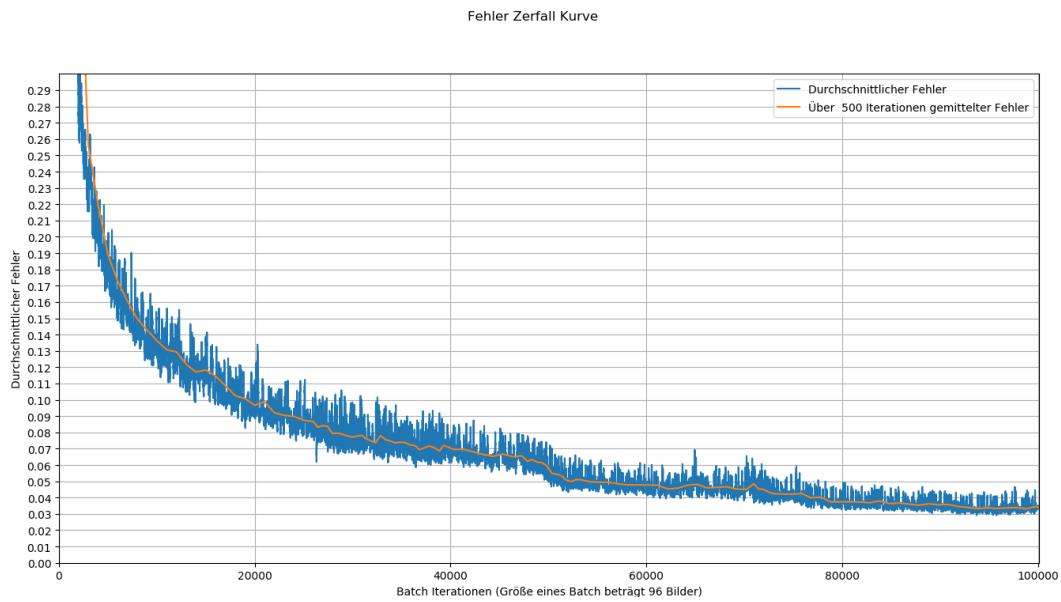


Abb. 4.3.: Durchschnittlicher Fehler über die Anzahl der Iterationen. Die blaue Kurve zeigt einen typischen Verlauf einer Fehlerkurve bei Verwendung von Mini-Batch Gradient. Die gelbe Kurve mittelt den Fehler jeweils über 500 Iterationen. Der Plot wurde mit Python anhand der Ausgabe des Trainings von Darknet erstellt.
Quelle: Eigene Darstellung

4.3.3 Trainingsende

Nachdem das Training beendet ist, werden die Gewichte, die Trainingsvariablen, die Einstellungen im KNN und der Verlauf des Fehlers gespeichert. Für alle gespeicherten Gewichte werden die Werte mAP, Precision, Recall, TP, FP und FN berechnet. Die Bedeutung dieser Werte wird später erläutert. Dazu wird die `map` Funktion des Darknet Forks von AlexeyAB² verwendet:

```
darknet detector map datensatz.data netzwerk.cfg netzwerk.weights
```

Die Funktion wird mit `darknet detector map` aufgerufen und erwartet anschließend drei Pfade für die Datensatzdatei, die Konfigurationsdatei und die Gewichte. Das Ergebnis wird direkt auf die Standardausgabe ausgegeben. Ein selbst geschriebenes Java Programm wird verwendet, um die Daten in einer Tabelle darzustellen, siehe Tabelle 4.4 und im Anhang Listing B.6.

²<https://github.com/AlexeyAB/darknet/>

	weight	detections	ap	precision	recall	F1	TP	FP	FN	IoU
3	445500	12866	72,01	92,71	72,62	81,45	9657	758	3641	76,23
4	448000	12230	72,02	93,9	71,74	81,34	9540	620	3758	77,13
7	455500	13368	71,94	91,61	72,79	81,13	9680	886	3618	75,32
25	final	11788	71,9	94,22	71,24	81,14	9474	581	3824	76,77

Abb. 4.4.: Beispieldaten der berechneten Genauigkeitswerte. Aufgelistet sind für verschiedene Gewichte (weight) die Anzahl der Detektionen (detections), FP, TP und FN, sowie die prozentuale Angabe der durchschnittlichen Präzision (ap), der Präzision (precision), der Trefferrate(recall), der Genauigkeit (F1) und der Überschneidung (IoU).

Quelle: Eigene Darstellung

4.4 Hardware

Für diese Arbeit wird auf einen Computer mit zwei NVIDIA GeForce 1080 TI Grafikkarten mit jeweils 11 GB Arbeitsspeicher verwandt. Außerdem wird ein Laptop mit einer NVIDIA GeForce 1050 TI Grafikkarte mit 4 GB Arbeitsspeicher genutzt. Auf beiden Systemen wird Ubuntu als Betriebssystem verwendet. Auf ersten Computer wird ausschließlich per SSH zugegriffen. Das Framework Darknet wird verwendet und mit Funktionen erweitert. Auf dem Laptop wird die integrierte Entwicklungs-umgebung Eclipse für C und C++ Entwicklung verwendet. Dort werden neue Funktionen entwickelt, getestet und Fehler behoben. Für die Versionsverwaltung wird Git verwendet.³

Das Trainieren der Gewichte des KNN ist sehr rechenintensiv und wird daher nicht direkt im eingebetteten System durchgeführt. Das Training erfolgt mittels Darknet auf beiden GPUs des Computers gleichzeitig, wobei sechs Bilder je GPU in dessen Arbeitsspeicher geladen werden. Nach dem Training wird der Algorithmus auf mehreren Mikrocontrollern integriert und getestet: ODROID XU4, Nvidia Jetson AGX Xavier, Nvidia Jetson TX2 und Nvidia Jetson TK1. Die Mikrocontroller haben jeweils ein auf Linux basierendes Betriebssystem installiert. OpenCV, CUDA und CUDNN sind für das jeweilige System in den aktuellsten Versionen installiert. Eine Kamera vom Modell DFM 22BUC03-ML wird jeweils per USB an den Mikrocontroller angeschlossen und die dafür benötigten Treiber werden installiert.

Um die Verwendung in einem PKW zu simulieren, wird das sogenannte VeloxCar verwendet. Das VeloxCar entstammt aus dem Projekt „Velox“ des Unternehmens Expleo Germany GmbH. Dieses Projekt ist ein Ausbildungs- und Forschungsprojekt und wird im Embedded Lab des Kompetenzcenters Safety und Systems Engineering der Firma Expleo Germany GmbH durchgeführt. Hauptziel dieses Projektes ist die

³Das Repository kann von Mitgliedern der Beuth-Hochschule unter <https://gitlab.beuth-hochschule.de/s72834/darknet> begutachtet werden und wird zusätzlich der Arbeit beigelegt.

Ausstattung von Modellfahrzeugen mit modernen Antriebs- und Fahrerassistenzfunktionen, die autonomes Fahren ermöglichen sollen. Die Verwendung des FFDS auf dem VeloxCar ist vergleichbar mit den anderen Mikrocontrollern, nur dass zusätzlich ein Feldbus zur Datenübermittlung zur Verfügung steht. Die Veröffentlichung der Ausgabedaten des FFDS erfolgt im Rahmen dieser Arbeit auf dem VeloxCar über ein ROS Node, welches mit dem CAN-Bus in einem Auto vergleichbar ist. Ein Empfänger wird installiert, der die Ausgabedaten des FFDS empfängt. Der Empfänger verarbeitet die Daten nicht weiter, da dies nicht im Anwendungsbereich dieser Arbeit liegt. Im folgenden Kapitel wird die Struktur des fertigen Systems und seine konkrete Funktionsweise mithilfe von UML Diagrammen erläutert.

Systementwurf

Dieses Kapitel befasst sich mit dem Systementwurf des im VeloxCar integrierten Fahrradfahrer Detektionssystem (FFDS). Im ersten Schritt wird eine Übersicht der Systemarchitektur gegeben, um die verwendeten Hard- und Softwarekomponenten aufzuzeigen. Anschließend wird detailliert auf die Schnittstellen und Beziehungen der Softwarekomponenten eingegangen. Danach wird die dynamische Struktur vom Einschalten der Hardware bis hin zum Abschalten der Hardware im Detail erläutert.

5.1 Systemarchitektur

Zunächst wird eine Übersicht der Systemarchitektur gegeben und die Hard- und Softwarekomponenten identifiziert. Anschließend werden anhand eines Komponentendiagramms die Bestandteile der Komponenten und die verwendeten Schnittstellen aufgezeigt. Mit Hilfe eines Klassendiagramms werden die Attribute und Methoden, sowie die Abhängigkeiten der wichtigsten Klassen in den Softwarekomponenten abgebildet.

5.1.1 Gesamtsystem

Das im VeloxCar eingebettete FFDS ist in Abbildung 5.1 mit Hilfe eines Verteilungsdiagramm für eine erste Übersicht dargestellt. Auf dem VeloxCar besteht die verwendete Hardware aus einer USB-Kamera vom Modell DFM 22BUC03-ML und einem Mikrocontroller mit GPU vom Modell Nvidia Jetson TX2. Im Zentrum des Systems steht der Mikrocontroller, welcher die drei Aufgaben der Bilderbeschaffung, der Detektion von Fahrradfahrern und das Veröffentlichen der Detektionsergebnisse erledigt. Jede Aufgabe wird von einer Softwarekomponenten übernommen.

Auf dem Mikrocontroller befinden sich fünf Softwarekomponenten. Zu den Softwarekomponenten gehört erstens die Kameraanbindung. Die Kameraanbindung ist dafür zuständig, eine Verbindung zur Kamera herzustellen und die Bilddaten in ein bestimmtes Format zu überführen. Die Kameraanbindung muss auf anderen Systemen, je nach verwendeter Schnittstelle mit der Kamera, ggf. ergänzt oder ausgetauscht

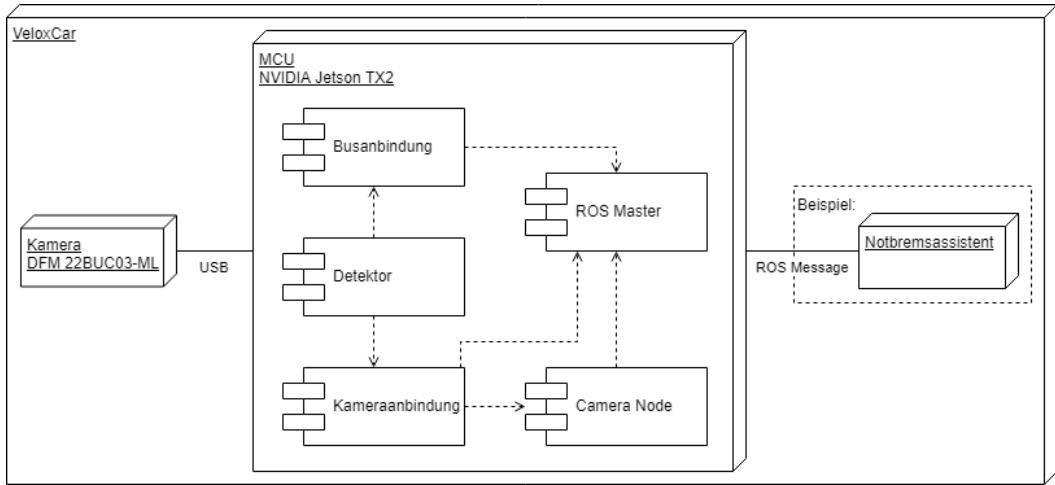


Abb. 5.1.: VeloxCar mit eingebettetem FFDS. Die verwendete USB-Kamera und der verwendete Mikrocontroller sind zu sehen. Im Mikrocontroller sind alle Softwarekomponenten enthalten. Beispielhaft ist ein Notbremsassistent dargestellt, der über ROS Messages mit dem Mikrocontroller kommunizieren kann.

Quelle: Eigene Darstellung

werden. Das Kernstück der Software ist der Detektor, der die Bilddaten verarbeitet und anschließend das Ergebnis an die Busanbindung übergibt. Die Busanbindung ist die dritte Komponente der Software, die über einen oder mehrere Feldbusse das Ergebnis der Detektionen veröffentlicht. Die Busanbindung muss auf anderen Systemen, je nach vorhandenen Feldbussen, ggf. ergänzt oder ersetzt werden. Die vierte Softwarekomponente ist der Camera Node, der über den Feldbus im VeloxCar Bilddaten der Kamera veröffentlicht. Die letzte Komponente ist der ROS Master, welcher den Feldbus im VeloxCar koordiniert. ROS wird im folgenden Unterkapitel genauer erläutert. Die Softwarekomponenten Camera Node, Kameraanbindung und Busanbindung sind ROS Nodes die mit ROS Messages kommunizieren. Im abgebildeten Verteilungsdiagramm ist beispielhaft ein Notbremsassistent aufgeführt. Auf dem VeloxCar ist das Assistenzsystem nur ein weiterer ROS Node auf dem Mikrocontroller, welcher die Ergebnisse empfängt und nicht weiterverarbeitet.

Robot Operating System

Bei dem Robot Operating System (ROS) handelt es sich nicht um ein unabhängiges Betriebssystem, sondern um eine Betriebssystemerweiterung. ROS dient in erster Linie zum Erstellen von Roboter Applikationen. Dazu bietet ROS u.a. neben Bibliotheken, Hardware Abstraktion und Gerätetreibern auch Werkzeuge zur Visualisierung, Nachrichtenübermittlung und Paketverwaltung an. ROS ist quelloffen und enthält eine BSD-Lizenz.

Ein ROS Knoten (ROS Node) ist ein unabhängiger Prozess, der Berechnungen durchführt und eine ROS client library verwendet. Knoten kommunizieren über Nachrichten (ROS message). Eine Nachricht ist eine Datenstruktur mit typisierten Feldern; Standard Datentypen (Integer, Float, Boolean) und Arrays derselben werden unterstützt. Nachrichten können beliebig verschachtelte Strukturen enthalten. Das Standardkommunikationsprotokoll ist TCPROS. Nachrichten werden mit einem Beobachter Entwurfsmuster (Publisher-Subscriber) verteilt. Ein Knoten veröffentlicht Nachrichten zu einem Thema (publish to a topic), diese werden über einen eindeutigen Namen identifiziert. Ein anderer Knoten, der an diesen Nachrichten Interesse hat, abonniert das Thema (subscribe). Die Namen der Themen werden über den ROS Master registriert bzw. nachgeschlagen. Beim Abonnieren eines Themas erhält ein ROS Node Verbindungsinformationen, damit kommunizieren Knoten direkt miteinander, vgl. Online Dokumentation [Tul19]. Die Schnittstelle, um ein Thema anzumelden oder zu abonnieren, nennt sich in der C++ Bibliothek `NodeHandle`.

5.1.2 Softwarekomponenten

Die Softwarekomponenten mit verwendeten Schnittstellen, Dateien und Bibliotheken sind in der Abbildung 5.2 als Komponentendiagramm abgebildet.

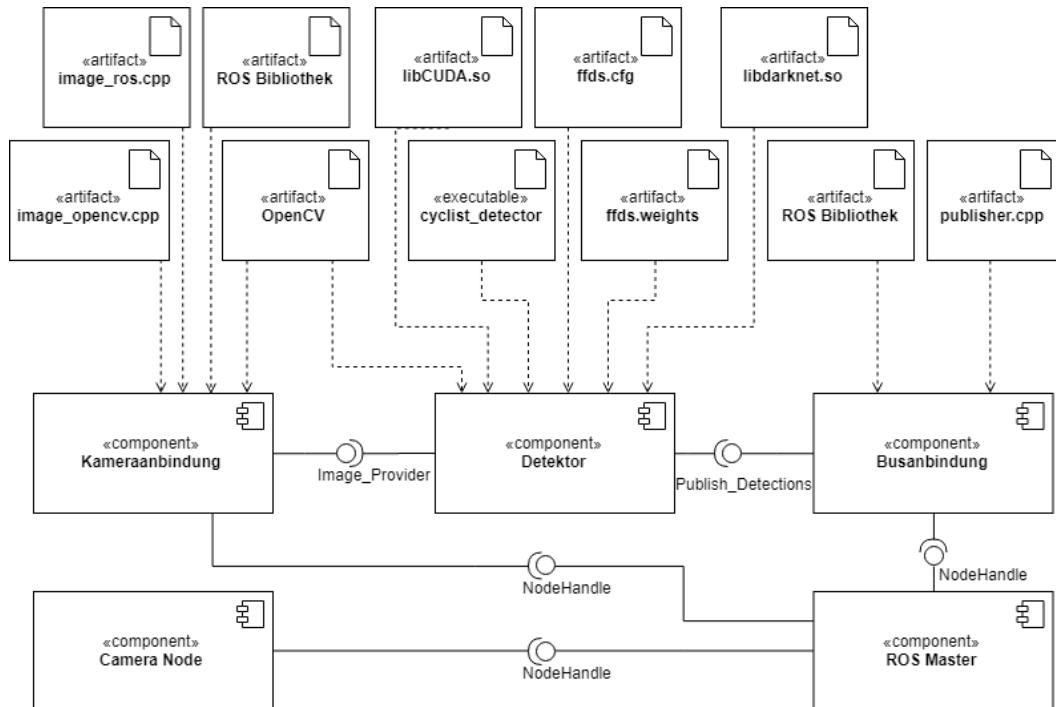


Abb. 5.2.: Softwarekomponenten des FFDS mit verwendeten Schnittstellen, Bibliotheken und Dateien.

Quelle: Eigene Darstellung

Die erste Komponente - die Kameraanbindung - stellt zum einen die Verbindung zur Kamera her, zum anderen liefert sie Bilddaten an den Detektor. Die Kameraanbindung bietet die Schnittstelle `Image_Provider` an. Die vom `Image_Provider` gelieferten Bilddaten werden in einem bestimmten Format erwartet. Das Format beinhaltet Informationen zur Breite, Höhe und Anzahl der Farbkanäle des Bildes sowie die Bilddaten selbst. Die Kameraanbindung verwendet auf dem VeloxCar einen ROS `NodeHandle`, da auf dem VeloxCar die Kamerabilder über ROS Messages veröffentlicht werden. Die Komponente Camera Node veröffentlicht Bilddaten und Informationen über die Bilder über zwei separate Themen. Die Kameraanbindung verwendet den `NodeHandle`, um die Verbindungsinformationen vom ROS Master zu erhalten und dadurch direkt mit dem Camera Node zu kommunizieren. Die Kameraanbindung, die ein `NodeHandle` zum Erhalten der Bilddaten verwendet, ist in der Datei `image_ros.cpp` enthalten. Die C++ Bibliothek für ROS wird verwendet. Für die Tests auf den anderen Mikrocontrollern und Rechnern verwendet die Kameraanbindung die OpenCV Bibliothek. Die Komponente Kameraanbindung ist in diesem Fall in der Datei `image_opencv.cpp` enthalten.

Der Detektor - die zweite Komponente des Systems - verwendet die Schnittstelle `Image_Provider` um erstens die Verbindung zur Kamera zu initialisieren und zweitens Bilder abzurufen. Der Detektor verwendet das im Rahmen dieser Arbeit modifizierte Darknet als Bibliothek (`libdarknet.so`). Darknet benötigt die Konfigurationsdatei für die Architektur des KNN, diese befindet sich in der Datei `ffds.cfg`. Zusätzlich werden die trainierten Gewichte aus der Datei `ffds.weights` benötigt. Die ausführbare Datei zum Starten des FFDS ist `cyclist_detector`. Der Detektor verwendet die Schnittstelle `Publish_Detections`, um erstens die Verbindung zum Feldbus zu öffnen und zweitens die Ergebnisse der Fahrradfahrer Detektion auf einem Bild zu veröffentlichen.

Die dritte Komponente, die Busanbindung, bietet die Schnittstelle `Publish_Detections` an. Die Busanbindung verwendet zur Veröffentlichung der Ergebnisse einen auf der Plattform verfügbaren Feldbus. Die erhaltenen Ergebnisse werden in das entsprechende Format übertragen und veröffentlicht. Der auf dem VeloxCar vorhandene Feldbus verwendet die Übertragung von ROS Messages zwischen ROS Nodes. Deswegen verwendet die Busanbindung auf dem VeloxCar ein `NodeHandle` und meldet das Thema zum Veröffentlichen der Detektionen beim ROS Master an. Die Ergebnisse werden jeweils in das Format einer ROS Message überführt und unter dem angemeldeten Thema veröffentlicht. Die Busanbindung verwendet dazu die C++ Bibliothek für ROS und ist in der Datei `publisher.cpp` enthalten. Die Komponenten ROS Master und Camera Node sind nicht Teil des FFDS und sind auch nicht im Rahmen dieser Arbeit implementiert worden. Für andere Plattformen gilt, dass je nach vorhandenem Feldbus die Busanbindung entsprechend implementiert werden muss.

Die wichtigsten Klassen des FFDS sind mit ihren Beziehungen und Methoden zur besseren Übersicht als Klassendiagramm in Abbildung 5.3 dargestellt.

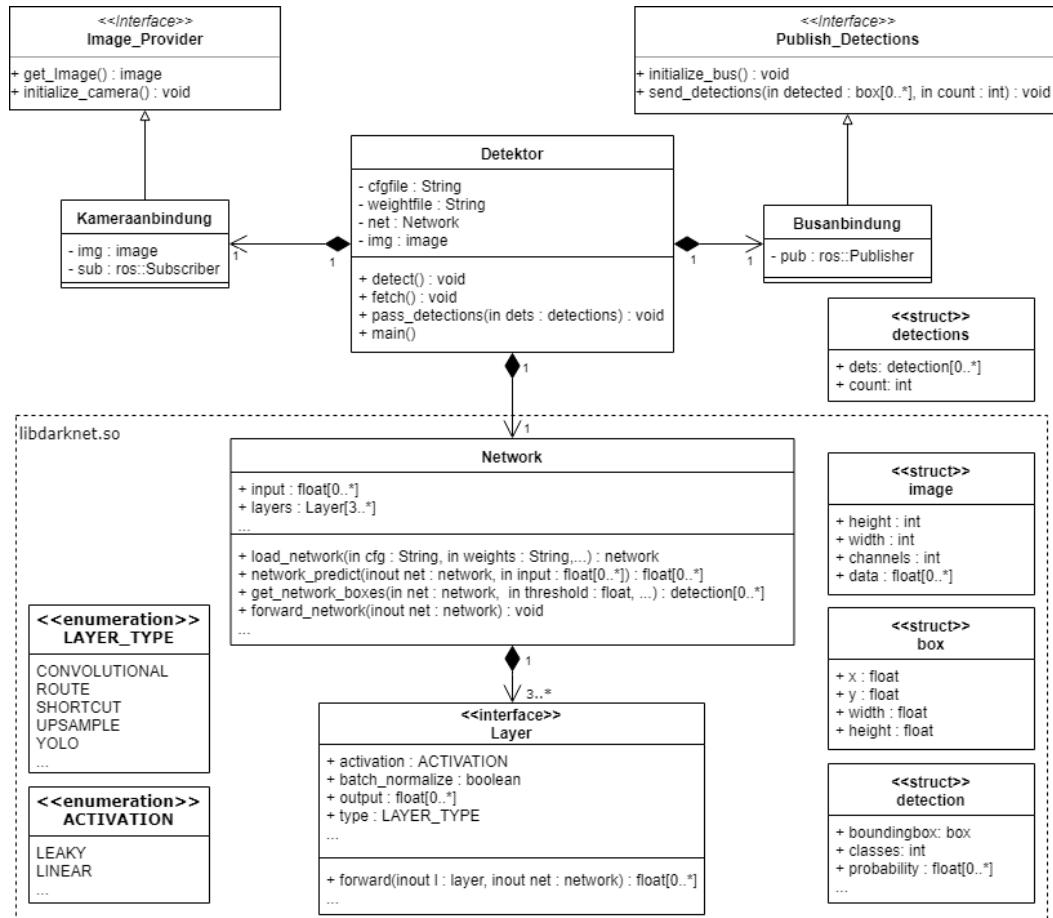


Abb. 5.3.: Die wichtigsten Klassen und Methoden des FFDS. Die im unteren Bereich eingeklammerten Klassen, Strukturen und Enumerationen befinden sich in der Bibliothek libarknet.so.

Quelle: Eigene Darstellung

Auf ein vollständiges Klassendiagramm wird aus Gründen der Komplexität in der Darstellung verzichtet. Zur Verdeutlichung: die Klasse `Network` besitzt beispielsweise 55 Attribute, die meisten davon sind Trainingsvariablen, welche nur während des Trainings verwendet werden. Einige Attribute sind für die Vermehrung der Daten wichtig. Das wichtigste Attribut für die Berechnung eines Bildes ist das im Diagramm aufgeführte Attribut `layer`, welches ein Array aller Schichten des Netzwerks enthält. Jede Schicht ist dabei eine Spezialisierung der Schnittstelle `Layer`. Die Schnittstelle `Layer` ist als C Struktur mit über 250 Attributen implementiert.

Die in der Abbildung 5.3 nicht aufgeführten Klassen und Methoden von Darknet lassen sich einteilen in:

- Allgemeine Werkzeuge, die z.B die aktuelle Zeit wiedergeben, die Summe berechnen, oder die Konfigurationsdatei einlesen.
- Werkzeuge zur Manipulation von Bildern.
- Die einzelnen von Darknet unterstützten Schichten.
- Die Algorithmen für das maschinelle Lernen mit und ohne GPU Unterstützung, u.a. die Berechnung der Kostenfunktion, die Fehlerrückführung und die Anpassung der Gewichte.
- Verschiedene Anwendungen um z.B. einen Klassifizierer oder Segmentierer zu trainieren, validieren oder verwenden.

Im Fokus der Abbildung 5.3 stehen die Klassen Kameraanbindung, Detektor und Busanbindung, sowie die Schnittstellen `Image_Provider` und `Publish_Detections`. Im Klassendiagramm lässt sich erkennen, dass die Klasse Kameraanbindung die Schnittstelle `Image_Provider` implementiert. Die Schnittstelle `Publish_Detections` wird von der Klasse Busanbindung implementiert. Diese beiden Klassen müssen ggf. auf einer anderen Plattform angepasst werden. Das Austauschen dieser Klassen erfolgt über den `include` Befehl in der Klasse Detektor. Die Schnittstelle `Image_Provider` besitzt zwei Methoden. Die erste Methode `initialize_camera` etabliert die Verbindung zu einer Kamera abhängig davon wie die Kamera ansprechbar ist. Die zweite Methode `get_image` liefert ein aktuelles Bild in einer `image` Struktur. Die erste Methode muss vor der zweiten aufgerufen werden. Die Schnittstelle `Publish_Detections` muss, je nachdem welcher Feldbus vorhanden ist bzw. wie die Kommunikation mit anderen Systemen verläuft, entsprechend implementiert werden. Diese Schnittstelle besitzt ebenfalls zwei Methoden. Die erste Methode `initialize_bus` führt die notwendigen Schritte durch, um Ergebnisse über den Feldbus veröffentlichen zu können. Nachdem die Verbindung zum Feldbus hergestellt ist, soll zu jederzeit eine Veröffentlichung von Daten stattfinden können. Das Veröffentlichen der Daten wird von der zweiten Methode `send_detections` umgesetzt. Dazu werden die Ergebnisse im richtigen Format mit dem richtigen Protokoll über die richtigen Schnittstellen versendet. Die Methode `send_detections` erhält als Parameter eine Liste aller Begrenzungsrahmen, in denen sich auf dem analysierten Bild Fahrradfahrer befinden. Ein weiterer Parameter gibt die Anzahl der Detektionen an.

Im folgenden Kapitel wird der genaue Ablauf des FFDS erklärt und insbesondere der Algorithmus für die Detektionen beschrieben.

5.2 Dynamisches Verhalten

Für die Beschreibung des dynamischen Verhaltens vom FFDS wird zunächst ein UML Aktivitätsdiagramme angeführt, welches den Programmablauf des Detektors vom Start über die Verarbeitung von Bildern bis hin zur Veröffentlichung der Ergebnisse abbildet. Anschließend wird in einem Sequenzdiagramm der gesamte Ablauf vom Start des Systems bis hin zur Veröffentlichung der Ergebnisse modelliert.

Das VeloxCar bietet über Akkumulatoren Strom an. Die Stromzufuhr kann an- und ausgeschaltet werden. Alternativ lässt sich das VeloxCar auch mit einer externen 12V Stromquelle betreiben. Sobald die Stromzufuhr angeschaltet ist, wird der Mikrocontroller mit Strom versorgt und startet automatisch das Betriebssystem. Nachdem das Betriebssystem geladen ist, startet ein Skript. Das Skript startet wiederum u.a. den ROS Master, den Camera Node und das FFDS. Das Start-Skript startet auch das Programm, welches dazu dient, die veröffentlichten Detektionen zu empfangen.

Der Programmablauf wird in Abbildung 5.4 dargestellt. Der Ablauf beginnt sobald das Start-Skript das FFDS startet. Zuerst wird die Architektur des KNN aus der Konfigurationsdatei `ffds.cfg` ausgelesen. Damit einhergehend wird im Arbeitsspeicher der GPU Speicherplatz reserviert. Anschließend werden die Gewichte aus der Datei `ffds.weights` eingelesen und in den Arbeitsspeicher der GPU geladen. Ab diesem Zeitpunkt ist der Objekt Detektor fertig initialisiert und bereit, Bilddaten zu verarbeiten. Als nächstes wird die Verbindung zur Kamera initialisiert. Auf dem VeloxCar veröffentlicht ein ROS Node (Camera Node) Bilddaten der Kamera über das Thema „`camera/image_color`“. Zusätzlich veröffentlicht der Camera Node über das Thema „`camera/camera_info`“ Meta-Informationen über Breite, Höhe und Anzahl der Farbkanäle der Bilder. Bei der Initialisierung der Verbindung zur Kamera wird ein ROS Node für die Kommunikation gestartet. Die Kameraanbindung meldet sich für das Thema mit den Meta-Informationen an. Nach Erhalt der Informationen über die Bilder meldet sich die Kameraanbindung wieder von dem Thema „`camera/camera_info`“ ab. Danach abonniert die Kameraanbindung zum Erhalten der Bilddaten das Thema „`camera/image_color`“. Durch die Anmeldung wird die direkte Verbindung zum Camera Node hergestellt. Die Kameraanbindung erhält in einer unendlichen Schleife ROS Messages vom Camera Node mit den aktuellen Bilddaten. Ab diesem Zeitpunkt können jederzeit Bilddaten abgerufen werden.

Der letzte Schritt des Startvorgangs ist die Initialisierung der Kommunikation über den Feldbus. Bei der Initialisierung der Busanbindung wird ein ROS Node gestartet, falls die Kameraanbindung dies zuvor nicht getan hat. Mit einem `NodeHandle` wird beim ROS Master das Thema „`CyclistDetectOnCAM1`“ angemeldet. Über das angemeldete Thema werden die Begrenzungsrahmen detekterter Fahrradfahrer auf Bildern vom Camera Node veröffentlicht.

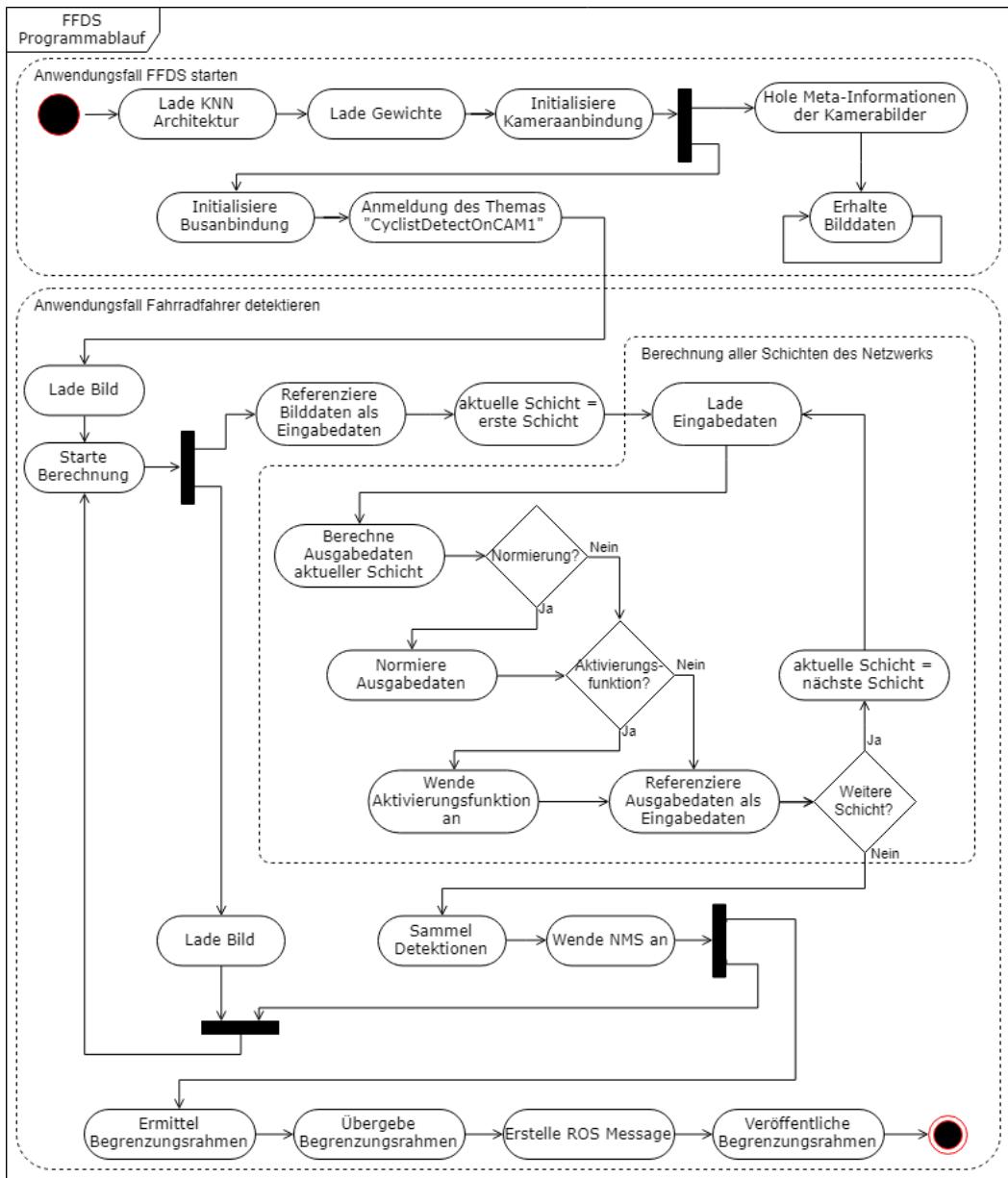


Abb. 5.4.: Vollständiger Programmablauf des FFDS. Die beiden Anwendungsfälle „FFDS starten“ und „Fahrradfahrer detektieren“ sind mit einer gestrichelten Linie umrandet. Zusätzlich ist innerhalb des Anwendungsfalls „Fahrradfahrer detektieren“ die Berechnung durch das Netzwerk umrandet, dieser Teil wird von der GPU berechnet.

Quelle: Eigene Darstellung

Ab diesem Zeitpunkt ist das System bereit, Bilder der Kamera zu erhalten, auf den Bildern Fahrradfahrer zu detektieren und die ermittelten Detektionen über den Feldbus zu veröffentlichen.

Der Startvorgang endet mit dem Laden des ersten Bildes. Sobald ein Bild angefordert wird, werden die Bilddaten und die Meta-Informationen dem Detektor zur Verfügung gestellt. Nachdem das erste Bild geladen ist, beginnt eine unendliche Schleife.

Jedes Bild wird wie folgt verarbeitet:

Zunächst werden die Bilddaten in die Eingabeschicht des Netzwerks geladen. Dazu werden die Bilddaten als Eingabedaten im Netzwerk referenziert. Nun beginnt die Berechnung der Bilddaten durch das KNN, dies geschieht in einer Schleife solange, bis alle Schichten durchlaufen sind. Das Netzwerk lädt die Eingabedaten für die Berechnung in den Arbeitsspeicher der GPU. Dann wird die Berechnung der aktuellen Schicht gestartet. Nach der Berechnung wird ggf. die Normierung der Daten durchgeführt. Abschließend wird ggf. die Aktivierungsfunktion berechnet. Die Ausgabedaten der jeweils aktuellen Schicht werden als Eingabedaten im Netzwerk referenziert. Hier endet die Schleife, sobald alle Schichten durchlaufen sind.

Nachdem alle Schichten berechnet sind, befinden sich die Ergebnisse der Detektionen in den yolo-Schichten, siehe Kapitel 4.1. Alle Detektionen werden gesammelt und anschließend wird die Non Maximum Supression (NMS) angewendet. Die NMS dient zum Unterdrücken von mehrfach Detektionen eines Objekts. Ein separater Thread wird gestartet, um die Begrenzungsrahmen detekterter Fahrradfahrer über den Feldbus zu veröffentlichen. Aus den Detektionen werden die relevanten Begrenzungsrahmen ermittelt. Die Begrenzungsrahmen, welche mit einer Wahrscheinlichkeit von über 25% korrekt sind, werden als relevant eingestuft. Die Begrenzungsrahmen werden als Liste, zusammen mit deren Anzahl der Busanbindung übergeben. Die Busanbindung erstellt aus den Begrenzungsrahmen eine kommaseparierte Liste als String. Jeder Listeneintrag besteht dabei aus vier Gleitkommazahlen. Diese repräsentieren die X-, Y-Koordinaten, Breite und Höhe eines Begrenzungsrahmen und sind relativ zur Bildgröße zu verstehen. Der String mit der Liste wird dann als ROS Message über das zuvor angemeldete Thema „CyclistDetectOnCAM1“ veröffentlicht. Parallel zur Berechnung eines Bildes werden über die Kameraanbindung die nächsten Bilddaten angefordert. Nach der Berechnung eines Bildes folgt der nächste Schleifendurchlauf und das nächste Bild wird berechnet.

Das Programm läuft in einer Endlosschleife. Wenn der Strom zum Mikrocontroller abgestellt wird, stoppt das Programm. Beim erneuten Anschalten beginnt der Programmablauf wieder mit dem Start des FFDS, wodurch immer ein definierter Zustand gegeben ist.

Der Programmablauf vom Start des FFDS über die Detektionsroutine, bis hin zum Veröffentlichen der Detektionen wird in Abbildung 5.5 als Sequenzdiagramm dargestellt. Dabei sind zur besseren Übersicht nur die Klassen und Strukturen aufgeführt, die auch im Klassendiagramm (Abbildung 5.3) abgebildet sind. Einzelne Methoden und Klassen werden aufgrund der Komplexität nur schriftlich aufgeführt. Zum Programmstart erhält die Methode `load_network` die Konfigurationsdatei mit der Architektur von YOLOv3 und die trainierten Gewichte. Nicht abgebildet sind das Auslesen der Datei, das Erstellen der einzelnen Schichten, das Allozieren von Arbeitsspeicher der GPU für jede Schicht, das Einlesen der Gewichte und das Laden der Gewichte in den Arbeitsspeicher der GPU. Die Methode `load_network` gibt schließlich eine Instanz der Klasse `Network` zurück.

Anschließend wird die Kameraanbindung mit dem Aufruf der Methode `initialize_camera` initialisiert. Die Kameraanbindung startet als erstes einen separaten Thread und kehrt zum Hauptprogramm zurück. Die Kameraanbindung verwendet ein `NodeHandle`, um beim ROS Master Themen zu abonnieren. Zuerst meldet sich die Kameraanbindung für das Thema der Meta-Informationen der Kamerabilder mit der Methode `subscribe` an. Die Kommunikation zwischen `NodeHandle` und ROS Master, sowie der Verbindungsauflauf zwischen Kameraanbindung und Camera Node sind nicht abgebildet. Die Methode `subscribe` liefert einen `Subscriber` zurück. Die Methode `unsubscribe` wartet auf das einmalige Empfangen der Meta-Informationen der Bilder und meldet sich dann mit Hilfe des `Subscriber` ab. Anschließend abonniert die Kameraanbindung das Thema der Bilddaten. In der Methode `listen` werden die Bilddaten empfangen. Bei jedem empfangenem Bild wird eine abrufbare `image` Struktur mit den Bilddaten und den Meta-Informationen befüllt. Dieser Thread bleibt in einer Endlosschleife und hält stets die aktuellsten Bilddaten in einer `image` Struktur bereit.

Die Busanbindung wird mit der Methode `initialize_bus` gestartet. Das Thema für die Veröffentlichung der Daten wird mit einem `NodeHandle` angemeldet. Das Anmelden des Themas „`CyclistDetectOnCAM1`“ mit der Methode `advertise` liefert eine Instanz der Klasse `Publisher` zurück. Mit dieser Klasse wird letztendlich die ROS Message veröffentlicht.

Das KNN, die Bus- und die Kameraanbindung sind zu diesem Zeitpunkt bereit und das erste Bild wird geladen. Bei jedem Aufruf von `get_image` wird das Bild mit schwarzen Pixeln aufgefüllt, damit es quadratisch ist.

Die unendliche Detektionsroutine wird gestartet. Für jedes Bild werden zwei Threads mit der Methode `pthread_create` gestartet. Der `fetch_thread` lädt das nächste Bild mit der Methode `get_image`. Der Thread `main` wartet, bis die beiden erzeugten Threads beendet sind, um die Schleife von vorne zu durchlaufen. Im `detect_thread` wird die Methode `network_predict` aufgerufen, dabei werden das Netzwerk und die Bilddaten als Parameter übergeben. Das Netzwerk wird mit den Bilddaten befüllt. Anschließend beginnt die Berechnung mit Hilfe des KNN durch den Aufruf der Methode `forward_network`.

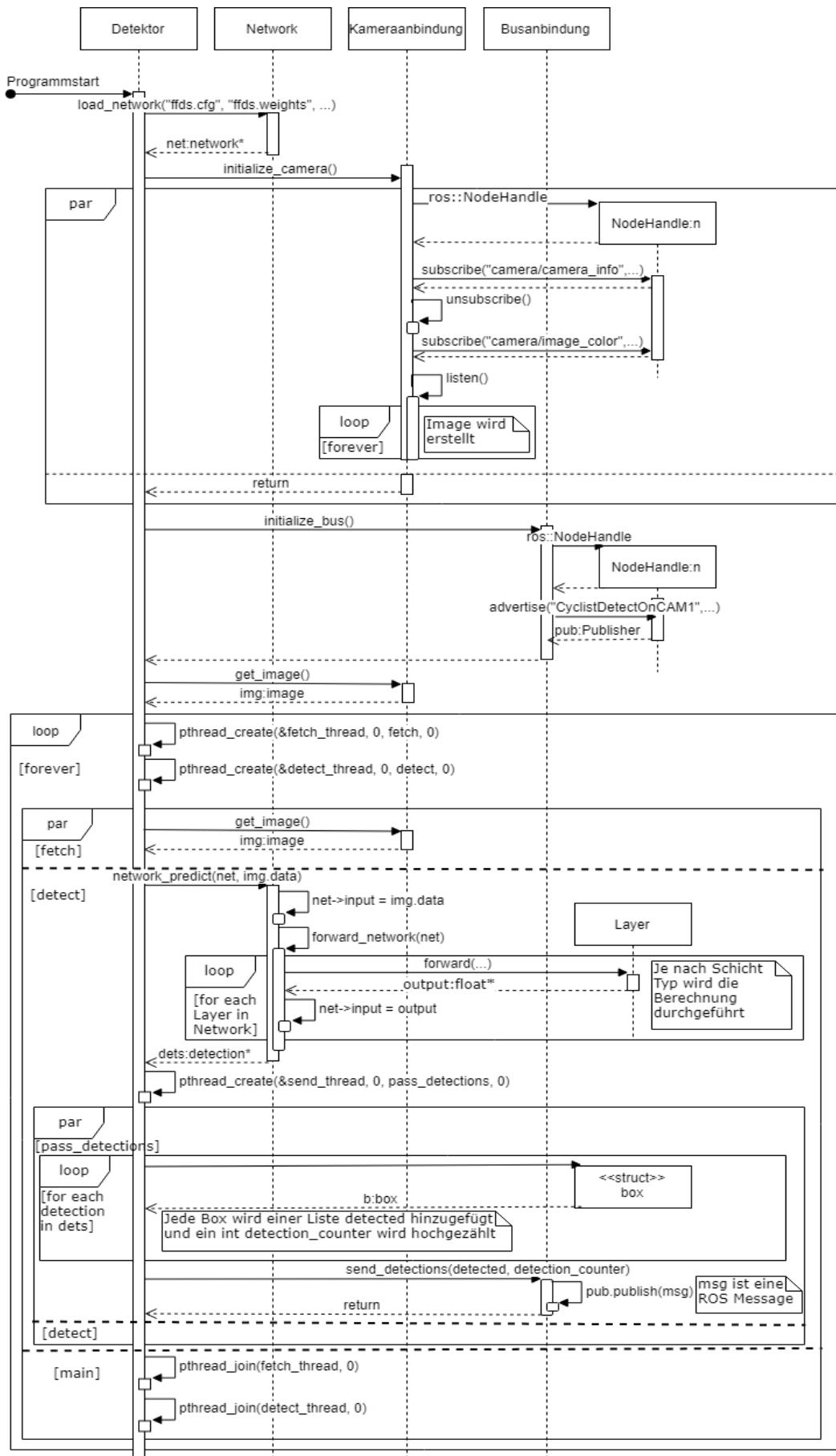


Abb. 5.5.: Die vollständige Sequenz vom Laden des Netzwerks bis zur Veröffentlichung, mit Klassen und Methodenaufrufen.
Quelle: Eigene Darstellung

Die Bilddaten werden in den Arbeitsspeicher der GPU geladen. Das Laden geschieht mit Hilfe der CUDA Runtime API. Für die Berechnung einer Schicht werden in CUDA geschriebene Funktionen verwendet, die die GPU zum Rechnen verwenden. Nachdem alle Schichten berechnet wurden, muss die Ausgabe des Netzwerkes aus dem Arbeitsspeicher der GPU in den allgemeinen Arbeitsspeicher geladen werden. Aus der Ausgabe des Netzwerks werden die Detektionen aus den yolo-Schichten ermittelt. Die Detektionen werden dann als Liste von `detection` zurückgegeben. In einem separaten Thread `send_thread` werden aus den Detektionen die Begrenzungsrahmen ermittelt. Jeder Begrenzungsrahmen wird in einer Liste aus `box` Strukturen gespeichert. Schließlich wird die Liste der Busanbindung mit der Methode `send_detections` übergeben.

Die Methode `send_detections` erzeugt aus der Liste der `box` Strukturen eine kommasseparierte Liste als String. Mit der erzeugten Liste wird eine ROS Message gefüllt. Schließlich werden die Ergebnisdaten mit dem Publisher veröffentlicht.

Das dynamische Verhalten lässt sich hauptsächlich mit zwei UML Diagrammen beschreiben. Der vollständige Ablauf mit allen Methodenaufrufen findet sich im Quellcode, der dieser Arbeit in digitaler Form mitgegeben wird. Alternativ sind im Anhang der Quellcode der Kameraanbindung, der Busanbindung und vom Detektor in den Listings B.7, B.8 und B.9 zu finden.

Evaluierung

Für die Evaluierung des Fahrradfahrer Detektionssystems (FFDS), welches im Rahmen dieser Arbeit entstanden ist, werden die erhobenen Anforderungen aus Kapitel 3.4.2 als Wertekriterien verwendet. Die Anforderungen wurden so definiert, dass sie entweder zutreffen oder nicht zutreffen. Dementsprechend ist der Leistungsstandard für ein Wertekriterium erreicht, sobald die entsprechende Anforderung zutrifft. Der Zweck der Evaluierung liegt in erster Linie in der Überprüfung der Zielerreichung. Weitere Nutzen der Evaluierung liegen in der Verwendung der Ergebnisse bei Benutzung des FFDS, dem Vermerk von Beobachtungen, der Auflistung von Limitationen und Schwachstellen und daraus abgeleitete Implikationen für Folgearbeiten. Dabei sollen Grenzen und Schwachstellen herausgearbeitet werden. Die Evaluierung erfolgt anhand einer Analyse von Soll- und Ist-Werten.

Alle funktionalen Anforderungen an das FFDS sind erfüllt, darunter zählen zum Beispiel das Einholen von Bilddaten, die Verarbeitung von Bilddaten und das Versenden der Ergebnisse. Aufgrund dessen werden in diesem Kapitel nur die nicht funktionalen Anforderungen bewertet. Zuerst wird die Geschwindigkeit und anschließend die Genauigkeit des FFDS evaluiert. Die Geschwindigkeit wird auf verschiedenen Mikrocontrollern getestet. Die Genauigkeit wird mit Hilfe der Testdaten berechnet. Anschließend erfolgt die Fehleranalyse, dazu werden stichprobenartig falsch berechnete Bilder analysiert. Abschließend werden die Tests und die Fehleranalyse ausgewertet und Limitationen des FFDS festgehalten.

6.1 Geschwindigkeitstest

Die Geschwindigkeit des FFDS wird anhand der durchschnittlich berechneten Bilder pro Sekunde (engl.: Frames per Second (FPS)) gemessen. Der Durchschnitt der berechneten Bilder pro Sekunde wird als Bildfrequenz bezeichnet. Die Geschwindigkeit wird zum Vergleich auf verschiedenen Mikrocontrollern getestet: ODROID XU4, Nvidia Jetson TK1, Nvidia Jetson TX2 und Nvidia Jetson AGX Xavier. Zusätzlich wird eine Testreihe auf dem Computer durchgeführt, um einen Vergleichswert zu erhalten.

Der ODROID wird als Referenz für Mikrocontroller ohne GPU Unterstützung verwendet, er erschien im Februar 2016. Der Jetson TK1 von NVIDIA ist ein älteres Modell, dieses wurde bereits im März 2014 herausgegeben. Der TK1 unterstützt cuDNN nicht. Drei Jahre später, im März 2017, erschien der NVIDIA Jetson TX2. Der Jetson AGX Xavier von NVIDIA ist eine weitere Lösung, um einen Graphikprozessor für künstliche neuronale Netzwerke in einem eingebetteten System zu verwenden. Der AGX ist seit Dezember 2018 erhältlich und ist somit das zum Zeitpunkt dieser Arbeit aktuellste, zu erwerbende Modell. Eine detaillierte Übersicht der verwendeten Rechner mit Informationen zu ihren Systemen ist im Anhang in der Tabelle A.1 aufgelistet.

Auf den genannten Systemen werden die beiden trainierten Netzwerke YOLOv3 und YOLOv3-tiny mit dem unmodifizierten Darknet Framework¹ getestet. Die Tests werden stichprobenartig für unterschiedliche Auflösungen durchgeführt, die folgenden Werte sind in Pixel und gelten für die Breite und die Höhe: 224, 320, 416, 512, 608, 704, 768, 800, 960, 1120, 1280. Die Wahl des Spektrums der Auflösungen beruht auf zwei Aspekten. Der erste Aspekt ist, dass während des Trainings die Bilder auf minimal 224 x 224 Pixel skaliert werden. Der zweite Punkt ist die physische Grenze des Arbeitsspeichers des TX2. Der TX2 kann mit YOLOv3 aufgrund des Mangels an Arbeitsspeicher keine Bilder der Auflösung 1280 x 1280 Pixel berechnen.

Für die beiden Netzwerke YOLOv3 und YOLOv3-tiny wird je eine Tabelle mit Vergleichswerten erstellt. Die Tabelle 6.1 enthält die Werte für YOLOv3, die Tabelle 6.2 zeigt die Werte für YOLOv3-tiny. In beiden Tabellen sind nur Auszüge aus der Auswertung enthalten, die vollständigen Tabellen mit allen Messwerten befinden sich im Anhang in der Tabelle A.2 für YOLOv3 und in der Tabelle A.3 für YOLOv3-tiny. Dort sind u.a. die Tests für die unterschiedlichen Bilddatenquelle Kamera und Videodatei aufgeführt, wobei sich die Datenquelle nur bei der Berechnung mit YOLOv3-tiny niederschlägt. Das Netzwerk rechnet bei Bilddaten aus einer Kamera so schnell, dass auf das nächste Bild der Kamera gewartet werden muss. Aus diesem Grunde wird sich in den nachfolgenden Tabellen 6.1 und 6.2 ausschließlich auf Videodatei als Bilddatenquelle fokussiert. Die Auswertung des ODROID ist in beiden Tabellen außerdem nicht aufgeführt, weil das System zu langsam ist: YOLOv3-tiny benötigt u.a. auf dem ODROID mit der kleinsten Auflösung von 224 x 224 Pixel über anderthalb Sekunden, um ein Bild zu berechnen.

In den Tabellen 6.1 und 6.2 sind die Systeme über den Tests aufgetragen. Die Werte in den Zellen sind die ermittelten Bildfrequenzen in Hz bzw. Bildern pro Sekunde. Die hervorgehobenen Werte genügen den Anforderungen, d.h. einer Bildfrequenz von 10 FPS und bieten gleichzeitig die bestmögliche Auflösung für das System.

¹<https://github.com/pjreddie/darknet/tree/61c9d02ec461e30d55762ec7669d6a1d3c356fb2>

System	Auflösung											
	224	320	416	512	608	704	768	800	960	1120	1280	
TK1	3,61	2,03	1,35	NA	NA	NA	NA	NA	NA	NA	NA	NA
TX2	10,25	5,61	3,87	3,04	1,93	1,54	1,36	1,15	0,86	0,61	NA	
AGX	16,07	9,32	6,4	5,1	3,28	2,67	2,35	2,00	1,52	1,09	0,88	
PC	47,46	47,88	27,95	21,93	16,08	12,01	10,92	9,76	7,74	5,30	4,37	

Tab. 6.1.: Ergebnisse der Geschwindigkeitstests für das YOLOv3 Netzwerk, auf unterschiedlichen Systemen mit unterschiedlichen Auflösungen. Die Werte zeigen die durchschnittliche Bildfrequenz in Bildern pro Sekunde.

Für das YOLOv3 Netzwerk ergibt die Analyse der Bildfrequenzen, dass nur die kleinste Auflösung von 224 x 224 Pixeln eine den Anforderungen genügende Bildfrequenz erreicht. Das gilt nur für den TX2 und den AGX, wobei der AGX bei der Auflösung von 320 x 320 Pixeln die gewünschte Bildfrequenz um 7% unterschreitet. Der AGX erreicht bei den Tests immer über 50% mehr Bildverarbeitungen als der TX2. Der PC schafft bei der Auflösung von 224 x 224 Pixeln deutlich mehr Bilder pro Sekunde, fast die fünffache Menge im Gegensatz zum TX2 und fast die dreifache Menge im Gegensatz zum AGX. Der PC kann bis zu einer Bildgröße von 768 x 768 Pixeln die gewünschte Bildfrequenz erreichen. Der TK1 ist für ein Szenario mit einem so tiefen Netzwerk ungeeignet. Ab einer Auflösung von 512 x 512 Pixeln reicht der Arbeitsspeicher des TK1 nicht mehr aus, um die Berechnung durchzuführen.

System	Auflösung											
	224	320	416	512	608	704	768	800	960	1120	1280	
TK1	22,14	11,88	9,32	5,85	4,90	2,94	2,75	2,58	1,73	1,14	1,13	
TX2	62,49	41,84	30,95	24,13	14,86	12,26	10,80	9,24	7,09	4,93	4,04	
AGX	74,59	61,91	46,18	36,35	22,78	20,83	18,56	16,04	12,39	8,67	7,08	
PC	111,7	93,58	81,16	69,64	62,69	53,16	50,37	47,81	37,63	31,23	26,27	

Tab. 6.2.: Ergebnisse der Geschwindigkeitstest für das YOLOv3-tiny Netzwerk, auf unterschiedlichen Systemen mit unterschiedlichen Auflösungen. Die Werte zeigen die durchschnittliche Bildfrequenz in Bildern pro Sekunde.

Für das YOLOv3-tiny Netzwerk zeigen die Testergebnisse, dass der TK1 bis zu einer Bildgröße von 320 x 320 Pixeln über 10 FPS erreicht, bei 416 x 416 Pixeln erreicht der TK1 immerhin noch über 9 FPS. Der TX2 berechnet mit dem kleinen Netzwerk bei einer Auflösung von 768 x 768 Pixeln fast 11 Bilder pro Sekunde. Der AGX sticht mit einer Bildfrequenz von über 12 FPS bei einer Auflösung von 960 x 960 Pixeln hervor. Der PC verarbeitet selbst bei einer Auflösung von 1280 x 1280 Pixeln über 26 Bilder pro Sekunde.

6.2 Genauigkeitstest

Zum Testen der Genauigkeit liegen 10 122 Bilder vor, auf denen 13 298 Fahrradfahrer abgebildet sind. Die Testdaten bestehen aus Datenpaaren, die jeweils Eingabedaten und Ausgabedaten enthalten. Die Eingabedaten sind die Bilddaten eines Bildes. Die Ausgabedaten sind die Begrenzungsrahmen der auf dem Bild enthaltenen Fahrradfahrer.

Bei der Ermittlung der Genauigkeit werden zwei Faktoren berücksichtigt, die Klassifizierung und die Korrektheit. Wenn das Netzwerk einen Fahrradfahrer klassifiziert, dann wird die Klassifizierung als positiv bezeichnet, andernfalls als negativ. Diese Klassifizierung wird mit den Ausgabedaten verglichen und anschließend als korrekt (engl.: true) oder falsch (engl.: false) bezeichnet. Bei einer Klassifikation können demnach vier Fälle eintreten:

- **Vorhandener** Fahrradfahrer **wird** klassifiziert (engl.: True Positive (TP))
- **Nicht vorhandener** Fahrradfahrer **wird** klassifiziert (engl.: False Positive (FP))
- **Vorhandener** Fahrradfahrer **wird nicht** klassifiziert (engl.: False Negativ (FN))
- **Nicht Vorhandener** Fahrradfahrer **wird nicht** klassifiziert (engl.: True Negativ (TN))

Dabei ist zu beachten, dass das Szenario „Nicht Vorhandener Fahrradfahrer wird nicht klassifiziert“ bei der Detektion von Objekten auf Bildern problematisch ist. Die Anzahl der nicht vorhandenen Fahrradfahrer auf einem Bild lässt sich nur schwer definieren, deswegen wird auf die Ermittlung dieses Wertes bei der Objekt Detektion verzichtet. Mit der Anzahl von TP, FP und FN lassen sich Präzision (engl.: positive predictive value (PPV)), Trefferrate (engl.: true positive rate (TPR)) und der F1-Score berechnen. Die Präzision ist im Bereich der Informationsrückgewinnung als die Anzahl der erhaltenen korrekten Ergebnisse durch die Anzahl aller Ergebnisse definiert und wird wie folgt berechnet:

$$PPV = TP / (TP + FP)$$

Die Trefferrate ist die Anzahl der erhaltenen korrekten Ergebnisse durch die Anzahl aller korrekter Ergebnisse und wird folgendermaßen berechnet:

$$TPR = TP / (TP + FN)$$

Der F1-Score der Statistik dient der Messung der Genauigkeit eines Tests. Zur Berechnung wird das harmonische Mittel aus Präzision und Trefferrate berechnet:

$$F1 = 2 * PPV * TPR / (PPV + TPR)$$

Mit Hilfe der Überschneidung von korrekten und berechneten Begrenzungsrahmen (engl.: Intersection over Union (IoU)) wird ein Ergebnis als korrekt oder falsch gewertet. Dabei zählt eine Überschneidung von über 50% als korrektes Ergebnis bzw. TP, andernfalls als falsches Ergebnis, also FP. Wenn für einen markierten Begrenzungsrahmen kein detekter Begrenzungsrahmen gefunden wird, dann werden die FN hochgezählt. Zusätzlich zur Genauigkeit wird die durchschnittliche IoU ermittelt. Die Werte werden mit der `map` Funktion berechnet, die bereits nach dem ein Training beendet ist verwendet wird.

Für die beiden Netzwerke YOLOv3 und YOLOv3-tiny werden jeweils die Gewichte mit der höchsten Genauigkeit ausgewählt. Mit diesen Gewichten wird die Genauigkeit des Netzwerks für verschiedene Bildgrößen berechnet. Dazu wird in der Konfigurationsdatei die gewünschte Auflösung eingestellt und anschließend wieder die `map` Funktion verwendet. Die Bildgrößen, bei denen die Bildfrequenz über 10 FPS beträgt, werden hierzu verwendet: 224 x 224 Pixel, 320 x 320 Pixel, 608 x 608 Pixel, 768 x 768 Pixel und 960 x 960 Pixel. Die Ergebnisse sind für das Netzwerk YOLOv3 in der Tabelle 6.3 aufgelistet und für das Netzwerk YOLOv3-tiny in der Tabelle 6.4.

Auflösung in Pixel	Präzision	Treffer- rate	F1- Score	IoU	TP	FP	FN
224 x 224	94,37%	64,87%	76,88%	73,50%	8626	515	4672
320 x 320	95,19%	72,65%	82,41%	78,14%	9661	488	3637
608 x 608	95,82%	78,62%	86,37%	79,55%	10455	456	2843
768 x 768	91,91%	79,66%	85,34%	74,41%	10593	933	2705
960 x 960	81,39%	79,63%	80,50%	64,04%	10589	2421	2709

Tab. 6.3.: Ergebnisse des Genauigkeitstests für YOLOv3. Die berechneten Werte für Präzision, Trefferrate, Genauigkeit (F1-Score) und Überschneidung (IoU) sind für verschiedene Netzwerk Auflösungen aufgelistet. Zusätzlich sind die Anzahl der TP, FP und FN aufgeführt.

Auffällig ist, dass beide Netzwerke ihre maximalen Werte bei einer Bildauflösung von 608 x 608 haben. Das ist dieselbe Auflösung, die in der Konfigurationsdatei für die Netzwerke während des Trainings eingestellt war. Die beste Präzision von YOLOv3 liegt bei 95,82%, im Gegensatz zu der Präzision von YOLOv3-tiny mit 92,86%. Die Trefferrate hat ihr Maximum bei 78,62% für YOLOv3 und 75,06% bei YOLOv3-tiny. Die Genauigkeit unterscheidet sich um 3,36% zwischen YOLOv3 mit 86,37% und YOLOv3-tiny mit 83,01%. Die beste durchschnittliche IoU beträgt bei YOLOv3 79,55% und 74,82% bei YOLOv3-tiny.

Auflösung in Pixel	Präzision	Treffer- rate	F1- Score	IoU	TP	FP	FN
224 x 224	86,66%	68,56%	76,56%	65,76%	9117	1403	4181
320 x 320	91,57%	68,81%	78,57%	73,84%	9150	842	4148
608 x 608	92,86%	75,06%	83,01%	74,82%	9981	768	3317
768 x 768	91,04%	73,67%	81,44%	70,64%	9796	964	3502
960 x 960	86,66%	68,56%	76,56%	65,76%	9117	1403	4181

Tab. 6.4.: Ergebnisse des Genauigkeitstests für YOLOv3-tiny. Die berechneten Werte für Präzision, Trefferrate, Genauigkeit (F1-Score) und Überschneidung (IoU) sind für verschiedene Netzwerk Auflösungen aufgelistet. Zusätzlich sind die Anzahl der TP, FP und FN aufgeführt.

6.2.1 Fehleranalyse

Eine manuelle Fehleranalyse wird durchgeführt, um die auftretenden Fehler zu kategorisieren und daraus Limitationen abzuleiten. Dazu werden alle Bilder mit FP und FN untersucht. Die beiden Fälle werden differenziert. Für die Analyse werden die berechneten und markierten Begrenzungsrahmen auf den Bildern visualisiert. Die `map` Funktion wird angepasst, um die berechneten und markierten Begrenzungsrahmen auf die Bilder zu setzen und diese Bilder zu speichern. Dazu wird eine neue Funktion eingeführt, die während der Berechnung der Genauigkeit nach jedem verrechnetem Bild ausgeführt wird, siehe im Anhang Listing B.10. Die nachfolgenden Kategorien treffen auf beide Netzwerke zu, wobei die prozentuale Angabe nur für YOLOv3 gilt.

Die Untersuchung der Bilder mit falsch berechnetem Begrenzungsrahmen ergibt folgende Fehlerkategorien: Fußgänger; bemannte Mopeds, Roller und Dreiräder; Fußgänger die Fahrräder schieben; fehlende Begrenzungsrahmen; doppelte Erkennungen; zu niedrige Überschneidung; Fahrräder; Fahrzeugheck und Sonstige. Die Abbildung 6.1 zeigt in einem Kreisdiagramm die prozentuale Verteilung der FPs nach Kategorie. Am Ende dieses Kapitels sind einige Beispielbilder mit falschen Detektionen aufgeführt, siehe Abbildung 6.4.

Zu der Kategorie „fehlende Begrenzungsrahmen“ zählen fehlerhafte Daten im Datensatz, die bei der Datensatzbereinigung nicht entfernt wurden. „Doppelte Erkennungen“ beschreibt lediglich, dass um einen Fahrradfahrer zwei Begrenzungsrahmen gefunden wurden. In dem Fall hat die Non Maximum Suppression versagt, jedoch ist an dieser Stelle tatsächlich ein Fahrradfahrer, aber nicht zwei. Die Kategorie „Sonstige“ beschreibt hier die Detektion von diversen Gegenständen in der Umgebung als Fahrradfahrer, z.B. einen Teil vom Baum, ein Ast, ein kleines Stück Dreck oder einen Pfeiler in der Ferne.

Diese Kategorien lassen sich in drei Gruppen einteilen. Die erste Gruppe fasst andere Verkehrsteilnehmer zusammen, dazu zählen die Fußgänger und die bemannten

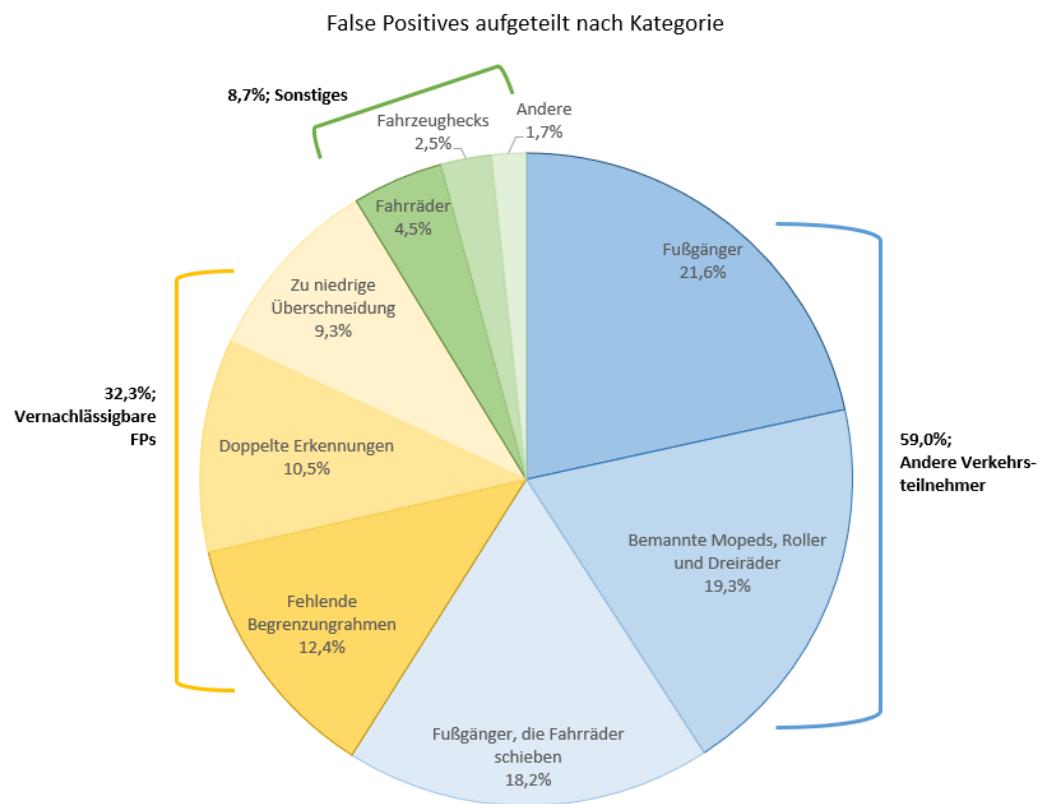


Abb. 6.1.: Falsch berechnete Begrenzungsräume aufgeteilt nach Fehlerkategorien. Zusätzlich sind die Kategorien in Gruppen zusammengefasst.
Quelle: Eigene Darstellung

Mopeds, Roller und Dreiräder. Eine weitere Gruppe sind die vernachlässigbaren FPs, dazu zählen die fehlenden Begrenzungsräume, die doppelten Erkennungen und die zu niedrigen Überschneidungen. Die markierten Begrenzungsräume im Datensatz liegen nicht immer perfekt um einen Fahrradfahrer. Zusammen mit einem nicht ganz korrekt detektierten Begrenzungsräumen ergibt sich eine zu niedrige Überschneidung. Jedoch befindet sich an dieser Stelle ein Fahrradfahrer. Die letzte Gruppe besteht aus unbemannten Fahrrädern, Fahrzeughecks und Sonstiges.

Die Bilder, auf denen Fahrradfahrer nicht detektiert wurden, lassen sich in folgende Ursachen kategorisieren: für Menschen nicht erkennbar; kleine Objekte; größtenteils verdeckte Fahrradfahrer; fast alles gut; alles gut und verschwommene Bilder oder schlechte Lichtverhältnisse. Die Abbildung 6.2 zeigt in einem Kreisdiagramm die prozentuale Verteilung der FNs nach Kategorie. Am Ende dieses Kapitels sind einige Beispielbilder mit fehlenden Begrenzungsräumen in der Abbildung 6.3 aufgeführt. Bei der Kategorie „für Menschen nicht erkennbar“ sind Fahrradfahrer zum Großteil so klein, dass sie selbst mit Bildvergrößerung nicht zu identifizieren sind. Einige Fahrradfahrer sind fast vollständig verdeckt und dadurch nicht erkennbar.

False Negatives aufgeteilt nach Kategorie

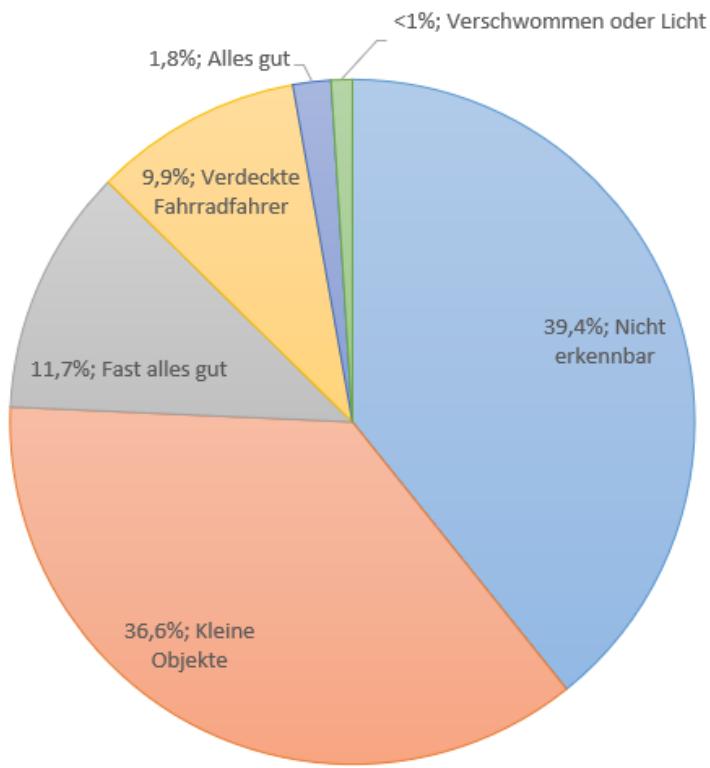


Abb. 6.2.: Nicht erkannte Fahrradfahrer aufgeteilt nach Fehlerkategorien.
Quelle: Eigene Darstellung

Unter die Kategorie „kleine Objekte“ fallen Fahrradfahrer, die für Menschen teilweise nur mit Bildvergrößerung erkennbar sind. Die Begrenzungsrahmen nehmen in diesen Fällen weniger als 2% der Bildbreite und weniger als 5% der Bildhöhe ein. Wobei 42,8% der Begrenzungsrahmen, die unter diese Größe fallen, korrekt erkannt werden. Insbesondere werden kleine Objekte auf Bildern aus dem Tsinghua-Daimler-Datensatz mit der hohen Auflösung von 2048 x 1024 Pixel nicht erkannt. Die Kategorie „fast alles gut“ bedeutet, dass das Bild leicht verschwommen, der Fahrradfahrer seitlich abgebildet, der Begrenzungsrahmen relativ klein, der Fahrradfahrer leicht verdeckt oder mehrere Dinge in Kombination zutreffend sind. Die Kategorie „alles gut“ beschreibt Bilder, die mühelos erkannt werden sollten, aber nicht werden. Auffällig sind dabei zwei Dinge: erstens befinden sich die Füße des Fahrradfahrer nicht an den Pedalen des Fahrrads, sondern werden vom Fahrrad weg ausgestreckt. Zweitens wird die Hand des Fahrradfahrers vor das Gesicht gehalten, weil die Sonne blendet. Bei einem sehr kleinen Teil des Datensatzes tritt ein Fehler bei verschwommenen Bildern und Bildern, die gegen den Sonnenuntergang aufgenommen wurden, auf.

6.3 Auswertung

Die Auswertung erfolgt im ersten Schritt für die Geschwindigkeitstests und die Genauigkeitstests separat. Anschließend werden die Ergebnisse in Zusammenhang gebracht und ausgewertet. Abschließend werden die Erkenntnisse zusammengefasst.

Die Geschwindigkeit fällt überraschend schlecht aus. Das liegt in erster Linie daran, dass das 107 Schichten große YOLOv3 Netzwerk nicht für die Anwendung im eingebetteten Bereich konzipiert wurde. Nur auf der niedrigsten Auflösung überschreiten die beiden neusten Mikrocontroller von NVIDIA den Benchmark von 10 FPS. Je nach Reaktionszeit des Gesamtsystems und Verwendung der Ergebnisse könnte eine niedrigere FPS ausreichend sein und damit einhergehend eine größere Auflösung gewählt werden.

Beim 25 Schicht großen YOLOv3-tiny Netzwerk zeigt sich ein, auch im Vergleich, deutlicher Unterschied zwischen den getesteten Mikrocontrollern. Hier erreicht sogar der TK1 trotz seines Alters bei einer Auflösung von 320 x 320 Pixeln über 11 Bildern pro Sekunde. Der TX2 ist drei bis viermal schneller als der TK1. Die neuste zugängliche Technologie von NVIDIA, der AGX Xavier, schafft es bei einer Auflösung von einem Megapixel (1024 x 1024 Pixel) knapp unter 10 Bilder pro Sekunde zu berechnen. Einerseits zeigt sich ein deutlicher Abstand zwischen der Leistung von GPUs auf Mikrocontrollern und jenen in PCs. Andererseits sind sie trotzdem leistungsfähig. Insbesondere beim Vergleich der Effizienz erzielen Mikrocontroller bessere Werte als Grafikkarten. Der AGX Xavier verbraucht 30 Watt, im Gegensatz dazu verbraucht die GeForce 1080 GTX 270 Watt. Der in Mikrocontrollern verbaute Arbeitsspeicher für die GPUs war, bis auf den TK1, für die hier getesteten Auflösungen vollkommen ausreichend.

Das im Rahmen dieser Arbeit implementierte System verfügt über eine relativ gute Präzision, d.h. die Detektionen, die ausgegeben werden, sind korrekt. Die gewünschte Präzision von 95% wird bei YOLOv3 erreicht, jedoch bei YOLOv3-tiny um 2,1% unterschritten. Die Präzision wird durch die Anzahl der FPs beeinflusst. Durch die Fehleranalyse fällt auf, dass über ein Fünftel der FPs durch nicht korrekte Validierungsdaten entstehen (12,4% fehlende und 9,3% nicht perfekte Begrenzungsrahmen). 58,8% der FPs sind Fußgänger, die z.B. ein Fahrrad schieben, sowie bemannte Zwei- und Dreiräder. Eine Verwechslung zwischen Fußgängern bzw. anderen Fahrradfahrer-ähnlichen Verkehrsteilnehmern und Fahrradfahrern wäre bei einem autonomen Brems- oder Ausweichvorgang ggf. nicht kritisch. Nur 1,7% der fälschlicherweise erkannten Fahrradfahrer sind tatsächlich weder Verkehrsteilnehmer noch andere Objekte. Diese Detektionen können in realen Verkehrsszenarien für Fahrerassistenzsysteme verwirrend sein und unter Umständen zu gefährlichen Verkehrssituationen führen.

Die erzielte Trefferrate von unter 80% ist ungenügend, d.h. jeder fünfte Fahrradfahrer wird nicht erkannt. Die gewünschte Trefferrate von 95% als Ergebnis für diese Arbeit wurde nicht erreicht. Bei der Fehleranalyse fällt deutlich auf, dass Fahrradfahrer, die nur einen kleinen Teil des Bildes einnehmen, nicht erkannt werden. Dieses Phänomen beruht auf der Skalierung des Bildes. Durch das Runterskalieren des Bildes gehen Informationen verloren, so dass nicht mehr genug Informationen übrigbleiben, um die notwendigen Merkmale eines Fahrradfahrers zu erkennen. Das ist ein bekanntes Problem aktueller Objekt Detektoren wie Singh und Davis [SD18] beschreiben. Der Ansatz der Merkmals Pyramiden in YOLOv3 ist nicht ausreichend, um kleine Objekte zu erkennen. Das Runterskalieren der Bildern mit der Auflösung von 2048 x 1024 Pixeln auf 608 x 608 Pixel geht mit einem zu hohen Informationsverlust einher.

Die gewünschte Korrektheit der Lokalisierung auf dem Bild (>75%) wird von YOLOv3 erreicht und von YOLOv3-tiny leicht unterschritten mit 74,82%. Das bedeutet, dass die Fahrradfahrer zum Großteil genau dort auf dem Bild zu finden sind, wo sie detektiert wurden. Das kann insbesondere für die weitere Verarbeitung z.B. mit Sensor Fusion hilfreich sein.

Die Genauigkeit ist mit 86,4% für YOLOv3 und 83,0% für YOLOv3-tiny mangelhaft und kann zu erheblichen Problemen im realen Einsatz führen. Durch die Fehleranalyse wurde festgestellt, dass der Datensatz teilweise fehlerhaft ist und damit zu einer Verfälschung der Ergebnisse führt. Aus diesem Grund wird die Genauigkeit wiederholt berechnet, unter Ausschluss folgender Werte: die Berechnung der Präzision erfolgt unter Ausschluss der FP aus der Gruppe der „vernachlässigbaren FP“; die Berechnung der Trefferrate erfolgt einmal mit Ausschluss der FN der Kategorie „für Menschen nicht erkennbar“ und ein weiteres Mal unter zusätzlichem Ausschluss der FN aus Bildern mit kleinen Objekten. Die Genauigkeit wird schließlich mit der alternativen Präzision und für beide Alternativen der Trefferrate berechnet. Das

Netzwerk	Präzision ohne vernachläs- sigbare	Trefferrate ohne nicht erkennba- re	Trefferrate zusätzlich ohne kleine Objekte	Genauigkeit ohne nicht erkennba- re	Genauigkeit zusätzlich ohne kleine Objekte
YOLOv3	97,1%	85,9%	93,8%	91,2%	95,4%
YOLOv3-tiny	95,0%	83,2%	92,6%	88,8%	93,8%

Tab. 6.5.: Alternative Berechnung der Genauigkeit für die beiden Netzwerke YOLOv3 und YOLOv3-tiny. Die Berechnung erfolgt unter Ausschluss bestimmter Kategorien.

sind simulierte Ergebnisse, die jedoch durch einen sauberen Datensatz und einer höheren Auflösung des Netzwerks erreichbar wären. Allerdings erreicht auch bei diesen simulierten Werten keines der beiden Netzwerke die geforderte Trefferrate.

Grundsätzlich lässt sich beim Vergleich zwischen YOLOv3 und YOLOv3-tiny festhalten, dass beide Netzwerke mit den im Rahmen dieser Arbeit trainierten Gewichten ähnlich in ihrer Genauigkeit sind; hier beläuft sich der Unterschied auf 3,5%. Das ist in Anbetracht der Tiefe des Netzwerks (25 Schichten im Gegensatz zu 107 Schichten) erstaunlich. Bei der Geschwindigkeit liegt YOLOv3-tiny mit einer durchschnittlich sechs Mal schnelleren Verarbeitung pro Bild deutlich vorne. Abschließend soll erörtert werden, welches der beiden Netzwerke im eingebetteten System vorteilhafter ist. Dazu werden die maximal möglichen Auflösungen bei Einhaltung einer Bildfrequenz von mindestens 10 FPS betrachtet. Für YOLOv3 überschreiten der AGX und TX2 die Bildfrequenz von 10 FPS nur bei einer Auflösung von 224 x 224 Pixeln. Bei dieser Auflösung liegt die Genauigkeit bei 77%. Für YOLOv3-tiny liegt die höchste Genauigkeit bei einer Auflösung von 608 x 608 Pixeln, und zwar mit 83%. Der AGX und der TX2 erreichen bei dieser Auflösung eine Bildfrequenz von über 10 FPS. Insgesamt liefert das Netzwerk YOLOv3-tiny die besseren Ergebnisse für den Einsatz im eingebetteten System.

Um ein für den Straßenverkehr geeigneten Objekt Detektor zu erhalten, müssen insbesondere folgende Dinge beachtet werden: Die von Fahrerassistenzsystemen verlangte Bildfrequenz ergibt sich aus der Auflösung und der Komplexität des künstlichen neuronalen Netzwerks in Zusammenhang mit der Geschwindigkeit der Hardware. Je nachdem, welche Kamera verwendet wird, sollte nur mit dieser Auflösung trainiert werden. Der Datensatz für das Training sollte hauptsächlich aus Bildern mit dieser Auflösung bestehen. Im Datensatz muss jedes zu erkennende Objekt korrekt markiert sein. Die Auflösung des Netzwerkes sollte möglichst hoch und nah an der Auflösung der Kamerabilder sein. Je nach Auflösung der Kamerabilder und bzw. oder des Netzwerks werden Objekte ab einer bestimmten Entfernung nicht zuverlässig erkannt. Diese maximale Entfernung muss ermittelt und berücksichtigt werden.

Zusammenfassend lassen sich folgende Punkte festhalten. Die Erreichung einer gewünschten Bildfrequenz hängt primär von der Tiefe des Netzwerks und der Geschwindigkeit der verwendeten GPU abhängt. Die Genauigkeit hängt primär von der Bildauflösung des Netzwerks ab. Anhand der Testergebnisse lässt sich sehr gut erkennen, dass eine reziproke Proportionalität zwischen Genauigkeit und Schnelligkeit besteht. Die hier erreichten Bildfrequenzen können im Straßenverkehr durchaus verwendet werden. Für eine endgültige Aussage ist jedoch das gesamte Timing entscheidend, von der Aufnahme des Bildes bis hin zur abschließenden Verarbeitung durch ein Fahrerassistenzsystem bzw. der Ausführung einer ggf. notwendigen Aktion. Der verwendete Datensatz ist suboptimal in Bezug auf die Korrektheit der Markierungen und Vielfalt der Szenarien. Insbesondere das Erkennen von Fahrradfahrern auf Bildern mit hoher Auflösung ist ein großes Problem. Dadurch ist die Genauigkeit mit 83% für den Straßenverkehr mangelhaft. Insgesamt ist der Einsatz des Fahrradfahrer Detektionssystem (FFDS) in einem Kraftfahrzeug im jetzigen Zustand nicht geeignet.

6.4 Limitationen und Handlungsempfehlungen

In diesem Unterkapitel werden die Einschränkungen bei der Detektion von Fahrradfahrern verdeutlicht und Lösungsansätze genannt. Die verwendeten Netzwerke können in Echtzeitszenarien nicht ohne GPU Unterstützung verwendet werden, da die Bildfrequenz zu niedrig ist. Die Tests auf den Mikrocontrollern zeigen eindeutig, dass die verwendete GPU ausschlaggebend ist, um eine ausreichend hohe Bildfrequenz zu erreichen. Eine Alternative zur GPU Unterstützung kann die parallele Verarbeitung mit FPGAs oder ASICs sein, z.B. mit dem OpenVino Framework von Intel. Dabei könnte auch untersucht werden, wie hoch die Auflösung von Gleitkommazahlen sein muss, um genaue Ergebnisse zu erhalten. Eine weitere günstige und effektive Alternative zur GPU Unterstützung können USB Sticks sein, die speziell für die Berechnung von neuronalen Netzwerken erstellt werden, wie z.B. Intel® Movidius™ Neural Compute Stick. Beide Alternativen wurden im Rahmen dieser Arbeit nicht betrachtet.

Im aktuellen FFDS werden Fußgänger; Fußgänger, die ein Fahrrad schieben; Roller; Mopeds und Dreiräder fälschlicherweise als Fahrradfahrer erkannt. Diesem Problem kann mit einem erweiterten Datensatz entgegengewirkt werden, der mehr Bilder dieser Fälle beinhaltet. Die Fehleranalyse zeigt, dass der Datensatz nicht vielfältig genug ist. Der Datensatz muss entsprechend mit Bildern erweitert werden, die folgendes ausweisen: verschiedene Umgebungen; unterschiedliche Lichteinstrahlung und Lichtquellen; teilweise bis größtenteils verdeckte Fahrradfahrer (jedoch eindeutig identifizierbar); verschwommene Bilder; Bilder mit Fahrradfahrer, die einen exakten Begrenzungsrahmen haben; Fahrradfahrer in verschiedenen Posen und Bilder mit Fahrrädern. Durch das Training mit einem erweiterten Datensatz können die falschen und die fehlenden Detektionen voraussichtlich gesenkt werden.

Das größte Problem des FFDS ist die Tatsache, dass kleine Objekte nicht detektiert werden. Insbesondere bei höheren Geschwindigkeiten im Straßenverkehr ist es ein schwerwiegendes Problem, dass Fahrradfahrer ab einer bestimmten Entfernung auf Bildern mit höherer Auflösung nicht erkannt werden. Der Ansatz der Merkmals Pyramiden - wie von YOLOv3 verwendet - ist unzureichend. Ein Forschungsansatz wäre, das große Bild in kleinere zu Teilen und diese Teilbilder mit dem Objekt Detektor zu analysieren. Dieser Ansatz führt zu einer niedrigeren Bildfrequenz. Ein weiterer Forschungsansatz könnte die Skalierung der Bilder auf eine höhere Auflösung sein. Wobei folgende Frage geklärt werden muss: wie weit darf ein Bild runterskaliert werden, damit ein künstliches neuronales Netzwerk kleine Objekte darauf erkennen kann. Für den Betrieb des FFDS in einer deutlich höheren Auflösung muss das Netzwerk neu trainiert werden, das Training beansprucht mehr Zeit und das Detektieren im Betrieb dauert ebenfalls länger.

Im aktuellen Fahrradfahrer Detektionssystem können einige Qualitätsanforderungen nicht erfüllt werden, insbesondere im Bereich Geschwindigkeit und Genauigkeit. Es gibt mehrere Optionen, diese Werte zu optimieren. Um die Geschwindigkeit zu verbessern, empfiehlt es sich eine Netzwerkarchitektur auf einem FPGA zu realisieren. Dabei steht im Fokus, die Anzahl der Bits für eine Gleitkommazahl zu optimieren. Um die Genauigkeit zu verbessern, ist ein erweiterter Datensatz und das Training in einer höheren Auflösung nötig.

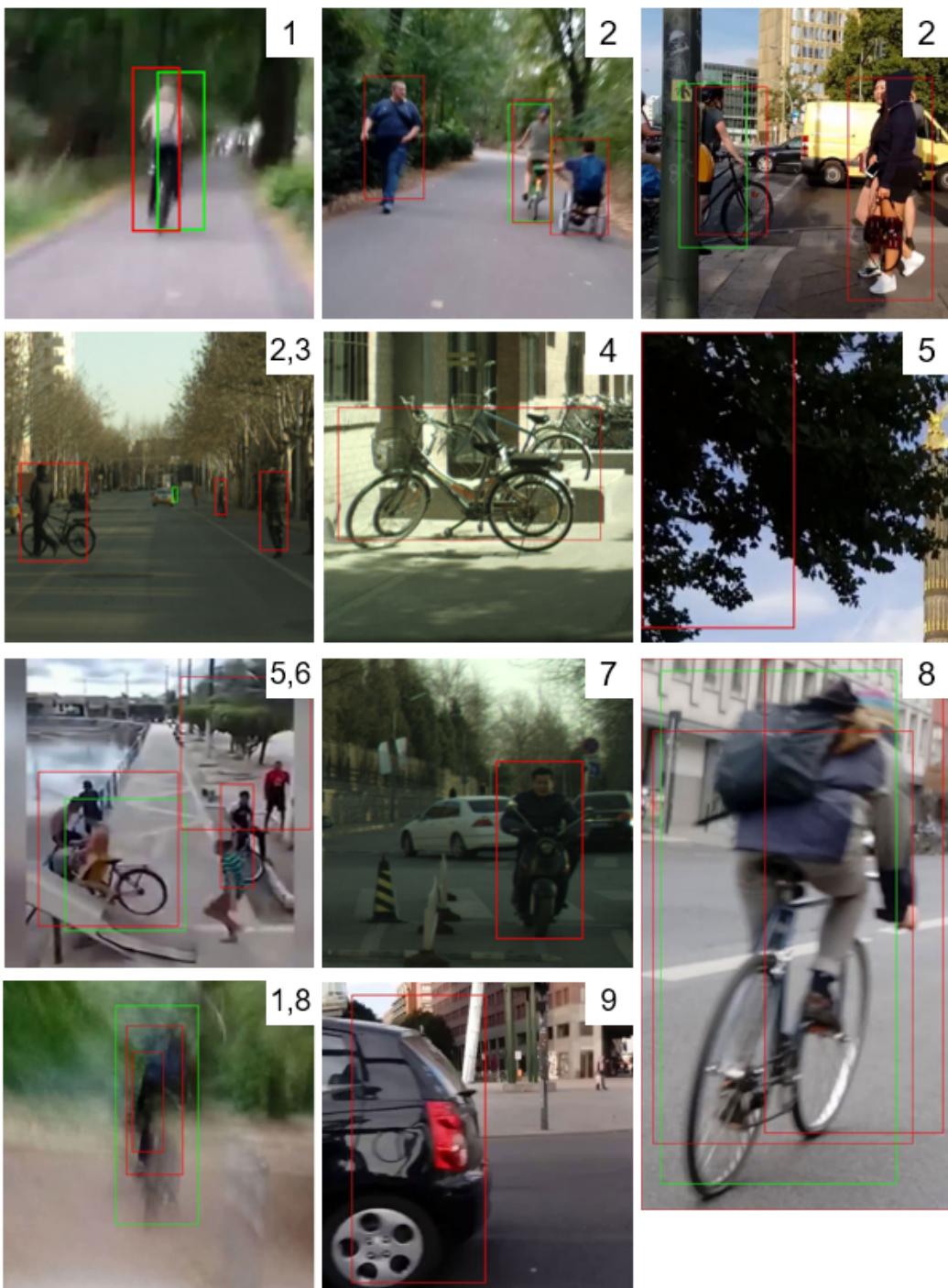


Abb. 6.3.: Beispielbilder auf denen fälschlicherweise Fahrradfahrer detektiert wurden (roter Begrenzungsrahmen). Die Nummern zeigen die Fehlerkategorien an, zu denen sich die Fehler auf einem Bild zuordnen lassen:

- 1 = Überschneidung
- 2 = Fußgänger
- 3 = Fußgänger, die Fahrräder schieben
- 4 = Fahrräder
- 5 = Sonstige
- 6 = Fehlende Begrenzungsrahmen
- 7 = Moped
- 8 = Mehrfachdetektion
- 9 = Fahrzeugheck

Quelle: Adaptiert von [Li+16]

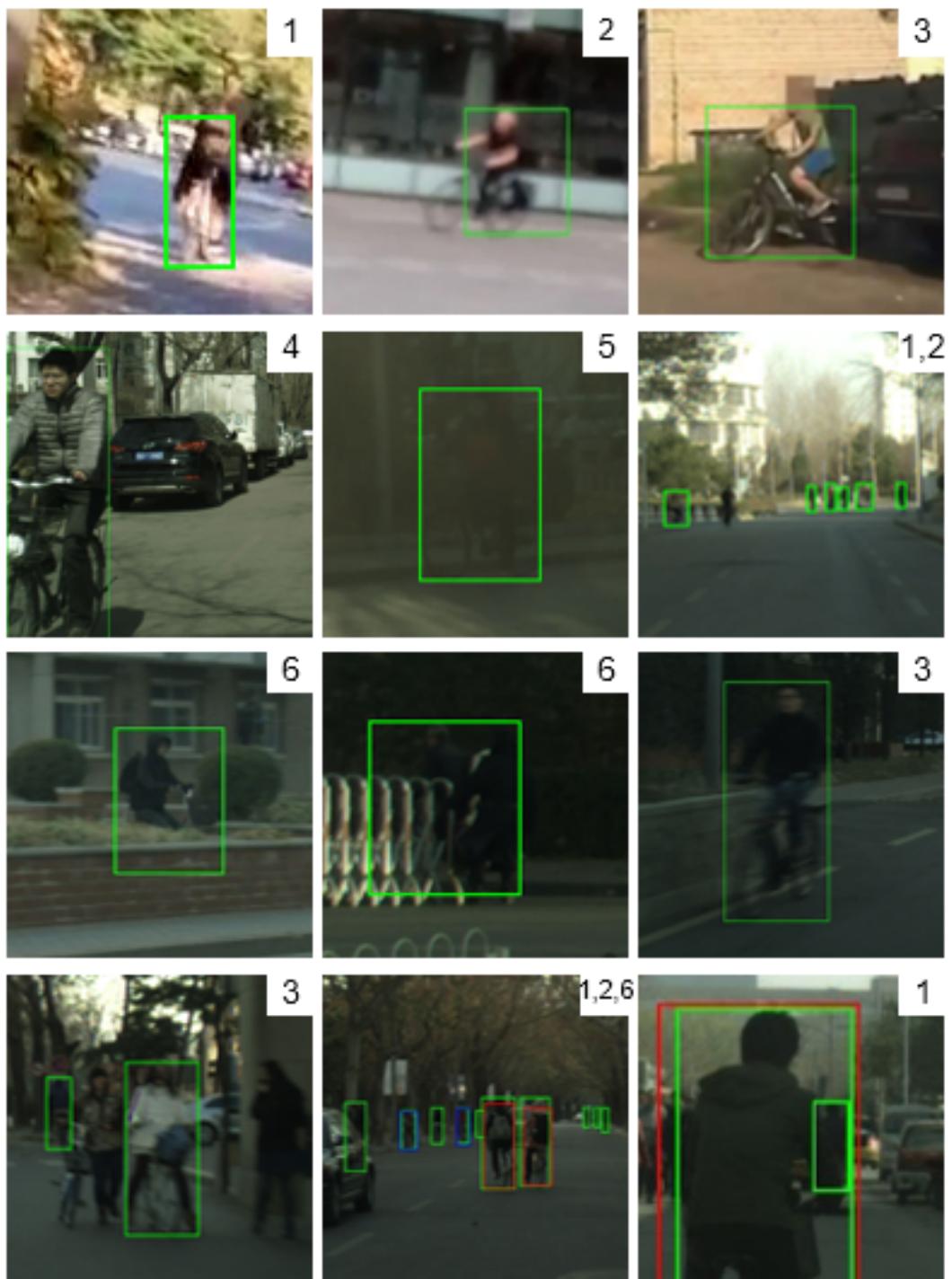


Abb. 6.4.: Beispielbilder, auf denen das Netzwerk keine Fahrradfahrer detektieren konnte. Die grünen Begrenzungsrahmen sind aus den Markierungsdateien im Datensatz. Die roten Begrenzungsrahmen wurden vom Netzwerk berechnet. Die blauen Begrenzungsrahmen zeigen eine Detektion mit einer Wahrscheinlichkeit von unter 25%. Die Nummern zeigen die Fehlerkategorien an, zu denen sich die Fehler auf einem Bild zuordnen lassen:

1 = Nicht erkennbar

2 = Kleine Objekte

3 = Fast alles gut

4 = Alles gut

5 = Licht

6 = Verdeckt

Quelle: Adaptiert von [Li+16]

Zusammenfassung

Ziel dieser Arbeit ist es, dem Objekt Detektor YOLOv3 mit Transfer Learning beizubringen, Fahrradfahrer auf Kamerabildern zu erkennen und auf dem VeloxCar, einer Plattform zum Entwickeln von autonomen Technologien, zu implementieren. Hiermit soll der Fragestellung nachgegangen werden, wie zuverlässig und schnell Fahrradfahrer auf Kamerabildern mit einem künstlichen neuronalen Netzwerk detektiert werden können.

Hochautomatisiertes Fahren ist ein stark präsenzes Thema in der Forschung unter Automobilherstellern, ein Schwerpunktthema ist u.a. das Erkennen von anderen Verkehrsteilnehmern. Zum aktuellen Zeitpunkt ist das Ergebnis in Bezug auf das Erkennen von Fahrradfahrern noch mangelhaft. Diese Arbeit zeigt, dass durch die Verwendung moderner Technologien Fahrradfahrer auf Kamerabildern grundsätzlich erkannt werden, damit diese Daten anschließend von Notbremsassistenten oder anderen Fahrerassistenzsystemen genutzt werden können.

Als theoretische Basis für diese Arbeit dienen aktuelle Erkenntnisse zum maschinellen Lernen, zur Struktur künstlicher neuronaler Netzwerke und Objekt Detektoren. Mit Hilfe dieser Grundlagen sowie einer Anforderungsanalyse mit Anwendungsfällen werden Anforderungen an ein Fahrradfahrer Detektionssystem erstellt, die sich in funktionale und nicht funktionale Anforderungen unterteilen lassen.

Grundsätzlich wird im Laufe dieser Arbeit mit Hilfe des Transfer Learnings einem künstlichen neuronalen Netzwerk beigebracht, Fahrradfahrer zu detektieren. Die Auswahl der vortrainierten Netzwerke fällt aufgrund ihrer Schnelligkeit auf die YOLOv3 und YOLOv3-tiny Architekturen. Genutzt wird das Framework Darknet, weil es quelloffen und in C und CUDA geschrieben ist. Ein Fork von Darknet wird für die Berechnung der Ankerrahmen und der Genauigkeit verwendet. Die für das Training verwendeten Daten sind ausschlaggebend für die abschließende Genauigkeit. Um eine höhere Varianz der Bilddaten zu erhalten, wird der Tsinghua-Daimler-Datensatz mit eigenen Daten aus YouTube und selbstaufgenommene Videos erweitert. Zur Erstellung der Datenpaare aus den Videos wird Darknet mit einer Funktion ergänzt, die aus Detektionen von Menschen und Fahrrädern die Begrenzungsrahmen für Fahrradfahrer berechnet und in Markierungsdateien speichert. Alle Bilddaten werden auf Qualitätskriterien überprüft. Die effizientesten Werte der Trainingsvariablen - der Lernrate, des Gewichtszerfalls und des Momentum - werden beim trainieren mit Validierungsdaten gefunden. Der Trainingsablauf orientiert sich an Early-Stopping-

Kriterien und endet, wenn die Lernrate nicht weiter sinkt.

Die genutzte Hardware des im VeloxCar eingebetteten Systems beschränkt sich auf die Kamera und den Mikrocontroller NVIDIA Jetson TX2. Die drei Softwarekomponenten Kameraanbindung, Busanbindung und Detektor befinden sich auf dem Mikrocontroller und verwenden die vorhandenen Komponenten Camera Node und ROS Master. Mit ROS wird ein Beobachter Entwurfsmuster implementiert, um Kameradaten zu erhalten und die Ergebnisdaten zu veröffentlichen.

Abschließend wird das System auf dem VeloxCar und auf verschiedenen Rechnern getestet. Dabei zeigt sich, dass nicht alle Anforderungen gleichzeitig erfüllt werden können und ein Zusammenhang zwischen Genauigkeit, Hardware, Geschwindigkeit und Netzwerkgröße besteht. Im Detail liegen die besten Werte für ein eingebettetes System auf einem NVIDIA Jetson AGX Xavier mit YOLOv3-tiny bei über 22 FPS Bildfrequenz, einer durchschnittlichen Überschneidung von 74,8% und 83,0% Genauigkeit. Die Genauigkeit setzt sich aus 92,9% Präzision und 75,1% Trefferrate zusammen.

Die Schwäche des Fahrradfahrer Detektionssystems ist die Genauigkeit und lässt sich insbesondere durch einen fehlerhaften Datensatz und einer zu niedrigen Auflösung des Netzwerks erklären. Der fehlerhafte Datensatz zeigt sich durch fehlende Begrenzungsrahmen, Begrenzungsrahmen auf nicht erkennbaren Fahrradfahrern und falsch positionierten Begrenzungsrahmen. Die niedrige Auflösung des Netzwerks führt zu einem Großteil der fehlenden Erkennungen und lässt sich auf Fahrradfahrer zurückführen, die weiter entfernt sind und dementsprechend nur einen sehr kleinen Bildausschnitt einnehmen.

Um das System zu optimieren, bietet diese Arbeit mehrere Ansätze für weiterführende Arbeiten. Insbesondere muss das System in Hinsicht auf das Erkennen von kleineren Objekten optimiert werden, um die Genauigkeit zu steigern. Für die Erhöhung der Geschwindigkeit könnte die Netzwerkarchitektur in einem FPGA realisiert werden und dabei die Auflösung für Gleitkommazahlen optimiert werden.

Die wichtigsten Aspekte für das Erstellen eines Objekt Detektors für den Straßenverkehr sind das Gleichhalten der Bildauflösung von Kamera, Trainingsbildern und Netzwerk; die Verwendung eines vollständig und korrekt markierten Datensatzes; die passende Wahl der Hardware und Tiefe des Netzwerkes je nach zeitlichen Anforderungen, sowie die Berücksichtigung der maximalen Entfernung. Zusammengefasst zeigt sich, dass das Erkennen von Fahrradfahrern auf Kamerabildern mit einem künstlichen neuronalen Netzwerk grundsätzlich möglich ist. Die Schwächen des Systems, z. B. die fehlende Genauigkeit, lassen sich z.B. durch bessere Trainingsdaten optimieren. Um ein treffsicheres, präzises und genaues Fahrradfahrer Detektionsystem für den realen Straßenverkehr zu entwickeln, obliegt es der Forschung und insbesondere der Automobilindustrie, Ressourcen zu investieren um diese Technologie wortwörtlich straßentauglich zu machen. Die Relevanz und Notwendigkeit ist akuter denn je, wie steigende Unfallstatistiken beweisen.

Literaturverzeichnis

- [And07] Raymond Anderson. *The credit scoring toolkit: theory and practice for retail credit risk management and decision automation*. Oxford University Press, 2007 (zitiert auf Seite 7).
- [Aut15] Verband der Automobilindustrie. „Automatisierung–Von Fahrerassistenzsystemen zum automatisierten Fahren“. In: *VDA, Berlin* (2015) (zitiert auf Seite 1).
- [Ber17] Der Polizeipräsident in Berlin. *Sonderuntersuchung Radfahrverkehrsunfälle in Berlin*. 2017, S. 2 (zitiert auf Seite 2).
- [Bis+95] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995 (zitiert auf den Seiten 7, 10).
- [Hai16] Tobias Haist. „Autonomes Fahren: Eine kritische Beurteilung der technischen Realisierbarkeit“. In: (2016) (zitiert auf Seite 1).
- [Har12] Peter Harrington. *Machine learning in action*. Bd. 5. Manning Greenwich, 2012 (zitiert auf Seite 5).
- [He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren und Jian Sun. „Deep residual learning for image recognition“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, S. 770–778 (zitiert auf Seite 17).
- [HK17] Mark Harmon und Diego Klabjan. „Activation ensembles for deep neural networks“. In: *arXiv preprint arXiv:1702.07790* (2017) (zitiert auf Seite 12).
- [IS15] Sergey Ioffe und Christian Szegedy. „Batch normalization: Accelerating deep network training by reducing internal covariate shift“. In: *arXiv preprint arXiv:1502.03167* (2015) (zitiert auf Seite 44).
- [Jam+13] Gareth James, Daniela Witten, Trevor Hastie und Robert Tibshirani. *An introduction to statistical learning*. Bd. 112. Springer, 2013 (zitiert auf Seite 7).
- [KSH12] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. „Imagenet classification with deep convolutional neural networks“. In: *Advances in neural information processing systems*. 2012, S. 1097–1105 (zitiert auf Seite 2).
- [Kuh09] Jan Kuhlmann. „Ausgewählte Verfahren der Holdout-und Kreuzvalidierung“. In: *Methodik der empirischen Forschung*. Springer, 2009, S. 537–546 (zitiert auf Seite 9).

- [Li+14] Mu Li, Tong Zhang, Yuqiang Chen und Alexander J Smola. „Efficient mini-batch training for stochastic optimization“. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2014, S. 661–670 (zitiert auf Seite 45).
- [Li+16] Xiaofei Li, Fabian Flohr, Yue Yang et al. „A new benchmark for vision-based cyclist detection“. In: *2016 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2016, S. 1028–1033 (zitiert auf den Seiten 38, 39, 76, 77).
- [Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997 (zitiert auf Seite 6).
- [MRT18] Mehryar Mohri, Afshin Rostamizadeh und Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018 (zitiert auf Seite 7).
- [PR15] Klaus Pohl und Chris Rupp. *Basiswissen Requirements Engineering: Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level*. dpunkt. verlag, 2015 (zitiert auf Seite 29).
- [Pre98] Lutz Prechelt. „Automatic early stopping using cross validation: quantifying the criteria“. In: *Neural Networks* 11.4 (1998), S. 761–767 (zitiert auf Seite 10).
- [PY10] Sinno Jialin Pan und Qiang Yang. „A survey on transfer learning“. In: *IEEE Transactions on knowledge and data engineering* 22.10 (2010), S. 1345–1359 (zitiert auf Seite 33).
- [Red+16] Joseph Redmon, Santosh Divvala, Ross Girshick und Ali Farhadi. „You only look once: Unified, real-time object detection“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, S. 779–788 (zitiert auf Seite 35).
- [Red16] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. <http://pjreddie.com/darknet/>. 2013–2016 (zitiert auf Seite 37).
- [RF16] Joseph Redmon und Ali Farhadi. „YOLO9000: Better, Faster, Stronger“. In: *arXiv preprint arXiv:1612.08242* (2016) (zitiert auf Seite 35).
- [RF18] Joseph Redmon und Ali Farhadi. „YOLOv3: An Incremental Improvement“. In: *arXiv* (2018) (zitiert auf den Seiten 34, 35, 45).
- [RHW86] David E Rumelhart, Geoffrey E Hinton und Ronald J Williams. „Learning representations by back-propagating errors“. In: *nature* 323.6088 (1986), S. 533 (zitiert auf Seite 2).
- [Ros58] Frank Rosenblatt. „The perceptron: a probabilistic model for information storage and organization in the brain.“ In: *Psychological review* 65.6 (1958), S. 386 (zitiert auf den Seiten 7, 12).
- [Rup06] Chris Rupp. „die SOPHISTen“. In: *Requirements-Engineering und Management, 4th ed.* Carl Hanser Verlag (2006) (zitiert auf Seite 30).
- [SD18] Bharat Singh und Larry S Davis. „An analysis of scale invariance in object detection snip“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, S. 3578–3587 (zitiert auf Seite 72).
- [Spi18] M Spitzer. „Meditation, Mäuse und Myelin“. In: *Nervenheilkunde* 37.10 (2018), S. 745–748 (zitiert auf Seite 11).

- [WP17] Jason Wang und Luis Perez. „The effectiveness of data augmentation in image classification using deep learning“. In: *Convolutional Neural Networks Vis. Recognit* (2017) (zitiert auf Seite 42).
- [Xia+13] Wei Xia, Csaba Domokos, Jian Dong, Loong-Fah Cheong und Shuicheng Yan. „Semantic segmentation without annotating segments“. In: *Proceedings of the IEEE international conference on computer vision*. 2013, S. 2176–2183 (zitiert auf Seite 17).
- [Zei12] Matthew D Zeiler. „ADADELTA: an adaptive learning rate method“. In: *arXiv preprint arXiv:1212.5701* (2012) (zitiert auf Seite 8).
- [Zel94] Andreas Zell. *Simulation neuronaler netze*. Bd. 1. 5.3. Addison-Wesley Bonn, 1994 (zitiert auf Seite 8).

Webseiten

- [Fah18] Allgemeiner Deutscher Fahrrad-Club. *ADFC Fakten-Überblick für Fahrradunfälle*. 2018. URL: <https://adfc-berlin.de/radverkehr/sicherheit/information-und-analyse/121-fahrradunfaelle-in-berlin-unfallstatistik/153-adfc-fakten-ueberblick-fuer-fahrradunfaelle.html> (besucht am 28. Aug. 2018) (zitiert auf Seite 2).
- [Rat16] Uwe Rattay. *Notbremsassistenten mit Fußgänger- und Radfahrer-Erkennung*. 2016. URL: https://www.adac.de/infotestrat/tests/assistenzsysteme/fussgae%20ngererkennung_2016/default.aspx (besucht am 28. Aug. 2018) (zitiert auf Seite 2).
- [Red18] Jospeh Redmon. *YOLO: Real-Time Object Detection*. 2018. URL: <https://pjreddie.com/darknet/yolo/> (besucht am 22. Apr. 2019) (zitiert auf Seite 34).
- [Tul19] TullyFoote. *Documentation - ROS Wiki*. 2019. URL: <https://wiki.ros.org/> (besucht am 11. Mai 2019) (zitiert auf Seite 53).
- [Unb18] Unbekannt. *Neural Networks*. 2018. URL: https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/neural_networks.html (besucht am 28. Nov. 2018) (zitiert auf Seite 11).

Tabellen

In diesem Kapitel sind die in der Arbeit erwähnten Tabellen aufgeführt.

Rechner	CPU	GPU	CUDA	cuDNN	OpenCV
ODROID XU4	Exynos 5422 Octa big.LITTLE ARM Cortex-A15 @ 2.0 GHz quad-core and Cortex-A7 quad-core CPUs	NA	NA	NA	3.3.1
NVIDIA Jetson TK1	NVIDIA "4-Plus-1" 2.32GHz ARM quad-core Cortex-A15 CPU with Cortex-A15	192-core Kepler @ 852MHZ	NVIDIA 6.5	NA	3.3.1
NVIDIA Jetson TX2	ARM Cortex-A57 (quad-core) @ 2GHz + NVIDIA Denver2 (dual-core) @ 2GHz	256-core Pascal @ 1300MHz	NVIDIA 9.0	7.1	3.3.1
NVIDIA Jetson AGX Xavier	8-core NVIDIA Carmel 64-bit ARMv8.2 @ 2,27GHz	512-core Volta @ 1377MHz with 64 TensorCores	NVIDIA 10.0	7.3	3.3.1
PC mit GeForce 1080 GTX	AMD Ryzen 7 1800X Eight-Core Processor 3,7GHz	2560-core GP104 @ 1607MHZ	NVIDIA 10.0	7.2	3.2.0

Tab. A.1.: Die verwendeten Systeme der Geschwindigkeitstests. Aufgelistet sind die Namen der Mikrocontroller bzw. der Grafikkarte, die Architektur der CPU und GPU und die Versionen der verwendeten Bibliotheken. Referenziert in Kapitel 6.1.

YOLOv3									
Auflösung		Kamera			Video				
		TK1	ODROID	TX2	TK1	ODROID	TX2	GeForce	AGX
Test 1	224	3,63	--	9,84	3,61	0,05	10,25	47,46	16,07
		870	--	785	883	--	791	1.021	1.374
Test 2	320	2,04	--	5,38	2,03	--	5,61	47,88	9,32
		1.106	--	880	1.122	--	950	1.243	1.566
Test 3	416	1,36	--	3,70	1,35	--	3,87	27,95	6,40
		1.408	--	1.228	1.420	--	1.237	1.541	1.775
Test 4	512	NA	--	2,88	NA	--	3,04	21,93	5,10
		NA	--	1.582	NA	--	1.598	1.891	2.005
Test 5	608	NA	--	1,84	NA	0,01	1,93	16,08	3,28
		NA	--	2.086	NA	--	2.105	2.475	2.451
Test 6	704	NA	--	1,47	NA	--	1,54	12,01	2,67
		NA	--	2.573	NA	--	2.604	2.951	2.864
Test 7	768	NA	--	1,30	NA	--	1,36	10,92	2,35
		NA	--	2.971	NA	--	3.007	3.585	3.168
Test 8	800	NA	--	1,11	NA	--	1,15	9,76	2,00
		NA	--	3.207	NA	--	3.231	3.733	3.234
Test 9	960	NA	--	0,84	NA	--	0,86	7,74	1,52
		NA	--	4.350	NA	--	4.369	4.889	4.332
Test 10	1120	NA	--	0,60	NA	--	0,61	5,30	1,09
		NA	--	5.754	NA	--	5.768	6.295	5.658
Test 11	1280	NA	--	NA	NA	--	NA	4,37	0,88
		NA	--	NA	NA	--	NA	7.989	7.571

Tab. A.2.: Ergebnisse der Geschwindigkeitstests mit dem Netzwerk YOLOv3. Getestet wurden die Schnelligkeit und der Verbrauch des Arbeitsspeichers der GPU auf verschiedenen Systemen. Als Bilddatenquelle diente die Kamera und eine Videodatei. Neben der Testnummer ist die Zahl angegeben, die für die Breite und die Höhe der Bildauflösung verwendet wird. Je zwei Zeilen gehören zu einem Test. Die obere Zeile enthält die durchschnittliche Bildfrequenz in FPS und die untere den verbrauchten Arbeitsspeicher der GPU. Referenziert in Kapitel 6.1.

YOLOv3-tiny										
		Kamera			Video					
Auflösung		TK1	ODROID	TX2	TK1	ODROID	TX2	GeForce	AGX	
Test 1	224	21,37	0,64	32,6	22,14	0,62	62,49	111,72	74,59	
		127	--	522	140	--	551	429	878	
Test 2	320	12,11	0,29	31,39	11,88	0,29	41,84	93,58	61,91	
		127	--	530	140	--	554	463	876	
Test 3	416	9,42	0,17	27,04	9,32	0,17	30,95	81,16	46,18	
		144	--	541	157	--	555	507	871	
Test 4	512	5,92	0,11	21,49	5,85	0,11	24,13	69,64	36,35	
		217	--	545	229	--	554	565	875	
Test 5	608	4,95	0,07	13,7	4,9	0,07	14,86	62,69	22,78	
		311	--	554	324	--	569	641	904	
Test 6	704	3,01	--	11,33	2,94	--	12,26	53,16	20,83	
		412	--	562	424	--	595	727	904	
Test 7	768	2,82	--	10,03	2,75	--	10,8	50,37	18,56	
		492	--	567	504	--	598	797	1.002	
Test 8	800	2,65	--	8,63	2,58	--	9,24	47,81	16,04	
		534	--	565	545	--	594	841	997	
Test 9	960	1,76	--	6,52	1,73	--	7,09	37,63	12,39	
		755	--	563	765	--	596	1.011	1.151	
Test 10	1120	1,17	--	4,44	1,14	--	4,93	31,23	8,67	
		1.039	--	692	1.048	--	702	1.261	1.640	
Test 11	1280	1,33	0,02	3,67	1,13	0,02	4,04	26,27	7,08	
		1.358	--	966	1.369	--	978	1.499	1.266	

Tab. A.3.: Ergebnisse der Geschwindigkeitstests mit dem Netzwerk YOLOv3-tiny. Getestet wurden die Schnelligkeit und der Verbrauch des Arbeitsspeichers der GPU auf verschiedenen Systemen. Als Bilddatenquelle diente die Kamera und eine Videodatei. Neben der Testnummer ist die Zahl angegeben, die für die Breite und die Höhe der Bildauflösung verwendet wird. Je zwei Zeilen gehören zu einem Test. Die obere Zeile enthält die durchschnittliche Bildfrequenz in FPS und die untere den verbrauchten Arbeitsspeicher der GPU. Referenziert in Kapitel 6.1.

Quellcode Auszüge

In diesem Kapitel sind alle in der Arbeit erwähnten Listings aufgeführt. Der Code ist in digitaler Form der Arbeit beigefügt und im GitLab der Beuth-Hochschule unter <https://gitlab.beuth-hochschule.de/master-ffds/masterarbeit> für angemeldete Benutzer einsehbar.

```
1 File f = new File("path/to/labeldata/directory");
2 File[] fileArray = f.listFiles();
3 for (int i = 0; i < fileArray.length; i++) {
4     if (fileArray[i].isDirectory()) {
5         continue;
6     }
7     try {
8         Object obj = parser.parse(new FileReader(fileArray[
9             i]));
10        String groundtruthfilename = fileArray[i].
11            getAbsolutePath().replace(".json", ".txt")
12            .replaceFirst("labelData", "labels").replace(
13                "labelData", "leftImg8bit");
14        FileWriter file = new FileWriter(
15            groundtruthfilename);
16        JSONObject jsonObject = (JSONObject) obj;
17        JSONArray children = (JSONArray) jsonObject.get(""
18            "children");
19        @SuppressWarnings("unchecked")
20        Iterator<JSONObject> iterator = children.iterator()
21            ;
22        AtomicBoolean firstline = new AtomicBoolean(true);
23        while (iterator.hasNext()) {
24            JSONObject child = iterator.next();
25            String id = (String) child.get("identity");
26            System.out.println(id + "\n" + i + "\n");
27            if (!id.equals("cyclist")) {
28                continue;
29            }
30            double dw = ((double) (1. / 2048));
```

```

25     double dh = (double) (1. / 1024);
26     double x = (((long) child.get("mincol")) + ((
27         long) child.get("maxcol"))) / 2) - 1;
28     double y = (((long) child.get("minrow")) + ((
29         long) child.get("maxrow"))) / 2) - 1;
30     double w = ((long) child.get("maxcol")) - ((long
31         ) child.get("mincol"));
32     double h = ((long) child.get("maxrow")) - ((long
33         ) child.get("minrow"));
34     x = (double) (x * dw);
35     w = w * dw;
36     y = y * dh;
37     h = h * dh;
38     file.write(firstline.get() ? String.format("0 %s
39             %s %s %s", x, y, w, h)
40             : String.format("\n0 %s %s %s %s", x, y, w
41                     , h));
42     firstline.set(false);
43 }
44 file.flush();
45 file.close();
46 if (firstline.get()) {
47     nocyclist.write(fileArray[i].getAbsolutePath() +
48         "\n");
49     new File(groundtruthfilename).delete();
50 }
51 } catch (IOException e) {
52     e.printStackTrace();
53 } catch (ParseException e) {
54     try {
55         parseexceptions.write("Parse Exception in File "
56             + fileArray[i].getAbsolutePath() + "\n
57             ");
58     } catch (IOException e1) {
59         e1.printStackTrace();
60     }
61 }
62 }
63 }

```

Listing B.1: Der Java Code zum Parsen der Labeldateien des Tsinghua-Daimler-Datensatzes. Die JSON Dateien werden mit Hilfe der org.json.simple Bibliothek eingelesen. Nur Begrenzungsrahmen von Fahrradfahrern werden gespeichert. Die Begrenzungsrahmen werden in das von Darknet benötigte Format übertragen. Referenziert in Kapitel 4.2.1.

```

1 int getoverlapping(customBox person[], int pcount,
2                     customBox bikes[], int bcount, customBox
3                     cyclists[], image im) {
4
5     boxing mappings[maxCycles];
6     int mappingsCount = 0;
7     int cyclistCounter = 0;
8
9 //Alle Ueberschneidungen merken
10    for (int personindex = 0; personindex < pcount;
11          personindex++) {
12        customBox r1 = person[personindex];
13        for (int bikeindex = 0; bikeindex < bcount; ++
14            bikeindex) {
15            customBox r2 = bikes[bikeindex];
16
17            //Pruefen ob die sich ueberschneiden und ob die
18            //Person hoeher ist als das Fahrrad
19            if (!(r1.bleft + r1.width < r2.bleft || r1.btop
20                  + r1.height < r2.btop || r1.bleft > r2.
21                  bleft + r2.width
22                  || r1.btop > r2.btop + r2.height) && r1.
23                  bbot < r2.bbot) {
24                if (mappingsCount >= maxCycles) {
25                    return 500;
26                }
27                boxing map;
28                map.bikeindex = bikeindex;
29                map.personindex = personindex;
30                mappings[mappingsCount++] = map;
31            }
32        }
33    }
34
35 //Mehrfach ueberschneidungen finden und entfernen
36    for (int i = 0; i < mappingsCount; ++i) {
37        int actualperson = mappings[i].personindex;
38        if (actualperson == -1) {
39            continue;
40        }
41        mappings[i].personindex = -1;
42        int bikeoverlaps[maxCycles];
43        int overlaps = 0;
44        bikeoverlaps[overlaps++] = mappings[i].bikeindex;
45        mappings[i].bikeindex = -1;

```

```

34     for (int j = 0; j < mappingsCount; ++j) {
35         if (mappings[j].personindex == actualperson) {
36             next = 1;
37             bikeoverlaps[overlaps++] = mappings[j].
38                 bikeindex;
39             mappings[j].bikeindex = -1;
40             mappings[j].personindex = -1;
41         }
42     }
43     int actualbike = getBigestIntersectionIndex(person[
44         actualperson], bikes, bikeoverlaps,
45         overlaps);
46     int personoverlaps[maxCycles];
47     overlaps = 0;
48     personoverlaps[overlaps++] = actualperson;
49     for (int j = 0; j < mappingsCount; ++j) {
50         if (mappings[j].bikeindex == actualbike) {
51             next = 1;
52             personoverlaps[overlaps++] = mappings[j].
53                 personindex;
54             mappings[j].bikeindex = -1;
55             mappings[j].personindex = -1;
56         }
57     }
58     int theperson = getBigestIntersectionIndex(bikes[
59         actualbike], person, personoverlaps,
60         overlaps);
61     cyclists[cyclistCounter++] = makeCyclist(person[
62         theperson], bikes[actualbike], im.w, im.h)
63         ;
64     }
65     return cyclistCounter;
66 }
```

Listing B.2: Die Methode `getoverlapping` erhält alle Begrenzungsrahmen der auf einem Bild detektierten Fahrradfahrer und Personen. Die Begrenzungsrahmen werden per Überschneidung einander zugeordnet. Bei mehrfach Überschneidungen werden die Begrenzungsrahmen mit der höheren Überschneidung einander zugeordnet. Aus den zugeordneten Begrenzungsrahmen wird in der Methode `makeCyclist` eine großer Begrenzungsrahmen erstellt, der beide umschließt. Aus dem neuen Begrenzungsrahmen wird im Anschluss die Markierungsdatei erstellt. Referenziert in Kapitel 4.2.2.

```
1 ssh pacheco@10.149.71.3 sh train.sh
2 scp pacheco@10.149.71.3:~/laeufs C:\Temp\laeufs
3 set /p "x=<"C:\Temp\laeufs"
4 IF %x% EQU 0 (
5     C:\Users\awachtberger\Music\Kalimba.mp3
6     exit
7 )
8 scp pacheco@10.149.71.3:~/loss.txt C:\Users\awachtberger\
9     Desktop\Python\data\loss.txt
10 cd C:\Users\awachtberger\Desktop\Python\
11 python plot.py -all -i 1000
12 scp C:\Users\awachtberger\Desktop\Python\loss.txt
    pacheco@192.168.0.5:~/data/loss.txt
```

Listing B.3: Dieses Shell Skript wurde während des Trainings vom Windows Aufgabenplaner im 60 Minuten Takt aufgerufen. Auf dem Trainings-PC wird eine Shell-Skript ausgeführt. Das Musikstück Kalimba wird abgespielt, wenn das Training nicht mehr läuft. Andernfalls wird die Ausgabe des Trainings kopiert und mit einem Python Skript geplottet. Referenziert in Kapitel 4.3.2.

```

1 #!/bin/bash
2 # by Alexander Wachtberger
3 # Test if the process is running and giving information
4 # if so
5 wtf=$(ps -ef | grep [d]arknet | wc -l)
6 # if only 1 is only the ps call, musst be 2 while running
7 if [ $wtf = "1" ]
8 then
9     echo "TRAINING IS NOT RUNNING"
10    echo 0 > laeufts
11 else
12     echo "training is on"
13    echo 1 > laeufts
14 fi
15 nvidia-smi
16 ls -al /home/pacheco/darknet/backup | grep backup
17 ls -al /home/pacheco/ | grep loss.txt
18 ps -u pacheco -f | grep darknet
19 cat temp.txt | grep rate > loss.txt
20 exit

```

Listing B.4: Shell Skript zur Überprüfung des Trainings. Es wird geprüft, ob der Prozess läuft und entsprechend eine 0 oder 1 in die Datei laeufts geschrieben. Anschließend werden noch einige Ausgaben u.a. über GPU Nutzung aufgelistet. Zuletzt werden aus der kompletten Ausgabe vom Darknet Training, in temp.txt, die Zeilen mit den Fehlerwerten in die loss.txt geschrieben. Referenziert in Kapitel 4.3.2.

```

1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3 # Plot der Ausgabe vom Darknet Training
4 # by Alexander Wachtberger
5 import argparse
6 import sys
7 import matplotlib.pyplot as plt
8 import numpy as np
9 from pylab import *
10
11 def main(argv):
12     parser = argparse.ArgumentParser()
13     parser.add_argument(
14         "-i", dest='mod',
15         default=500, type=int,
16         help = "mean is calculated over i values"
17     )
18     parser.add_argument(
19         "-f", dest='file',
20         default="data\loss.txt",
21         help = "path to log file"
22     )
23     params = parser.parse_args()
24     f = open(params.file)
25     lines = [line.rstrip("\n") for line in f.readlines()]
26     numbers = {'1', '2', '3', '4', '5', '6', '7', '8', '9'}
27     iters = []
28     loss = []
29     avgloss = []
30     fig,ax = plt.subplots()
31     fig.suptitle("Fehler Zerfall Kurve")
32     prev_line = ""
33     avglossdict = {}
34     meanAvglossX = []
35     meanAvgloss = []
36     i=0
37     counter = 0.0
38     highest = 0.0
39     lowest = 2000.0
40     for line in lines:

```

```

41     args = line.split(' ')
42     if args[0][-1:]==':' and args[0][0] in numbers :
43         i += 1
44         iters.append(int(args[0][-1]))
45         loss.append(float(args[2]))
46         lowest = float(args[2]) if float(args[2])<
47                         lowest else lowest
48         #avg line
49         counter += float(args[2])
50         if i%params.mod == 0:
51             meanAvgloss.append(counter/i)
52             meanAvglossX.append(int(args[0][-1]))
53             highest= counter/i if counter/i>highest
54                 else highest
55             counter=0.0
56             i=0
57             meanAvgloss.append(counter/i)
58             meanAvglossX.append(int(args[0][-1]))
59             ticks = np.arange(0, 0.3, 0.01)
60             ax.set_yticks(ticks)
61             line1, = ax.plot(iters,loss)
62             line2, = ax.plot(meanAvglossX,meanAvgloss)
63             legend((line1, line2), ('Durchschnittliche
64                 Fehlerkurve', f"Ausgleichskurve des Fehlers"
65                 ))
66             plt.axis([0, iters[len(iters)-1]+100, 0, 0.3])
67             plt.xlabel('Anzahl der Iterationen')
68             plt.ylabel('Fehler in Prozent')
69             plt.grid()
70             plt.show()

71
72 if __name__ == "__main__":
73     main(sys.argv)

```

Listing B.5: Ein Python Skript, das eine Fehlerkurve plottet. Aus der Ausgabe des Trainings von Darknet werden die Zeilen mit den Angaben des aktuellen Fehlers gefiltert. Dann werden die Nummer der Iteration und der aktuelle durchschnittliche Fehler gespeichert. Zusätzlich wird der Fehler über 500 Werte gemittelt, um eine ruhige Kurve zu erhalten. Referenziert in Kapitel 4.3.2.

```

1 package mapparser;
2 import ...
3 ...
4 public class Mapparser {
5     private final static String filename = "C:\\\\Users\\\\
6         awachtberger\\\\Desktop\\\\Python\\\\data\\\\cyclist
7         .data";
8     static HashMap<String, Float> besteWerte;
9     private static HashMap<String, Integer> beste;
10    private static HashMap<Integer, LinkedList<String>>
11        fetteliste;
12    private static LinkedList<Result> results;
13
14    private static void init() {
15        results = new LinkedList<Result>();
16        beste = new HashMap<String, Integer>();
17        fetteliste = new HashMap<Integer, LinkedList<
18            String>>();
19        beste.put("detCount", 0);
20        beste.put("ap", 0);
21        beste.put("precision", 0);
22        beste.put("recall", 0);
23        beste.put("f1", 0);
24        beste.put("detCount", 0);
25        beste.put("tp", 0);
26        beste.put("fp", 0);
27        beste.put("fn", 0);
28        beste.put("f1", 0);
29        besteWerte = new HashMap<String, Float>();
30        besteWerte.put("detCount", (float) 0);
31        besteWerte.put("ap", (float) 0.0);
32        besteWerte.put("precision", (float) 0.0);
33        besteWerte.put("recall", (float) 0.0);
34        besteWerte.put("f1", (float) 0.0);
35        besteWerte.put("detCount", (float) 0);
36        besteWerte.put("tp", (float) 0);
37        besteWerte.put("fp", Float.MAX_VALUE);
38        besteWerte.put("fn", Float.MAX_VALUE);
39        besteWerte.put("f1", (float) 0.0);
40    }

```

```
38 public static void main(String[] args) {
39     init();
40     readResults();
41     System.out.println(String.format("      %6s  %10s
42                           %5s  %9s  %5s  %5s  %5s  %4s  %5s  %5s",
43                           "weight",
44                           "detections", "ap", "precision", "recall"
45                           , "F1", "TP", "FP", "FN", "IoU")
46                           );
47     results.stream().forEach(Result::print);
48     results.stream().forEach(Mapparser::beste);
49     createFetteListe();
50     besteAusgeben();
51     besteAlsHTMLAusgeben();
52 }
53
54 private static void readResults() {
55     FileReader fr = null;
56     try {
57         fr = new FileReader(filename);
58     } catch (FileNotFoundException e) {
59         e.printStackTrace();
60         return;
61     }
62     BufferedReader br = new BufferedReader(fr);
63     String zeile;
64     int i;
65     try {
66         while ((zeile = br.readLine()) != null) {
67             if (!zeile.contains("weights")) {
68                 continue;
69             }
70             Result result = new Result();
71             result.weight = zeile.substring(zeile.
72                     indexOf('_') + 1, zeile.indexOf(
73                         '.'));
74             zeile = br.readLine();
75             result.detCount = Integer.valueOf(zeile.
76                     substring(zeile.indexOf("=") +
77                         2, zeile.indexOf(",")));
78             while ((zeile = br.readLine()) != null) {
79                 if (!zeile.contains("class_id")) {
```

```

72             continue;
73     }
74     i = zeile.lastIndexOf("=");
75     result.ap = Float.valueOf(zeile.
76                             substring(i + 2, i + 7));
77     zeile = br.readLine();
78     i = zeile.lastIndexOf("precision =");
79     result.precision = Float.valueOf(
80                             zeile.substring(i + 12, i +
81                                           16));
82     i = zeile.lastIndexOf("recall");
83     result.recall = Float.valueOf(zeile.
84                                 substring(i + 9, i + 13));
85     i = zeile.lastIndexOf("F1-score");
86     result.f1 = Float.valueOf(zeile.
87                               substring(i + 11, i + 15));
88     zeile = br.readLine();
89     result.iou = Float.valueOf(zeile.
90                                substring(zeile.lastIndexOf(
91                                              "=") + 2, zeile.indexOf("%")
92                                              ));
93     Scanner scanner = new Scanner(zeile);
94     zeile = scanner.findInLine("TP =
95                               [0-9]*,");
96     result.tp = Integer.valueOf(zeile.
97                                 substring(zeile.indexOf("=")
98                                 + 2, zeile.indexOf(",,")));
99     zeile = scanner.findInLine("FP =
100                               [0-9]*,");
101    result.fp = Integer.valueOf(zeile.
102                                substring(zeile.indexOf("=")
103                                + 2, zeile.indexOf(",,")));
104    zeile = scanner.findInLine("FN =
105                               [0-9]*,");
106    result.fn = Integer.valueOf(zeile.
107                                substring(zeile.indexOf("=")
108                                + 2, zeile.indexOf(",,")));
109    results.add(result);
110    scanner.close();
111    break;
112 }
113 }
```

```

97     } catch (IOException e) {
98         e.printStackTrace();
99     } finally {
100        try {
101            br.close();
102        } catch (IOException e) {
103            e.printStackTrace();
104        }
105    }
106 }

107
108 private static void besteAlsHTMLAusgeben() {
109     System.out.println(String.format("<br/>      %6s
110                           %10s  %5s  %9s  %5s  %5s  %4s  %5s
111                           %5s<br/>", "weight",
112                           "detections", "ap", "precision", "recall"
113                           , "F1", "TP", "FP", "FN", "IoU")
114                           );
115     HashSet<Integer> indices = new HashSet<Integer>()
116         ;
117     indices.addAll(bestе.values());
118     for (Integer index : indices) {
119         System.out.println(String.format("%3d", index
120             ) + " "
121             + results.get(index - 1).toHTMLString
122             (fetteliste.get(index)) + "<
123             br/>");
124     }
125 }

126
127 private static void besteAusgeben() {
128     for (Entry<String, Integer> entry : bestе.
129         entrySet()) {
130         System.out
131             .println(entry.getKey() + ": " +
132                 bestе.get(entry.getKey()) +
133                 " " + bestеWerte.get(entry.
134                     getKey()));
135     }
136     System.out.println(String.format("\n      %6s  %10s
137                           %5s  %9s  %5s  %5s  %5s  %4s  %5s
138                           ", "weight",

```

```

125         "detections", "ap", "precision", "recall"
126             , "F1", "TP", "FP", "FN", "IoU")
127         );
128     HashSet<Integer> indices = new HashSet<Integer>()
129         ;
130     indices.addAll(beste.values());
131     for (Integer index : indices) {
132         results.get(index - 1).print();
133     }
134 }
135
136 private static void beste(Result r) {
137     if (besteWerte.get("detCount") < r.detCount) {
138         besteWerte.put("detCount", (float) r.detCount
139             );
140         beste.put("detCount", r.index);
141     }
142     if (besteWerte.get("ap") < r.ap) {
143         besteWerte.put("ap", (float) r.ap);
144         beste.put("ap", r.index);
145     }
146     if (besteWerte.get("precision") < r.precisionCalc
147         ) {
148         besteWerte.put("precision", (float) r.
149             precisionCalc);
150         beste.put("precision", r.index);
151     }
152     if (besteWerte.get("recall") < r.recallCalc) {
153         besteWerte.put("recall", (float) r.recallCalc
154             );
155         beste.put("recall", r.index);
156     }
157     if (besteWerte.get("f1") < r.f1Calc) {
158         besteWerte.put("f1", (float) r.f1Calc);
159         beste.put("f1", r.index);

```

```

160             beste.put("fp", r.index);
161         }
162         if (besteWerte.get("fn") > r.fn) {
163             besteWerte.put("fn", (float) r.fn);
164             beste.put("fn", r.index);
165         }
166     }
167
168     private static void addToFetteListe(Result r, String
169             string) {
170         LinkedList<String> liste = fetteliste.get(r.index
171             );
172         if (liste == null) {
173             liste = new LinkedList<String>();
174             fetteliste.put(r.index, liste);
175         }
176         liste.add(string);
177     }
178
179     private static void createFetteListe() {
180         for (Entry<String, Integer> entry : beste.
181             entrySet()) {
182             addToFetteListe(results.get(entry.getValue()
183                 - 1), entry.getKey());
184         }
185     }
186 }

```

Listing B.6: Das Java Programm ermittelt die besten Ergebnisse aus einer Datei. Die Datei enthält die Ausgaben der map-Funktion von AlexeysAB's Darknet Fork für mehrere Gewichte. Die Datei wird eingelesen und die Ergebnisse in einer Klasse Result gespeichert. Die Klasse Result enthält nur die Ergebnissdaten und zwei Methoden, um die Werte auszugeben. Nach dem Einlesen werden die besten Werte ermittelt. Das Programm kann noch stark optimiert werden. Referenziert in Kapitel 4.3.3.

```

1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3 #include "sensor_msgs/image_encodings.h"
4 #include "darknet.h"
5 #include <cv_bridge/cv_bridge.h>
6 /**
7  * @author Alexander Wachtberger
8 */
9 ros::Subscriber sub;
10 static int width;
11 static int height;
12 static image img;
13
14 void saveMetaInfo(const sensor_msgs::CameraInfoConstPtr&
15                   camInfo) {
16     width = camInfo.width;
17     height = camInfo.height;
18 }
19 void unsubscribe() {
20     ros::spinOnce();
21     sub.shutdown();
22 }
23 void listen() {
24     ros::spin();
25 }
26 void createImage(const sensor_msgs::ImageConstPtr&
27                   msgColor) {
28     cv_bridge::CvImagePtr cv_ptr;
29     try {
30         cv_ptr = cv_bridge::toCvCopy(msgColor,
31                                     sensor_msgs::image_encodings::BGR8);
32     } catch (cv_bridge::Exception& e) {
33         ROS_ERROR("cv_bridge exception: %s", e.what());
34         return;
35     }
36     img = mat_to_image(cv_ptr->image);
37     img.w = width;
38     img.h = height;
39 }
40 int initialize_camera() {
41     ros::init(0, NULL, "cyclist_detector");

```

```

40     ros::NodeHandle n;
41     sub = n.subscribe("camera/camera_info", 1,
42                         saveMetaInfo);
43     unsubscribe();
44     sub = n.subscribe("camera/image_color", 1, createImage
45                       );
46     pthread_t listen_thread;
47     pthread_create(&listen_thread, 0, listen, 0)
48     return 0;
49 }
50 }
```

Listing B.7: Diese C Klasse implementiert die Schnittstelle Image_Provider. Die Methode initialize_camera meldet sich beim ROS Master an, um die Meta-Informationen der Kamerabilder zu erhalten. Die Meta-Informationen werden zwischengespeichert. Anschließend wird eine Anmeldung für das Thema der Bilddaten durchgeführt. In einem neuen Thread werden die Bilddaten zum Abrufen mit der Methode get_image bereitgehalten. Referenziert in Kapitel 5.2.

```

1 #include <sstream>
2 #include "ros/ros.h"
3 #include "ros/rate.h"
4 #include "std_msgs/String.h"
5 #include "cyclist_detector.h"
6 /**
7  * This class is sending bounding boxes(x,y,width,height)
8   * relativ to the image size of the camera in a
9   * CSV list as String over the ROS system with
10  * topic name "CyclistDetectOnCAM1".
11  */
12 @author Alexander Wachtberger
13
14 ros::Publisher detection_publisher;
15
16 int initialize_bus() {
17     ros::init(0, NULL, "cyclist_detector");
18     ros::NodeHandle n;
19     detection_publisher = n.advertise< std_msgs::String >
20         ("CyclistDetectOnCAM1", 1);
21     return ros::ok();
22 }
23 void send_detections(box *detected, int count) {
24     std_msgs::String msg;
25     std::stringstream ss;
26     for (int i = 0; i < count; ++i) {
27         ss << detected[i].x << " " << detected[i].y << " "
28             << detected[i].w << " " << detected[i].h
29             << ";" ;
30     }
31     msg.data = ss.str();
32     if (ros::ok()) {
33         detection_publisher.publish(msg);
34     } else {
35         //TODO error handling
36     }
37 }

```

Listing B.8: Diese Klasse implementiert die Schnittstelle Publish_Detections. Bei der Initialisierung wird ein ROS Publisher beim ROS Master angemeldet. In der Methode send_detections werden über das zuvor angemeldete Thema Begrenzungsrahmendaten in einer CSV Liste als String versendet. Referenziert in Kapitel 5.2.

```

1 #include "darknet.h"
2 #include "cyclist_detector.h"
3 #include "publish_detections.h"
4 #include "image_provider.h"
5 /**
6  * @author Alexander Wachtberger
7  */
8 static char* cfgfile = "/home/nvidia/awachtberger/ffds/
9          ffds.cfg";
10 static char* weightfile = "/home/nvidia/awachtberger/ffds
11          /ffds.weights";
12 static network *net;
13 static image img;
14 static image img_raw;
15
16 void pass_detections(void *ptr) {
17     detections *dets;
18     dets = (detections*) ptr;
19     int i;
20     int detection_counter = 0;
21     box detected[dets->count];
22     for (i = 0; i < dets->count; ++i) {
23         if (dets->dets[i].prob[0] > 0.25) {
24             detection_counter++;
25             box b = dets->dets[i].bbox;
26             detected[detection_counter] = b;
27         }
28     }
29     send_detections(detected, detection_counter);
30     free_detections(dets->dets, dets->count);
31 }
32 void *detect_in_thread(void *ptr) {
33     float *X = img.data;
34     int nboxes = 0;
35     detection *dets = network_predict(net, X,&nboxes);;
36     detections detections;
37     detections.count = nboxes;
38     detections.dets = dets;
39     pthread_t send_thread;
40     if (pthread_create(&send_thread, 0, pass_detections, &
41                      detections))

```

```

39         error("send_thread creation failed");
40     return 0;
41 }
42 void *fetch_in_thread_ffds(void *ptr) {
43     free_image(img_raw);
44     img_raw = get_image();
45     letterbox_image_into(img_raw, net->w, net->h, img);
46     return 0;
47 }
48 int main(int argc, char **argv) {
49     pthread_t detect_thread;
50     pthread_t fetch_thread;
51     cuda_set_device(0);
52     net = load_network(cfgfile, weightfile, 0);
53     initialize_camera();
54     initialize_bus();
55     img = get_image();
56     while (42) {
57         if (pthread_create(&fetch_thread, 0,
58                           fetch_in_thread, 0))
59             error("Thread creation failed");
60         if (pthread_create(&detect_thread, 0,
61                           detect_in_thread, 0))
62             error("Thread creation failed");
63         demo_time = what_time_is_it_now();
64         pthread_join(fetch_thread, 0);
65         pthread_join(detect_thread, 0);
66     }
67 }

```

Listing B.9: Die Main Methode des Detektors ist in dieser Klasse enthalten. Das Netzwerk wird geladen, die Bus- und Kameranbindungen werden initialisiert, ein Bild wird vorgeladen und dann beginnt eine unendliche Schleife. Darin wird zum Einen das nächste Bild geladen und zum Anderen das aktuelle Bild vom Netzwerk berechnet. Nach der Berechnung werden die Ergebnisse der send_detections Methode übergeben. Referenziert in Kapitel 5.2.

```

1 void draw_and_save_wrong_images(int
2     checkpoint_detections_count, int
3     detections_count, int num_labels, int t, image *
4     val, box* truth, void nboxes, detection* dets,
5     float thresh_calc_avg_iou, char* path, int fp) {
6
7     int i, j;
8     //Draw labels and detection boxes
9     for (j = 0; j < num_labels; ++j) {
10         box tb = { truth[j].x, truth[j].y, truth[j].w,
11                     truth[j].h };
12         draw_bbox(val[t], tb, 2, 0.0, 1.0, 0.0);
13     }
14
15     for (i = 0; i < nboxes; ++i) {
16         if (dets[i].prob[0] > thresh_calc_avg_iou) {
17             draw_bbox(val[t], dets[i].bbox, 2, 1.0, 0.0,
18                         0.0);
19         } else if (dets[i].prob[0] > 0) {
20             draw_bbox(val[t], dets[i].bbox, 2, 0.0, 0.0,
21                         1.0);
22         }
23     }
24
25     //set correct path
26     int actual_detections = detections_count -
27         checkpoint_detections_count;
28
29     if (actual_detections < num_labels) {
30         find_replace(path, "/images/", "/false/
31                         falsenegative/", path);
32         find_replace(path, ".jpg", "_FN", path);
33         find_replace(path, ".png", "_FN", path);
34     } else if (actual_detections > num_labels && !fp) {
35         find_replace(path, "/images/", "/false/others/",
36                         path);
37         find_replace(path, ".jpg", "_false", path);
38         find_replace(path, ".png", "_false", path);
39     } else if (fp) {
40         find_replace(path, "/images/", "/false/
41                         falsepositives/", path);
42         find_replace(path, ".jpg", "_FP", path);
43         find_replace(path, ".png", "_FP", path);
44     } else {
45         return;
46     }
47 }

```

```
31     }
32     save_image(val[t], path);
33 }
```

Listing B.10: Die Methode draw_and_save_wrong_images malt erst alle Begrenzungsrahmen aus dem Array truth auf das aktuelle Bild. In truth sind die Begrenzungsrahmen aus der Markierungsdatei enthalten. Anschließend werden die detektierten Begrenzungsrahmen gemalt. Je nachdem, welcher Fehler auftritt, wird das Bild unter einem anderen Pfad und Namen gespeichert. Referenziert in Kapitel 6.2.1.

```

1 [net]
2 width=608
3 height=608
4 channels=3
5 batch=96
6 subdivisions=16
7 learning_rate=0.001
8 burn_in=1000
9 max_batches = 100010
10 policy=steps
11 steps=50000, 75000, 90000
12 scales=.5,.5,.1
13 momentum=0.9
14 decay=0.0005
15 angle=20
16 saturation = 1.5
17 exposure = 1.5
18 hue=.1
19
20 [convolutional]
21 batch_normalize=1
22 filters=32
23 size=3
24 stride=1
25 pad=1
26 activation=leaky
27 ...
28 [shortcut]
29 from=-3
30 activation=linear
31 ...
32 [upsample]
33 stride=2
34 ...
35 [route]
36 layers = -1, 36
37 ...
38 [yolo]
39 mask = 0,1,2
40 anchors = 12, 52, 21,103, 42, 63, 39,191, 80,110,
           66,307, 123,195, 186,311, 308,450

```

```
41 classes=1
42 num=9
43 jitter=.3
44 ignore_thresh = .5
45 truth_thresh = 1
46 random=1
```

Listing B.11: Ein Auszug aus der Konfigurationsdatei, welche YOLOv3 beschreibt. Alle von YOLOv3 verwendeten Schichten sind einmalig mit ihren Parametern aufgeführt. Referenziert in Kapitel 4.3.1.

Sonstiges

C.1 Inhalt der DVD

Hier befindet sich eine Auflistung der Dateien auf der beigefügten DVD. Jeweils mit einer kurzen Beschreibung dahinter in Klammern. Verzeichnisse haben „—“ vor ihrem Namen. Auf der DVD ist enthalten:

```
Root
|
| Inhaltsverzeichnis.txt (Dieses Inhaltsverzeichnis)
| Masterarbeit_Alexander_Wachtberger.pdf (Diese Masterarbeit als PDF Datei)
| —Dateien
|   | FFDS_loss_datei_zur_verwendung_mit_plot_py.txt (Fehlerwerte eines Trainings, kann mit „plot.py -f dateipfad“ und installiertem Python angezeigt werden)
|   | genauigkeitstest_auswertung.txt (Ergebnis der Berechnung der Genauigkeit für beide Netzwerke und unterschiedliche Auflösungen)
|   | geschwindigkeitstest_auswertung.xlsx (Ergebnisse der Geschwindigkeitstests für beide Netzwerke)
|   | map_ausgabe_von_vielen_gewichten.txt (Ergebnis eines Trainings)
|   | —Code,Skripts,Snippets (Dateien zum Aufsetzen eines Training, zum Checken ob ein Training läuft, zur Nachbearbeitung eines Training, zur Datensatzherstellung und -säuberung)
|   | —Videos (Enthält die Videos die zur Generierung des eigenen Datensatzes verwendet wurden)
|   | —Geschwindigkeitstest (Enthält die Dateien die für die Geschwindigkeitstest verwendet wurden und eine Anleitung zur Durchführung)
|   | —Repos (Enthält die gezippten Dateien von drei Repository: das FFDS; das modifizierten Darknet, zum trainieren und zur Datensatz Erstellung; das modifizierten Darknet, um es als Bibliothek zu für das FFDS zu verwenden)
```

C.2 Installationsanleitung für das FFDS

Diese Anleitung dient der Inbetriebnahme des im Rahmen dieser Arbeit entwickelten Fahrradfahrer Detektionssystems. Die Leser, die nur schnell etwas sehen wollen oder die Voraussetzungen nicht erfüllen, führen Schritt 5 durch und springen direkt zu Punkt 16. Diese Anleitung wurde auf einem Laptop mit einer GeForce GTX 1050 TI Grafikkarte, Ubuntu 18.04 als Betriebssystem, ROS Melodic, OpenCV 3.3.1 und CUDA 10.0 durchgeführt.

1. Das Betriebssystem, CUDA, ROS Melodic, Buildtools für C und C++ und OpenCV müssen installiert sein.
2. Repository klonen von (1,9GB, davon 1,6GB .git Ordner und 300MB Gewichte):
<https://gitlab.beuth-hochschule.de/master-ffds/darknet.git> oder mit SSH:
git@gitlab.beuth-hochschule.de:master-ffds/darknet.git
3. In der Datei `Makefile` für die Variable `ARCH` die Compute Capability der verwendeten GPU einstellen. Siehe <https://developer.nvidia.com/cuda-gpus>.
4. In der Kommandozeile den Befehl `make` ausführen.
5. Repository klonen von (500MB):
<https://gitlab.beuth-hochschule.de/s72834/ffds.git> oder mit SSH:
git@gitlab.beuth-hochschule.de:s72834/ffds.git
6. Die `libdarknet.so` aus dem `darknet` Verzeichnis in den `ffds` Ordner kopieren.
7. In der Datei `ffds/src/ffds_publisher/CMakeLists.txt` die 10. Zeile, die mit `set(FFDS_PATH` beginnt, den Pfad zum `ffds` Ordner anpassen.
8. In der Datei `ffds/src/ffds_publisher/ffds.cpp` die 6. und 7. Zeilen der Pfade anpassen:

```
char* cfgfile = "/absolut/path/to/ffds/ffds.cfg";
char* weightfile = "/absolut/path/to/ffds/ffds.weights";
```
9. In den Ordner `ffds` wechseln
10. In der Kommandozeile den Befehl `catkin_make` aufrufen

11. Bei erfolg das Programm in der Kommandozeile mit dem Befehl `./devel/lib/ffds_publisher/ffds` starten.
12. Das Netzwerk wird geladen und die einzelnen Schichten werden dabei aufgelistet.
13. Falls OpenCV eine Fehlermeldung ausgibt und das Programm weiterläuft, kann die Fehlermeldung ignoriert werden. Andernfalls kann es sein, dass der Treiber der Kamera nicht installiert ist, die Kamera nicht erkannt wird oder die Firmware der Kamera aktualisiert werden muss.
14. Falls der ROS Master nicht gestartet ist, erfolgt eine Fehlermeldung und die Begrenzungsrahmen der Detektionen werden nicht veröffentlicht.
15. Die Anzahl der erreichten FPS ist zu sehen. Wenn der ROS Master gestartet war, werden jetzt über das Thema „CyclistDetectOnCAM1“ die Begrenzungsrahmen detekтирter Fahrradfahrer übermittelt.
16. Damit ein Fenster mit Kamerabildern und darauf detektirten Objekten zu sehen ist, wie folgt vorgehen:
 - a) Der Anweisung auf <https://pjreddie.com/darknet/install/> zum installieren von Darknet folgen. Das heißt, wieder Repository klonen, Makefile einstellen und make aufrufen.
 - b) Mit folgendem Aufruf wird ein Fenster geöffnet und gezeigt, was die Kamera sieht:

```
./darknet detector demo cfg/coco.data /path/to/ffds.cfg /path/to/ffds.weights
```

/path/to muss jeweils mit dem richtigen Pfad zum geklonten FFDS Repository aus Schritt 5 ausgetauscht werden. Wenn keine GPU Unterstützung verwendet wird, sollte tiny.cfg und tiny.weights verwendet werden.
 - c) Mit einem Handy auf Google nach Fahrradfahrer Bildern suchen und vor die Kamera halten.
 - d) Durch die falsche Datensatzdatei (cfg/coco.data) werden Fahrradfahrer als lauter lustige Sachen bezeichnet.
17. Viel Spaß!

