

Master-Thesis

Institut für Informatik
Fachbereich Informatik und Mathematik

Video- und sensorgestütztes Autonomes Fahren für ein selbstbalancierendes Fahrrad

Yasin Giray
Matrikelnummer: 6490672

Prüfer/-in:
Prof. Dr. L. Hedrich

Betreuer:
M. Sc. Sascha Schmalhofer

Abgabedatum:
10. Oktober 2024

Frankfurt am Main, 10. Oktober 2024

Bitte dieses Formular zusammen mit der Abschlussarbeit abgeben!

Erklärung zur Abschlussarbeit

**gemäß § 34, Abs. 16 der Ordnung für den Masterstudiengang Informatik
vom 17. Juni 2019**

Hiermit erkläre ich

(Nachname, Vorname)

Die vorliegende Arbeit habe ich selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst.

Ebenso bestätige ich, dass diese Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

Zudem versichere ich, dass die von mir eingereichten schriftlichen gebundenen Versionen meiner Masterarbeit mit der eingereichten elektronischen Version meiner Masterarbeit übereinstimmen.

Frankfurt am Main, den

Unterschrift der/des Studierenden

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Alle Stellen und Textpassagen, die wörtlich oder sinngemäß aus veröffentlichten Quellen oder anderen fremden Texten entnommen sind, sind als solche kenntlich gemacht. Ebenso wurden alle Abbildungen, sofern nicht selbst von mir erstellt, mit entsprechenden Quellennachweisen versehen.

Diese Arbeit wurde noch nicht, auch nicht auszugsweise, für eine andere Studien- oder Prüfungsleistung verwendet.

Datum

Unterschrift

Inhaltsverzeichnis

1 Einleitung	1
1.1 Problemstellung	2
1.2 State of the Art	2
1.3 Grundlagen	3
1.3.1 Kamerasensoren	3
1.3.2 Machine Learning	3
1.3.2.1 Convolutional Neural Network	4
1.3.2.2 Convolutional Layer	4
1.3.2.3 Convolution Kernel	4
1.3.2.4 Pooling Layer	5
1.3.2.5 Pooling Kernel	5
1.3.2.6 Aufbau einer Machine Learning Pipeline	6
1.3.3 Softwarestack	6
1.3.3.1 Linux for ARM	7
1.3.3.2 Python3	7
1.3.4 Aktueller Stand des Forschungsprojekts	7
2 Konzeptionierung und Realisierung	10
2.1 Übersicht	10
2.1.1 Mobilität	11
2.1.2 Echtzeitkritisches System	11
2.1.3 Formfaktor und Leistungsaufnahme	13
2.1.4 Framework	13
2.2 Theoretisches Konzept	15
2.2.1 Hauptplatine	15
2.2.1.1 NVIDIA Jetson Nano	16
2.2.2 Kamerasensor	16
2.2.2.1 Auflösung	16
2.2.2.2 Seitenverhältnis und Sichtfeld	17
2.2.2.3 Bildwiederholrate	20
2.2.2.4 Verschluss	20

2.2.3	Objekterkennung mittels KI	23
2.2.3.1	Wahl des Maschinenmodells	24
2.2.3.2	Trainingsdaten	26
2.2.4	Steuer- und Reglereinheit	28
2.2.4.1	PID-Regelkreis	29
2.2.4.2	Model Predictive Controller	30
2.3	Technische Realisierung	35
2.3.1	Einrichten des Jetson Nano	35
2.3.2	Trainingsprozess und Datensatz	38
2.3.3	Aufbau der Inferenzpipeline	40
3	Ergebnisse und Diskussion	48
3.1	Inferenzpipeline mit YOLO	48
3.2	MPC-Regler	51
3.3	Limitationen	52
3.4	Zukünftige Arbeit	52
3.5	Abstract	53
Quellenverzeichnis		53
Abkürzungsverzeichnis		57

Abbildungsverzeichnis

1.1	Links: Fehlerrate relativ zur Größe des Datensatzes. Rechts: Differenz des Rechenaufwands relativ zu unterschiedlichen Kernelgrößen im Pooling- und Convolutional Layer	5
1.2	Illustration der Ausrichtungssachsen anhand eines Flugzeugs.	8
1.3	Schräglage eines Motorrads in einer Rechtskurve. \vec{F}_{Zf} ist dabei die Zentrifugalkraft, die das Motorrad aus der Kurve zieht.	9
2.1	Darstellung des toten Winkels und Berechnung der minimalen Objektdistanz .	17
2.2	Geometrische Darstellung der Horizontberechnung	18
2.3	Illustration eines Runaways und der entsprechenden Tangente	19
2.4	Illustration des Rolling Shutter- Effekts	20
2.5	Visualisierung des technischen Unterschieds eines Rolling Shutters und Global Shutters	21
2.6	Vergleich von Box-Prompt und Point-Prompt im FastSAM Framework	24
2.7	Segmentierung mit YOLO[1]	25
2.8	Segmentierung mit FastSAM[2]	25
2.9	Darstellung der mean Average Precision und Latenz der unterschiedlichen Modellgrößen	25
2.10	Ordnerhierarchie des YOLO-Datensatzes	27
2.11	Illustration der zeitdiskreten Verhältnisse der gemessenen, vorhergesagten und idealen Kurven eines MPC-Reglers	30
2.12	Vogelperspektive der planaren Straßenebene	31
2.13	Mögliche Schwerpunkte des Einspurmodells	32
2.14	Darstellung der geometrischen Komposition und des Momentanpols	33
2.15	Der zusammengebaute Jetson Nano	36
2.16	Beispiel für ein annotiertes Sample mit Roboflow[1]	38
2.17	Generierung des Datensatzes mit einem 80:10:10 Data Split	39
2.18	Die Verlustfunktionen des YOLOv8s-seg-Modells	40
2.19	Eine ungefilterte Inferenz[3]	41
2.20	Eine gefilterte Ausgabe der Inferenz[3]	42
2.21	Die aggregierten Bounding Boxen des Trainingsdatensatzes	43
2.22	Ein mit Kompression verzerrtes Bild[3]	44
2.23	Ein mit Aufblähen verzerrtes Bild[3]	44
2.24	Ein Beispiel der eingezeichneten Stützpunkte und -linien	45

2.25 Eine iterative Simulation des MPC-Reglers	46
2.26 Die Originalgröße des Graphen	46
2.27 Illustration der Einlenkgröße, die durch den MPC-Regler bestimmt wurde	46
3.1 Illustration des Konfidenzverlaufs des Objekts <i>street_main</i>	49
3.2 Der Konfidenzverlauf mit einer aufgeblähten Kameraperspektive	49
3.3 Ein möglicher Zuschnitt für den Preprocess[3]	50
3.4 Illustration der zeitdiskreten Verhältnisse der gemessenen, vorhergesagten und idealen Kurven eines MPC-Reglers	51

Tabellenverzeichnis

2.1 Gekürzte Übersicht der relevanten Inferenzparameter	26
2.2 Liste aller relevanten Git-Befehle[4]	37
2.3 Die verwendeten Parameter für das YOLO-Modell	41

1 Einleitung

Durch die erstmalige Adaption von Fotolithografie im Halbleiterbau 1955 und der darauffolgende technische Fortschritt, vor allem der sogenannten *Extreme Ultra Violet Lithografie*, gab es einen explosionsartigen Anstieg in der Rechenleistung und Effizienz von integrierten Schaltungen. Es ist mittlerweile möglich, mit Graphics Processing Units (kurz GPU) große Datensätze in Echtzeit auszuwerten. Dies hat eine Renaissance in der Datenverarbeitung ausgelöst, welche bereits bekannte Methoden wie Machine Learning, Deep Learning und Computer Vision modernisiert hat. Mit der Einführung von Mixed Precision Compute und dem Hinzufügen von Instruktionen wie Fused Multiply-Add (kurz FMA) in moderne Grafikkartenarchitekturen wurde das Trainieren oder auch das Berechnen von Inferenzen von Maschinenmodellen um ein Vielfaches beschleunigt.[5]

Diese äußereren Faktoren, die Beschleunigung der Rechenleistung und der enorme Fortschritt in der Künstlichen Intelligenz, bilden die Grundlage der Problemstellung dieser Masterarbeit, Autonomes Fahren. Durch EUV-Lithografie ist es heute möglich, Miniatur-System on Chips in Fahrzeugen unterzubringen. Mit der gestiegenen Effizienz und Rechenleistung von Grafikkartenarchitekturen ist es außerdem möglich, Datenströme von Kamerasensoren in Echtzeit zu inferieren. Hieraus kristallisieren sich diverse Probleme und Fragestellungen:

- Was ist die untere Schranke der Rechenleistung für ein echtzeitkritisches System
- Wie akkurat wird mittels Maschinellem Lernen ein Objekt erkannt?
- Ist es möglich ein Fahrzeug durch KI autonom fahren zu lassen?
- Ist es möglich Straßen mittels Computer Vision zu segmentieren?
- Wie akkurat kann ein KI-Modell andere Teilnehmer im Verkehr erkennen?
- Ist ein Computer Vision-Ansatz mit Sensor Fusion kompatibel?
- Wird Sensor Fusion durch Computer Vision obsolet?
- Wie verhält sich ein Computer Vision-Ansatz bei einem instabilen System wie einem Fahrrad?

Während ein Teil dieser Fragen bereits von marktführenden Unternehmen wie Tesla oder Honda adressiert werden, bleiben dennoch viele dieser Antworten unter Verschluss oder unbeantwortet.

1.1 Problemstellung

Im Vordergrund dieser Masterarbeit steht die Integration einer Computer Vision-Pipeline auf einem System on Chip für das fortlaufende Forschungsprojekt von Prof. Dr. Lars Hedrich, eines selbst-balancierenden Fahrrads. Das Fahrrad ist bereits Teil von mehreren Bachelor- und Masterarbeiten gewesen. Deshalb besitzt das Fahrrad bereits diverse Steuer-, Regler und Sensorelemente. Ein grundlegender Baustein fehlt jedoch. Dieser ist die visuelle Orientierung, eine Vorgabe der Fahrtrichtung und die Lenkersteuerung während der Fahrt. Ein großes Problem im Computer Vision-Bereich ist die Erkennung und Klassifizierung von Objekten innerhalb eines Bilds. Hier kommen oftmals KI-Modelle zum Einsatz, welche mit einem spezialisierten Datensatz trainiert werden, um Prädiktionsaufgaben zu übernehmen. Ein Ziel dieser Masterarbeit ist die Integration eines solchen Maschinenmodells in eine Computer Vision-Pipeline.

Außerdem soll geprüft werden, ob es möglich ist, diese Pipeline in ein eingebettetes System zu integrieren. Die Mobilität und Leistungsfähigkeit spielen dabei eine besondere Rolle. Hier soll herausgefunden werden, wie stark diese Restriktionen die Echtzeitfähigkeit der Pipeline einschränken.

Des Weiteren stellt sich die Frage der Bedienung und Steuerungslogik des Fahrrads. Das Fahrrad kann bereits mithilfe eines Servomotors den Einlenkwinkel einstellen. Das reicht jedoch nicht, um ein Fahrrad autonom fahren zu lassen. Hieraus entspringen mehrere Probleme. Zum einen muss eine Form der Straßenerkennung bereitgestellt werden, die es ermöglicht, eine Trajektorie zu berechnen. Zum anderen wird ein Regelkreis benötigt, der es ermöglicht, die berechnete Trajektorie entlangzufahren. Die Regelungstechnik ist ein breites Feld und bietet unterschiedliche Lösungen für unterschiedliche Probleme an. Ein weiteres Ziel ist, einen geeigneten Regelkreis für die Steuerung des Fahrrads zu finden.

1.2 State of the Art

Der wahrscheinlich bekannteste und erfolgreichste Hersteller von autonomen Fahrzeugen ist Tesla. In früheren Iterationen ihrer Fahrzeuge hat Tesla auf die Sensor Fusion gesetzt. Die Sensor Fusion bezeichnet das Zusammenspiel mehrerer Sensortypen, die eine Aufgabe bewältigen sollen. Hierbei wurden Ultraschallsensoren und Videokameras mit differenzierten Spezifikationen ausgewählt. Seit 2022 hat sich Tesla jedoch von diesem Konzept distanziert und fokussiert sich auf Tesla Vision. Die Idee dahinter sei, dass das Bedienen eines Fahrzeugs von einem Menschen durchgeführt werden könne und ein Mensch in der Lage sei, dies nur mit seinen eigenen zwei Augen zu bewerkstelligen.[6][7]

Honda hat 2017 den Riding Assist vorgestellt. Dabei handelt es sich um ein Motorrad, das sich selbst balancieren kann. Zwei elektrische Motoren sind innerhalb des Rahmens mit

zwei Schwingarmen verbaut, die mit der Sensormessung eines Gyroskops das Motorrad ausbalancieren.[8]

Im Machine Learning-Bereich ist NVIDIA der Marktführer. NVIDIA bietet seit 2014 unterschiedliche SoC für mobile Machine Learning-Plattformen an. Der Jetson Nano ist einer dieser SoC.[9] Diese Geräte sind mit Grafikeinheiten bestückt und sollen es ermöglichen, auf Low Power-Plattformen diverse Machine Learning Aufgaben zu bewältigen. In Kombination mit ausgereiften Maschinenmodellen ist es möglich, rechenintensive Inferenzen auf mobilen Plattformen durchzuführen. Im Computer Vision-Bereich sind nach GitHub-Sternen ULTRALYTICS[10] und SAM bzw. FastSAM[2] sehr beliebte Repositorys. In beiden Fällen handelt es sich um einen Softwarestack, der um die respektiven Maschinenmodelle aufgebaut wurde. Beide KI-Modelle sind auf Computer Vision spezialisiert und sind über große Datensätze, wie COCO2017[11] vorgenommen worden. ULTRALYTICS befindet sich weiterhin in aktiver Entwicklung und ist mit vielen Plattformen kompatibel.

1.3 Grundlagen

Im Grundlagenkapitel werden Aspekte, Informationen und Begrifflichkeiten erklärt, die im späteren Verlauf zum Verständnis der gesamten Masterarbeit notwendig sind. Dabei werden die Themen nicht vollständig ausgeführt, aber sollten dennoch eine gute Grundlage bieten.

1.3.1 Kameratasoren

Weil die Kameraunterstützung für das Autonome Fahren in dieser Masterarbeit im Vordergrund steht, ist es wichtig, sich mit den grundlegenden Technologien vertraut zu machen. Auch wenn im Abschnitt 2.2.2 einige Grundlagen und Zusammenhänge erklärt werden, ist es von Vorteil, sich mit der Datenstruktur von Bildern und Videos, mit Kamerataspezifikationen und allgemeinen optischen Problemen, wie dem Fluchtpunkt, vertraut zu machen.

1.3.2 Machine Learning

Im Zuge dieser Masterarbeit ist es notwendig, ein grundlegendes Verständnis zum Aufbau von neuronalen Netzen und insbesondere von Convolutional Neural Networks, von Annotations- und Trainingsprozessen zu besitzen. Hinzu kommen Themen wie Hyperparameter, Convolutional- und Pooling Layer, Kernel, der Inferenzprozess und die Konstruktion einer Machine Learning-Pipeline zu haben. Um Unklarheiten zu vermeiden und einen gemeinsamen Konsens zu schaffen, folgt hier eine kurze Begriffserläuterung.

1.3.2.1 Convolutional Neural Network

Wenn neuronale Netze generalisiert werden, werden sie klassischerweise in drei Layer kategorisiert. Dabei ist das erste der Input Layer, das zweite der Hidden Layer und das dritte der Output Layer. Die Eingabe bei der Bildverarbeitung besteht standardmäßig aus Pixeln mit Graustufen oder Farbkanälen in einem zweidimensionalen Array. Der Hidden Layer eines ordinären CNNs besteht in der Regel aus aufeinanderfolgenden Convolutional- und Pooling Layern. Diese können beliebig oft aneinander konkateniert werden.

1.3.2.2 Convolutional Layer

Der Input Layer ist direkt mit dem ersten Convolutional Layer verbunden. Der Zweck eines Convolutional Layers ist es, eine Feature Map zu erstellen. Dabei wird jedes Pixel auf seine Eigenschaften geprüft. Im Fall von Multichannel Inputs kann für jeden Farbkanal ein paralleles Convolutional Layer existieren. Um die Features aus einem Bild zu extrahieren, wird ein sogenanntes Convolution Kernel verwendet.

1.3.2.3 Convolution Kernel

Hinter dem Begriff Convolution Kernel steckt sowohl eine Rechenoperation als auch ein Operand. Bei der Operation handelt es sich um eine Faltung bzw. ein Skalarprodukt zweier Matrizen, das es ermöglicht, gewichtete Mittel zu berechnen. Anschaulich heißt das im Bereich des Maschinellen Lernens und der Bildverarbeitung, dass mittels einer Faltungsmatrix Korrelationen und Gewichte zwischen benachbarten Pixeln ermittelt werden können. Eine Faltungsmatrix ist dabei eine quadratische Matrix ungerader Dimension. Diese Eigenschaften sind erforderlich, um einen Mittelpunkt in der Faltungsmatrix zu erzeugen. Faltungsmatrizen können beispielsweise im Falle einer 3x3 Schärfungsmatrix wie folgt aussehen[12]:

$$\text{Schärfungsmatrix} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad (1.1)$$

Die Schärfungsmatrix spielt besonders im Hervorheben von Eigenschaften wie Kanten eine Rolle. Für die Objekterkennung ist diese unerlässlich, weil die aggregierten Operationen die Feature Map bilden. Diese wird im Pooling Layer weiterverarbeitet.

1.3.2.4 Pooling Layer

Das Pooling Layer erhält als Eingabe, die vom Convolutional Layer erstellten, Feature Maps. Die Hauptaufgabe im Pooling Layer ist das Downsampling. Mit Downsampling ist das Verkleinern der Dimensionen des Bildes gemeint. Bilder sind große Datenstrukturen. Ein reguläres Multichannel Farbbild mit einer Auflösung von 1920x1080 hat 2.073600 Pixel, mit je 3 Features pro Pixel und einer Farbtiefe von 8 Bits pro Feature. Für ein CNN mit einem Kernel würde das bedeuten, dass 6.220.800 Berechnungen von 8 Bit-Zahlen pro Convolutional Layer stattfinden müssen. Um diesem Rechenaufwand entgegenzuwirken und Overfitting zu vermeiden, wird hier ebenfalls ein Kernel verwendet.

1.3.2.5 Pooling Kernel

Das Pooling Kernel besteht typischerweise aus einer kleinen, quadratischen Matrix. Nagi et al.[13] haben gezeigt, dass bei großen Matrizengrößen der Informationsverlust ansteigt und das resultierende Maschinenmodell fehleranfälliger wird.

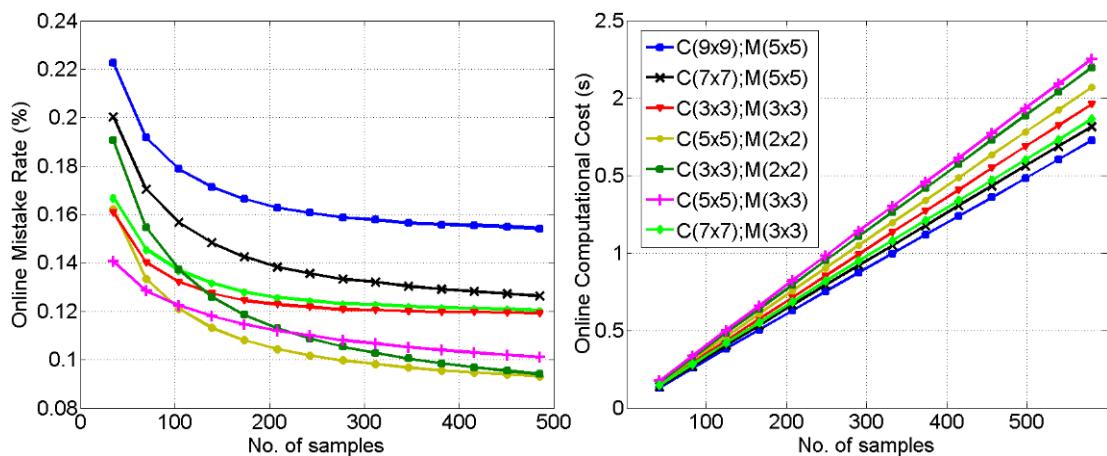


Abb. 1.1: Links: Fehlerrate relativ zur Größe des Datensatzes. Rechts: Differenz des Rechenaufwands relativ zu unterschiedlichen Kernelgrößen im Pooling- und Convolutional Layer.

Quelle: <https://ieeexplore.ieee.org/document/6907338>, zuletzt besucht 21.07.2024

Das Pooling Kernel wird entweder mit einem Max Pooling oder Average Pooling implementiert. Beim Pooling wird ein Bildausschnitt der Größe des Pooling Kernels entnommen. Im Fall einer 2x2 Matrix werden 4 Features, in diesem Fall Pixel, miteinander verglichen. Während das Max Pooling den lokalen maximalen Wert der vier Features bestimmt, wird beim Average Pooling das arithmetische Mittel berechnet. Anschließend werden diese vier Features auf den ermittelten Wert reduziert. Dadurch ist eine Reduktion in diesem Fallbeispiel von 75 % zu beobachten.

1.3.2.6 Aufbau einer Machine Learning Pipeline

Um mit Maschinenmodellen qualitative Ergebnisse zu erzielen, ist es wichtig, die Grundlagen des Entwicklungsprozesses zu verstehen. Vier wichtige Grundbausteine sind dabei das Annotieren von Datensätzen, das Trainieren eines geeigneten Maschinenmodells, das Exportieren eines Maschinenmodells und das Deployment bzw. die Integration in ein System. Hier eignet sich eine ganzheitliche Herangehensweise und Orientierung an Industriestandards, weil diese Schritte sich nur begrenzt separieren lassen. Es existieren beispielsweise bereits Dataset-Formate wie Common Objects in Context[11] (kurz COCO), die es erleichtern, eigene Datasets zu erstellen und zu annotieren. Hierbei können beispielsweise Open Source-Annotationstools für COCO-Datasets verwendet werden, wenn ein entsprechendes Maschinenmodell damit kompatibel ist. Des Weiteren wird der COCO-Datensatz ebenfalls von Entwicklern genutzt, um Maschinenmodelle mittels Benchmark miteinander zu vergleichen oder auch als Trainingsdatensatz verwendet, um eine Form des Grundwissens zu gewährleisten.

Export und Deployment sind hauptsächlich abhängig von Performance- und Kompatibilitätsanforderungen. Je nach Implementierung von Maschinenmodellen können sich hier große Unterschiede durch den Export manifestieren. Laufzeitumgebungen, oder auch Runtime Environments genannt, für Maschinenmodelle sind eine Low Level-Optimierungsinstanz. Je nach verwendeter Hardware und Softwarestack, können diese Laufzeitumgebungen den Trainings- und Inferenzprozess um Magnituden beschleunigen. Beispiele dafür sind beispielsweise Open Neural Network Exchange[14], OpenVINO[15] und TensorRT.[16] Besonders interessant im Bereich der eingebetteten Systeme ist TensorRT. NVIDIA bietet vollständige System on Chip (kurz SoC) an, die für mobile Inferenzaufgaben geeignet sind.

Wenn ein gegebenes System einen gesicherten Internetzugang besitzt, ist es ebenfalls möglich, eine Continous Integration/Continous Deployment (kurz CI/CD)- Pipeline mit einem Git[17]- Derivat zu erstellen. Der Vorteil hierbei ist, dass es möglich ist, Systeme inkrementell zu verbessern, neue Features zu integrieren oder auch die Inferenz-Engine seamless auszutauschen.

1.3.3 Softwarestack

Da es sich bei den gängigsten und erhältlichen SoCs um ARM-basierte Prozessoren handelt, ist es wichtig, sich mit dem Softwarestack auseinanderzusetzen. ARM ist mit x86 inkompatibel, was sich im Entwicklungsprozess bemerkbar macht. Ein Aspekt davon ist das Betriebssystem.

1.3.3.1 Linux for ARM

Das Zielsystem ist ein SoC. Diese sind oftmals mit ARM-basierten Prozessoren ausgestattet. Die Unterschiede im Instruktionssatz zwischen ARM-basierten und x86-basierten Prozessoren ist enorm. Deshalb kommt es oft vor, dass Programme und Libraries, die auf x86-basierten Systemen ohne Probleme laufen, zu Kompatibilitätsproblemen auf ARM-basierten Systemen führen. Da herkömmliche x86-Betriebssysteme nicht verwendet werden können, bietet sich hier Linux for ARM an. Das hat jedoch zur Folge, dass es Diskrepanzen in der Art und Weise der Implementierung gibt. Die Unterschiede zwischen dem Entwicklungssystem und dem SoC-System liegen hauptsächlich in den Dependencies, die nicht erfüllt werden können, und den Compilern, die zum Teil auf ARM nicht verfügbar sind. Konkret betreffen diese Limitationen Python3, PyTorch, Clang Compiler, torchvision, LibTorch, MATLAB Runtime Environment und JetPack. Bei der Entwicklung ist deshalb regelmäßig darauf zu achten, ob der benötigte Softwarestack auf dem Zielsystem verfügbar ist. In späteren Kapiteln wird diese Thematik genauer beleuchtet.

1.3.3.2 Python3

Als systemübergreifende Entwicklungsumgebung eignet sich Python3 besonders gut. Der Grund dafür ist, dass Python3 mit einem Interpreter arbeitet. Das erleichtert die Entwicklung, weil nur eine Code Base benötigt wird. Für compilerbasierte Programmiersprachen müsste man sowohl für x86 als auch ARM entsprechend die Code Base compilieren und unter gewissen Voraussetzungen anpassen. Des Weiteren bietet Python3 den Vorteil, dass es sich im Machine Learning-, Computer Vision- und Robotics-Bereich erfolgreich durchgesetzt hat. Dadurch existieren für diese Bereiche viele Libraries, die sich Python Wrapper und die Performance anderer Programmiersprachen wie C++ zunutze machen. Mittels Python Wrapper lassen sich rechenintensive Aufgaben und Funktionen in bereits compilierten C, C# und C++-Maschinencode umlagern. Diese können dann im Python-Skript eingebunden werden. Ein beliebtes Beispiel hierfür ist die Python OpenCV Library.[18]

1.3.4 Aktueller Stand des Forschungsprojekts

Das Forschungsprojekt besteht seit mehreren Jahren und hat bereits erfolgreich integrierte Funktionalitäten. Die letzte Arbeit, die das Forschungsprojekt fortgesetzt hat, ist von Alex Hunziker aus dem Jahr 2019.[19] Das selbstbalancierende Fahrrad benutzt als zentralen Baustein einen BeagleBoneBlack. Dieser ist ein auf ARM basierter SoC. Das Fahrrad befindet sich bereits in einem Stadium, wo es selbstständig beschleunigt und seine Geschwindigkeit halten kann. Hierfür wird ein Radnabenmotor verwendet. Diese sind in den meisten Fällen und auch in diesem Fall bürstenlose Motoren. Sie lassen sich mittels

eines Brushless Direct Current-Controllers ansteuern. Hierfür werden zwei 36 V Li-Ion-Batterien verwendet. Sowohl der BLDC-Controller als auch die Li-Ion-Batterien sind für Hochstrom-Anwendungen zertifiziert. Der Grund dafür ist, dass im Stillstand des Fahrrads die Trägheit der Masse am höchsten ist. Somit können Stromspitzen von 20 bis 30 A auftreten. Außerdem kann das Fahrrad für kurze Strecken bereits die Balance halten.

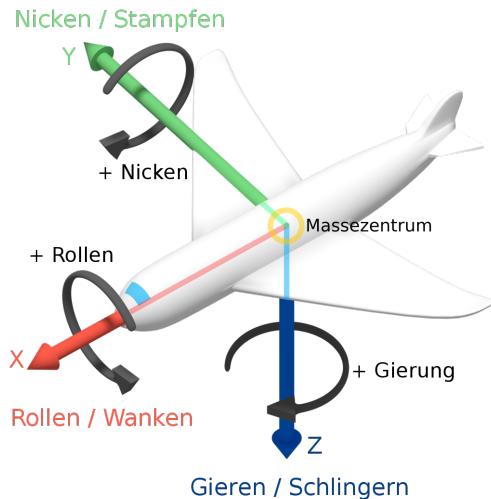


Abb. 1.2: Illustration der Ausrichtungssachsen anhand eines Flugzeugs.

Quelle: https://commons.wikimedia.org/wiki/File:Roll_pitch_yaw_gravitation_center_de.png, zuletzt besucht 01.09.2024

Hierfür wird ein Gyroskop und ein Tariergewicht orthogonal zur Ausrichtung des Fahrrads platziert. Im dreidimensionalen Raum existieren die Freiheitsgrade Roll (dt. Rollen), Pitch (dt. Nicken) und Yaw (dt. Gieren), welche benutzt werden, um die Ausrichtung von Objekten im Raum besser zu beschreiben. Als Illustration hierfür dient Abbildung 1.2. Hier ist kurz anzumerken, dass es zwar insgesamt sechs Freiheitsgrade in der Physik gibt, aber nicht alle zwingend relevant sind. Weil sich zweirädrige Fahrzeuge auf einer planaren Ebene mit unidirektionalen Rädern fortbewegen, liegt der Fokus auf der Vorwärtsbewegung, der Roll- und Yaw-Achse.

Die primäre Aufgabe des Gyroskops ist dabei, die Roll-Achse stabil zu halten, um ein Umkippen zu vermeiden. Die sekundäre Aufgabe ist es, Änderungen auf der Yaw-Achse auszumessen, weil diese eine wichtige Rolle in der Fahrtrichtung spielt. Bei zweirädrigen Fahrzeugen wird die Fahrtrichtung mithilfe des Rolls und Yaws modifiziert. Vereinfacht dargestellt besitzen zweirädrige Fahrzeuge jeweils nur einen Kontakt mit der Vorderrad- und Hinterradachse. Das hat zur Folge, dass die Zentrifugalkraft, zweirädrige Fahrzeuge aus einer eingeschlagenen Kreisbahn herausdrückt.

Des Weiteren hat sich die Arbeit von Hunziker[19] mit der Sensor Fusion und der Implementierung auf einem SoC beschäftigt. Dabei sollen mehrere Sensoren zu einem kohärenten

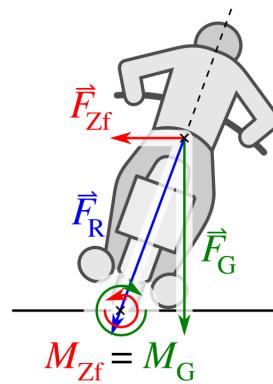


Abb. 1.3: Schräglage eines Motorrads in einer Rechtskurve. \vec{F}_{Zf} ist dabei die Zentrifugalkraft, die das Motorrad aus der Kurve zieht.

Quelle: <https://de.wikipedia.org/wiki/Datei:Dynamisches-gleichgewicht-motorrad-kurvenfahrt.svg>, zuletzt besucht 01.09.2024

System vereint werden. Als Grundlage für dieses System dient ein Raspberry Pi 4.[20] Die fusionierten Sensoren sind ein Ultraschallsensor, ein Light Detection and Ranging-Sensor, ein GPS-Tracker und eine Videokamera. Während Themen wie Model Predictive Control und Edge Detection bereits eingeführt wurden, soll diese Ausarbeitung hier ansetzen, ausbauen und weiterentwickeln. Weil ein RaspberryPi 4 nur wenig Prozessorleistung besitzt, ist es nicht möglich, eine Echtzeitanwendung mit einer Inferenz-Pipeline zu integrieren. Außerdem fehlt die nötige Kameraausstattung, weil hier nur eine RaspberryPi Camera Module 3 verwendet wurde. Also wird ein neuer Ansatz benötigt.

2 Konzeptionierung und Realisierung

In diesem Kapitel werden die Problemstellungen, Grundlagen, der aktuelle wissenschaftliche Stand berücksichtigt und ein theoretisches Konzept entwickelt. Dieses theoretische Konzept soll im Anschluss versucht werden, unter Einhaltung der technischen und finanziellen Rahmenbedingungen, zu realisieren. Da es sich hier um ein sehr facettenreiches und multivariates Problem handelt, ist es sehr unwahrscheinlich, das theoretische Konzept ohne weitere Probleme in die Realität umzusetzen. Daher sollte man bei der theoretischen Konzeptionierung eine sehr offene, ganzheitliche Herangehensweise beibehalten. Dadurch ist es einfacher, Änderungen im Konzept vorzunehmen, die technisch sonst nicht realisierbar wären.

Außerdem ist ein weiterer Faktor die Kompromissbereitschaft zwischen zwei Problemen. Es kann vorkommen, dass sich die Lösung eines Problems negativ auf ein anderes Problem auswirken kann. Ein konkretes Beispiel hierfür ist der Tradeoff zwischen Genauigkeit und Geschwindigkeit im Bilderkennungsprozess. Hierbei ist es unter anderem möglich, sehr akkurate Ergebnisse mit großen Maschinenmodellen zu erzielen. Jedoch wirkt sich die Größe des Maschinenmodells umgehend auf die Dauer der Inferenz aus.

Ein weiterer Aspekt sind die unterschiedlichen Programmiersprachen, Frameworks, die zur Verfügung stehen. Der entwickelte Prototyp ist hauptsächlich in C und Python entwickelt worden. Daher ist es ratsam, das bestehende System so homogen wie möglich zu halten, um Overheads, Arbeitsaufwand, Kompatibilitätsprobleme und Fehlerquellen zu reduzieren.

2.1 Übersicht

Eine Theoretische Konzeptionierung und technische Realisierung sind zwei unterschiedliche Disziplinen. Deshalb benötigt man zunächst eine konkrete Analyse zur Umsetzbarkeit. Aus der Problemstellung lassen sich bereits erste Konditionen ableiten. Diese sollten jedoch in essenzielle und nicht essenzielle Konditionen kategorisiert werden. Es handelt sich bei der Problemstellung um ein multivariates Problem, weswegen eine Reihenfolge von Prioritäten durchaus angebracht ist. Im folgenden Abschnitt sind die Unterkapitel entsprechend von höchster Priorität zur niedrigsten Priorität angeordnet.

2.1.1 Mobilität

Eine essenzielle Anforderung ist die Mobilität. Da sich das Fahrrad idealerweise auf Straßen und Wegen frei bewegen soll, ist diese Anforderung unerlässlich. Jedoch hat die Erfüllung dieser Anforderung auch negative Auswirkungen. Durch die Priorisierung von Mobilität wird der Formfaktor für die technische Realisierung limitiert. Das hat zur Folge, dass leistungsstarke Hardware wie dedizierte Grafikkarten, wie man es aus x86-Architekturen kennt, nicht benutzt werden können. Außerdem ist hierdurch auch implizit die Leistungsaufnahme eingeschränkt, weil das Fahrrad nur mit 36-Volt-Batterien ausgestattet ist. Des Weiteren entfallen Technologien wie Datentransfer mittels Wi-Fi zur Sensor- und Bildübertragung zu einem leistungsstarken System, weil diese mit parasitären Größen befallen sind. Diese ist zum einen die zunehmende Distanz vom Fahrrad zum Datenverarbeitungssystem, zum anderen die Interferenz von umliegenden Signalen mit der eigenen Signalfrequenz. Ebenso birgt eine drahtlose Verbindung die Gefahr, dass zwischen dem Fahrrad und dem Datenverarbeitungssystem andere Verkehrsteilnehmer oder Hindernisse geraten können. Dies würde die Reichweite signifikant reduzieren. Aileen et al. [21] haben festgestellt, dass insbesondere Beton, bei einer Materialdicke von 15 cm, ein 2,4 GHz-Wi-Fi-Signal um ca. 30 % reduzieren kann. Auch andere Materialien wie Kunststoff, die dünner ausfallen, können bereits Auswirkungen haben. Für Kunststoff wurde eine Abschwächung von 7,94 % pro cm Materialdicke ermittelt.

2.1.2 Echtzeitkritisches System

Noch wichtiger als die Mobilität ist eigentlich die Echtzeitfähigkeit eines eingebetteten Systems. Jedoch wurde diese der Mobilität untergeordnet aus dem einfachen Grund, weil ein System, welches sich nicht in den Prototypen integrieren lässt, nicht den Anforderungen entspricht. Prof. Dr. Brinkschulte et al. (2021)[22] unterteilen eingebettete Systeme in mehrere Klassen:

- Nicht-Echtzeitsysteme
- Echtzeitsysteme
- Weiche Echtzeitsysteme
- Feste Echtzeitsysteme
- Harte Echtzeitsysteme

Die Echtzeitfähigkeit im Straßenverkehr ist unerlässlich und eine der maßgebenden Kriterien im Zusammenhang mit autonomem Fahren. Deshalb werden Fahrassistenten, autonome Systeme, Spurassistenten und weitere automatisierte Fahrhilfen als ein hartes Echtzeit-system eingestuft. Ein hartes Echtzeitsystem hat zwei Hauptanforderungen. Zum einen die logische Korrektheit, zum anderen die zeitliche Korrektheit.[23]

Die logische Korrektheit ist die Anforderung an das System, dass sowohl syntaktisch als auch semantisch das System abgeschlossen ist. Zusätzlich müssen in allen Zuständen korrekte Ergebnisse liefern kann. Vor allem im Kontext des autonomen Fahrens ist es unerlässlich, dass diese Anforderung stets erfüllt bleibt, da sonst die Gefahr besteht, dass unter anderem Personenschaden oder Schaden am Fahrrad entstehen kann.

Die zeitliche Korrektheit ist die Anforderung an das System, Ergebnisse bzw. Aufgaben unter Vorgabe von festen Zeitschränen zu liefern und abzuschließen. Je nach Signifikanz der zeitlichen Relevanz für eine bestimmte Aufgabe im System können Ergebnisse verworfen werden, eine Routine zum Handling des Zeitschränkenverstoßes gestartet oder auch das System gestoppt werden. Ein Beispiel hierfür wäre eine Anhaltroutine, die das System ordnungsgemäß herunterfährt und stoppt.

Beide Hauptanforderungen müssen hinreichend erfüllt sein, damit das eingebettete System als ein hartes Echtzeitsystem kategorisiert werden kann und im Straßenverkehr benutzt werden darf. Aus diesen beiden Anforderungen lassen sich bereits mehrere Konditionen für das Systemdesign extrapolieren. Aus der logischen Korrektheit lässt sich ableiten, dass die Aussagekraft, der Wahrheitsgehalt von Sensorgrößen und Bildverarbeitung stets korrekt interpretiert werden muss. Damit wird verhindert, dass das System nicht in einen fehlerhaften Zustand fällt. Zusätzlich müssen die Regelkreise zur Kontrolle der Fahrtgeschwindigkeit und zur Vorhersage der Strecke fehlerresistent konzipiert sein.

Die zeitliche Korrektheit in diesem Kontext ist die Compoundierung der zuvor genannten Konditionen relativ zur Zeit. Das heißt, dass es nicht nur ausreichend ist, die zeitliche Korrektheit einzuhalten oder die logische Korrektheit zu erfüllen, sondern die logische Korrektheit unter Einhaltung gegebener Zeitschranke zu gewährleisten. Dieses kleine Detail ist besonders wichtig, weil es in einem Echtzeitsystem zu Race Conditions kommen kann.

Um Race Conditions genauer zu verstehen und warum sie auftreten, ist es notwendig, einen Blick in die Echtzeitprogrammierung zu werfen. Hier wird zwischen synchroner und asynchroner Programmierung unterschieden. Der Hauptunterschied dieser beiden Paradigmen ist, dass die synchrone Programmierung auf einem zeitlichen Intervall basiert, wohingegen die asynchrone Programmierung sowohl auf zeitlichen Intervallen basiert sein kann, als auch auf aperiodische Ereignisse nativ reagieren kann. Bei synchroner Programmierung sind die Hauptsteuerung des Programms und die Pipeline der Signal- und Datenverarbeitung an den Takt bzw. die Zeitschranke gebunden. Zwar wird damit das Einhalten der Zeitschränke trivialisiert, jedoch erschwert es, aperiodische Ereignisse abzufangen. In solch einem Fall wäre es notwendig, ein hochfrequentes Polling einzurichten, um Änderungen an Eingangssignalen oder auch in Bildverarbeitungsabläufen zu detektieren. Zusätzlich ist dies keine Garantie für einen synchronen Programmablauf, da Teile der Pipeline asynchrone Prozesse sein können und somit keine fixe Ausführungszeit haben. In solchen Fällen kann es zur Verletzung der Zeitschranke und zu Race Conditions kommen. Hier steht man

vor der einer Auswahl und müsste entscheiden, ob man verspätete Ergebnisse berücksichtigt oder das Prozessieren stoppt und der nächste Zyklus beginnt, weil die Zeitschranke verletzt wurde. Genauso wenig hat man die Gewissheit, ob das System in einem sicheren Zustand operiert, wenn Ausgangssignale nicht existieren.

Der Vorteil beim asynchronen Programmieren ist, dass das Hauptprogramm ereignisgesteuert sein kann. Das hat zur Folge, dass man zum bestmöglichen Zeitpunkt mit der Signal- und Datenverarbeitung beginnen kann. Jedoch gewährleistet dies nicht die Einhaltung der Zeitschranken und muss gründlich getestet werden. Hierfür eignet sich beispielsweise cProfiles aus der Python Standard Library.[24]

2.1.3 Formfaktor und Leistungsaufnahme

Wie bereits im Abschnitt 2.1.1 aufgegriffen wurde, ist eine weitere Anforderung der Formfaktor und die Leistungsaufnahme. Da es sich um eine mobile Plattform handelt, ist das Montieren von Hardware und auch deren Stromversorgung nur begrenzt möglich. Es folgen deshalb zwei Einschränkungen, zum einen die begrenzte Leistungsaufnahme, zum anderen die kleine Auswahl an Systemarchitekturen.

Die Leistungsaufnahme ist eine essenzielle Ressource und durch den Formfaktor und die Stromversorgung limitiert. Dadurch sollte der Fokus bei Mikrocontrollern oder System on Chip (kurz SoC)-Geräten liegen. Diese beiden Architekturen sind sowohl energieeffizient als auch kompakt und eignen sich optimal für diese Anforderungen. Im Angesicht, dass der Schwerpunkt dieser Arbeit im Computer Vision Bereich liegt, ist es ebenfalls unerlässlich, dass sich eine Grafikbeschleunigungseinheit (kurz GPU) im System befindet. Jedoch ist im Markt für Mikrocontroller und SoC-Geräte die Auswahl sehr beschränkt. Nach dem aktuellen Stand der Recherche (März 2024), eignen sich unter anderem Mikrocontroller wie der BeagleBone AI oder auch Derivate des Jetson Nano von NVIDIA.

2.1.4 Framework

Bei der gesamten Konzeptionierung des Projekts ist es notwendig, sich im Vorfeld zu informieren, welche Frameworks existieren. Vor allem im Computer Vision-Bereich gibt es bereits mächtige Toolboxen und Libraries wie Open CV oder auch ULTRALYTICS YOLO.[10] Dabei handelt es sich um stark optimierte und standardisierte Library, welche die Integration in ein bereits bestehendes System vereinfacht und ebenfalls mit der Laufzeitoptimierung helfen kann. Außerdem müssen Geräte wie der Beagle Bone Black (kurz BBB), der Mikrocontroller des Fahrrads, mit neuer Hardware kompatibel sein. Weil sich Regelkreise, Steuer- und Lenklogik auf dem BBB befinden, müssen hier Zustandsdaten für die weitere Computer Vision-Logik herausgelesen werden. Des Weiteren muss das

neue System Kursangaben und Korrekturen zurück zum BBB kommunizieren. Dafür eignet sich etwa ein Ethernet Interface, um eine bidirektionale Schnittstelle zu bieten. Diese ist universal und bietet eine nahezu echtzeitfähige Übertragung. Die niedrige Latenz ist größtenteils zu vernachlässigen, weil sich die jeweiligen Komponenten in kurzer Proximität zueinander befinden.

Das Framework zum Einbinden einer digitalen Kamera ist ebenfalls von Relevanz. Je nach ausgewählter Systemarchitektur, muss man hier zwischen zwei Kamera-Interfaces, MIPI-CSI und USB, unterscheiden. Diese haben je nach Anforderung unterschiedliche Vor- und Nachteile.

Mobile Industry Processor Interface-Camera Serial Interface (kurz MIPI-CSI) ist ein Industriestandard für Mikrocontroller und SoC-Geräte. Dieser Standard hat den Vorteil, dass sich Kamerasensoren direkt an der Platine anschließen und kontrollieren lassen. Daraus resultiert eine sehr *Bare Metal*-Programmierung, da der Kamerasensor wie eine interne Komponente des Systems behandelt werden kann. Latenzen und Computational Overheads werden auf ein Minimum reduziert, was für echtzeitfähige Systeme kritisch ist. Einer der Nachteile im Vergleich zu USB-Kameras ist, dass Treiber für bestimmte Platinen nicht existieren und Kompatibilität nicht immer gewährleistet ist. Dadurch ist die Auswahl an Kamerasensoren beschränkt.

Bei USB-Kamerasensoren sind die Vor- und Nachteile im Vergleich zu MIPI-CSI Kamerasensoren umgedreht. Wohingegen USB-Kameras eine weite Kompatibilität mit diversen Systemarchitekturen genießen, entsteht jedoch durch diese universale Verfügbarkeit sowohl ein zeitlicher Overhead als auch eine Ressourcen fordernde Konstellation. Die Ursache dafür ist, dass Universal Serial Bus (kurz USB)-Geräte, die Bildverarbeitung und der daraus entstehende Data Stream in ein USB-Protokoll encoded wird und im Anschluss auf dem Zielsystem decoded werden muss. Zusätzlich läuft man Gefahr, die Bandbreite des USB-Protokolls auszulasten.[25] Ein regulärer Full-HD-Videostream mit einer Auflösung von 1920x1080 bei 30 Frames Per Second (kurz FPS) und einer Pixeltiefe von 8 Bits resultiert in einer Bandbreitenanforderung von mindestens $497.664.000 \frac{\text{Bits}}{\text{s}}$ bzw. $59,33 \frac{\text{MBytes}}{\text{s}}$. Die Berechnung hierzu ist wie folgt:

$$\text{Bandbreite} = \frac{\text{Pixelbreite} \cdot \text{Pixelhöhe} \cdot \text{Bildwiederholrate} \cdot \text{Pixeltiefe}}{s} \quad (2.1)$$

$$59,33 \frac{\text{MBytes}}{\text{s}} = \frac{1920 \cdot 1080 \cdot 30 \cdot 8 \text{ Bits}}{2048 \cdot 8 \cdot s} \quad (2.2)$$

Bei einem USB 2.0-Peripheriegerät wäre dabei die Bandbreite ausgeschöpft, da die theoretische Übertragungsrate bei $480 \frac{\text{Mbps}}{\text{s}}$ bzw. $60 \frac{\text{MBytes}}{\text{s}}$ liegt.[26] Weil jedoch die Stromversorgung ebenfalls über den USB-Port läuft, können nie alle Lanes im USB-Kabel als Data Lanes benutzt werden. Obwohl ein USB 3.0-Gerät ca. die zehnfache Bandbreite[27] hätte,

um dieses Problem zu umgehen, bleibt der Overhead und die Ressourcenanforderung an die CPU weiterhin bestehen.

2.2 Theoretisches Konzept

In diesem Abschnitt wird das angestrebte Proof of Concept vorgestellt. Dabei wird sowohl die Problemstellung als auch die Übersicht berücksichtigt und eine vollständige Prozess-Pipeline zusammengesetzt. Zusätzlich ist zu beachten, dass die Fördermittel für das Projekt begrenzt sind und dementsprechend ein Kompromiss zwischen Ausgaben und Leistung gefunden werden muss. Am Ende dieses Abschnitts soll ein vollständiger Bauplan für ein eingebettetes System existieren, das technisch und finanziell realisierbar ist.

2.2.1 Hauptplatine

Die Hauptplatine ist der zentrale Baustein dieses Projekts und setzt die Rahmenbedingungen für alle sukzessiven Designentscheidungen der Pipeline. Wie bereits im Abschnitt 2.1.1 angeschnitten, unterliegt die Auswahl der Hauptplatine und der daraus resultierenden Systemarchitektur diversen Anforderungen. Um hier einen Kompromiss zwischen Leistung, Mobilität und finanziellen Mitteln zu finden, fokussiert sich die Recherche auf Einplatinencomputer. Dabei handelt es sich um ARM-Prozessoren basierte Systeme. Advanced RISC Machines (kurz ARM) ist eine Mikroprozessor-Architektur, die von ARM Limited entwickelt wird. Diese Prozessoren haben die Besonderheit, dass sie in ihrer Designphilosophie auf Reduced Instruction Set Computation (kurz RISC) setzen. Das Designziel solcher Prozessoren ist, mit einem kleinen Befehlssatz ausgestattet zu sein und die Ausführung dieser Befehle mit so wenigen Taktzyklen wie möglich geschehen zu lassen.[28] Das hat den Vorteil, dass man sowohl höhere Taktraten erreichen kann, energieeffizienter ist als das x86-Pendant.[29] Die leistungsstärksten Prozessoren im kommerziellen Sektor wie dem Intel I9-13900K mit $2,238 \frac{\text{TFLOP}}{\text{s}}$ [30] und AMD Ryzen 9 7950X3D mit $2,619 \frac{\text{TFLOP}}{\text{s}}$ [30] haben ausreichend Leistung für Inferenz-Berechnungen. Sie sprengen jedoch den finanziellen Rahmen des Projekts, laufen Gefahr thermisch gedrosselt zu werden, verbrauchen zu viel Energie und sind wegen des Formfaktors nur schwer am Fahrrad anzubringen.

Obwohl x86-Architekturen eine höhere Performance haben können, eignen sie sich für den Anwendungsfall dieser Masterarbeit nicht. Sie verbrauchen mehr Energie und sind auch nicht geeignet als primäre Verarbeitungsplattform. Bildverarbeitung mittels künstlicher Intelligenz profitiert enorm von parallelisiertem Rechnen, primär beim Trainieren von Maschinenmodellen. Während des Trainings- und Inferenzprozesses werden Daten in Batches gepackt. Über diese Batches hinweg werden diverse Matrix-Matrix-, Matrix-Vektor oder Multiplizieren-Akkumulieren-Operationen ausgeführt. In den meisten Fällen bestehen die

Operanden aus *FLOATs*, weswegen die Leistungsaufnahme auch oft in *Floating Point Operations per Second* (kurz FLOPs) gemessen wird. Hierfür eignen sich Grafikbeschleuniger, weil sie durch den breiten bzw. parallelen Aufbau ihrer ALU-Architekturen es erlauben, einen hohen FLOPs-Durchsatz zu erzielen. Im Einplatinenbereich kommt deshalb nur ein System infrage, der NVIDIA Jetson Nano.

2.2.1.1 NVIDIA Jetson Nano

NVIDIA bietet mit der Jetson Nano-Reihe eine ARM-basierte Plattform mit einem integrierten Grafikbeschleuniger an. Mit $472 \frac{GFLOP}{s}$ [9], einer maximalen Leistungsaufnahme von maximal 20 W, dem kleinen Formfaktor, einem guten Preis-Leistungs-Verhältnis und einem MIPI-CSI-Anschluss, eignet sich das Einsteigermodell hervorragend. Außerdem bietet NVIDIA mit *Jetpack SDK*[31] einen umfangreichen Softwarestack, der die Integration und Softwareentwicklung erleichtert. Der größte Nachteil der Plattform ist jedoch der Arbeitsspeicher. Dieser wird nämlich vom ARM-Prozessor und der Grafikeinheit als ein Shared Access Memory verwendet. Mit 4 GB sind somit die Ressourcen eingeschränkt. Das hat vor allem eine Auswirkung auf unterschiedliche Modellgrößen, da je nach Größe mehr Arbeitsspeicher für den Inferenzprozess benötigt wird. Des Weiteren wirkt sich ein kleiner Arbeitsspeicher unmittelbar auf die Dauer des Trainings für das Maschinenmodell aus, weil Trainings-Batches kleiner gehalten werden müssen.

2.2.2 Kamerasensor

Eine weitere Hardwarekomponente ist der Kamerasensor für das System. Weil der Jetson Nano über MIPI-CSI- und USB3.0-Anschlüsse verfügt, stehen zwar viele kompatible Optionen zur Verfügung, jedoch sollen folgende Kameraspezifikationen beachtet werden:

- Auflösung
- Seitenverhältnis
- Sichtfeld (Field of View)
- Bildwiederholrate
- Brennpunkt (Fokus)
- Verschlusstechnologie (Shutter)

2.2.2.1 Auflösung

Die Bildqualität ist wichtig, weil für jede KI-basierte Applikation die Prämisse "*Garbage in, Garbage out*" gilt. Im übertragenen Sinne heißt das, dass die Qualität von Trainings-,

Test- und Validierungsdatensätzen sich sehr stark auf die Performance des KI-Modells auswirken wird. Genauso muss im Inferenzprozess eine ausreichend gute Bildqualität gewährleistet werden. Eine Auflösung von 1920x1080 Pixeln wäre deshalb ideal. Hier ist ebenfalls anzumerken, dass höhere Auflösungen mehr Bandbreite und eine größere Belastung für die CPU des Systems sind. Mit Hinsicht auf die Verwendung von KI-Modellen ist außerdem der Berechnungs overhead relativ zur Bildauflösung zu beachten. KI-Modelle von ULTRALYTICS beispielsweise benutzen für ihre Leistungsbenchmarks und Standardeinstellungen eine Auflösung von 640x640.[10] Um das Seitenverhältnis und infolgedessen die Geometrien im Bild nicht zu verzerrn, findet hier ein Resizing statt. Dabei wird die längste Seite des Bilds auf die eingestellte Auflösung herunterskaliert und die kürzere Seite mit Padding von schwarzen Pixeln aufgefüllt.

2.2.2.2 Seitenverhältnis und Sichtfeld

Ein optimales Seitenverhältnis ist hauptsächlich abhängig von der Kameraausrichtung und des einfangbaren Sichtfelds mit der Kameralinse. Eine Landscape-Ausrichtung hat den Vorteil bei breiten Straßen die gesamte Fahrbahn und Verkehrsteilnehmer zu erfassen. Im Fall der Portrait-Ausrichtung hat man den Vorteil, dass man die voraus liegende Verkehrsführung detaillericher einfangen kann. Das setzt jedoch voraus, dass die Kamera am Prototypen höher angebracht werden muss als bei einer Landscape-Ausrichtung. Jedoch entsteht bei einer hohen Kameramontage in beiden Ausrichtungsmöglichkeiten ein vergrößerter toter Winkel unterhalb der Kameraposition.

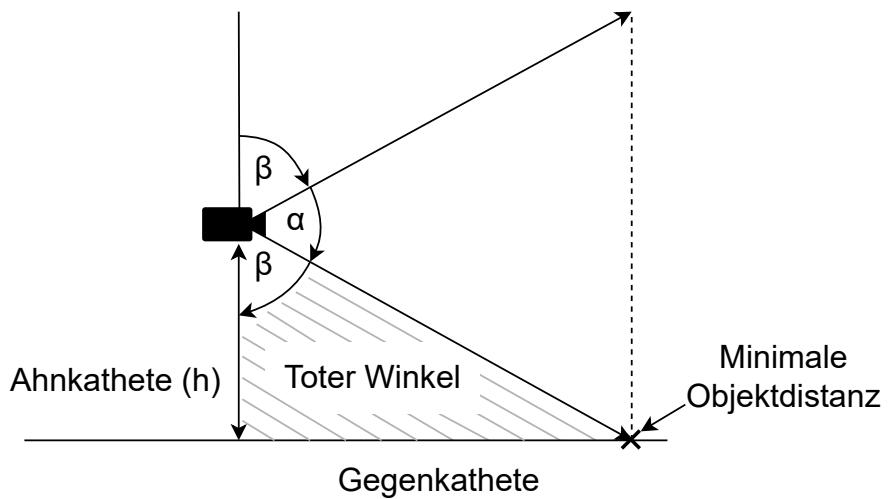


Abb. 2.1: Darstellung des toten Winkels und Berechnung der minimalen Objektdistanz

Mittels Trigonometrie, Höhe der Kameraplatzierung und vertikalem Sichtfeldwinkel lässt

sich die minimale Objektdistanz und entsprechend die Größe des toten Winkels berechnen. Sei α der vertikale Sichtwinkel, so gilt:

$$\beta = \frac{180^\circ - \alpha}{2} \quad (2.3)$$

Hieraus lässt sich nun die Gegenkathete ableiten mit:

$$\text{Gegenkathete} = \tan(\beta) \cdot \text{Ankathete} \quad (2.4)$$

Wenn man das Modell in geringen Maßen abstrahiert, kann man davon ausgehen, dass die Kamera und der Boden parallel zueinander platziert sind. Wenn man nun eine Orthogonale durch die Kamera- und Bodenebene legt, ist die Gegenkathete in diesem Fall der Abstand von der Kamera zur minimalen Objektdistanz auf der Bildfläche, siehe Abbildung 2.1. Diesen Abstand gilt es minimal zu halten. Der Grund dafür ist, dass der tote Winkel unterhalb der Kamera sich bei hoher Platzierung vergrößert. Aus der Gleichung 2.4 und Abbildung 2.1 kann entnommen werden, dass die Ankathete bzw. Höhe der Kamera direkt mit der Größe des toten Winkels korrelieren. Da α und β abhängig von der Wahl der Kamera sind und konstant bleiben, lässt sich mit der Winkelgröße nur begrenzt der tote Winkel modifizieren. Versucht man jedoch, den toten Winkel zu verkleinern, tritt das Problem auf, dass sich die Distanz des Horizonts verkleinert.

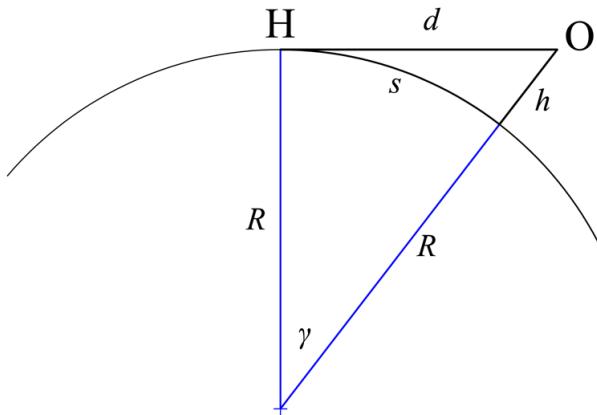


Abb. 2.2: Geometrische Darstellung der Horizontberechnung

Quelle: <https://commons.wikimedia.org/w/index.php?curid=14958770>, zuletzt besucht 10.07.2024

Mit dem Satz des Pythagoras lässt sich diese berechnen. Sei R der Radius der Erde, so gilt:

$$(R + h)^2 = R^2 + (BC)^2 \quad (2.5)$$

$$(BC)^2 = R^2 + 2 \cdot Rh + h^2 - R^2 \quad (2.6)$$

In Anbetracht, dass $R \gg h$, lässt sich der Ausdruck vereinfachen als:

$$BC \approx \sqrt{2 \cdot Rh} \quad (2.7)$$

Selbst bei einer niedrigen Höhe von 0,75 m und dem durchschnittlichen Erdradius von 6.371 km[32] ergibt sich daraus eine Distanz zum Horizont von ca. 3.091 m. Dies lässt suggerieren, dass man die Kamera um ihre eigene Vertikalachse rotieren lassen kann, um den Winkel β zu verkleinern und somit auch den toten Winkel.

Eine weitere Gefahr, die hier auftreten kann, ist der rapide Abfall der Straßenführung. Man kann die Straße als eine stetige Funktion interpretieren. Wenn man sich auf einem lokalen Maximum befindet und ein starkes Gefälle einsetzt, kann es zu einem temporären Runaway kommen. Ein Runaway in diesem Kontext ist ein Gefälle, das sich im toten Winkel der Kamera befindet und die Detektion der Straße unmöglich macht. Dieses Phänomen kommt genau dann vor, wenn man sich an einem lokalen Maximum befindet und das Gefälle größer ist als das Gefälle der linearen Funktion für die minimale Objektdistanz relativ zur Kameraposition.

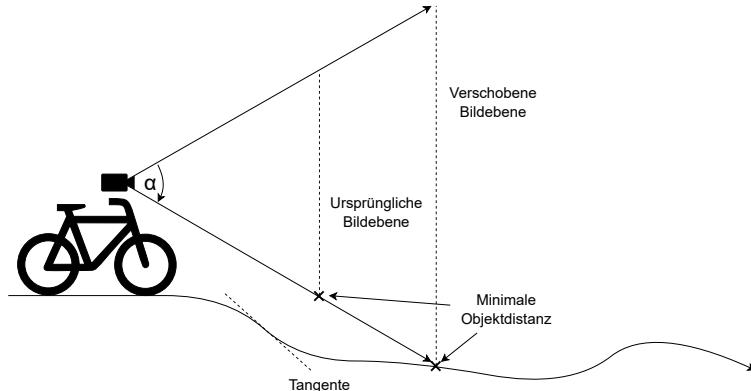


Abb. 2.3: Illustration eines Runaways und der entsprechenden Tangente

Für das maximale Gefälle muss zu jedem Zeitpunkt gelten:

$$m_{max} \leq \frac{y_2 - y_1}{x_2 - x_1} \quad (2.8)$$

Dadurch lässt sich ein Runaway vermeiden und garantiert, dass die Hypotenuse immer einen Schnittpunkt mit der Straßenoberfläche hat.

2.2.2.3 Bildwiederholrate

Als nächster zu beachtender Punkt ist die Bildwiederholrate. Hier kommen Aspekte der Echtzeitfähigkeit 2.1.2 ins Spiel. Je höher die Bildwiederholungsrate ist, umso frühzeitiger kann das System den Inferenzprozess starten. Dadurch wird das System umso reaktionsfähiger. Weit verbreitete Standards sind zum einen 30 und 60 FPS. Diese Geschwindigkeiten sind äquivalent zu Verzögerungszeiten zwischen zwei Bildern von ca. 33 ms und ca. 17 ms. Im Idealfall sollte die Pipeline in der Lage sein jedes Bild zu prozessieren. Jedoch lässt sich diese Performance nur schwer ermitteln, da mehrere Parameter die Inferenzdauer beeinflussen. Mehr dazu wird in Abschnitt 2.2.3 erörtert.

2.2.2.4 Verschluss

Das letzte Auswahlkriterium ist das Shutter-System der Kamera. Der Shutter in einer Kamera ist für das Belichten, Verschließen und Abbilden von Pixeln auf dem Kameratasensor verantwortlich. Für diesen Prozess gibt es zwei weitverbreitete Technologien: Rolling Shutter und Global Shutter.

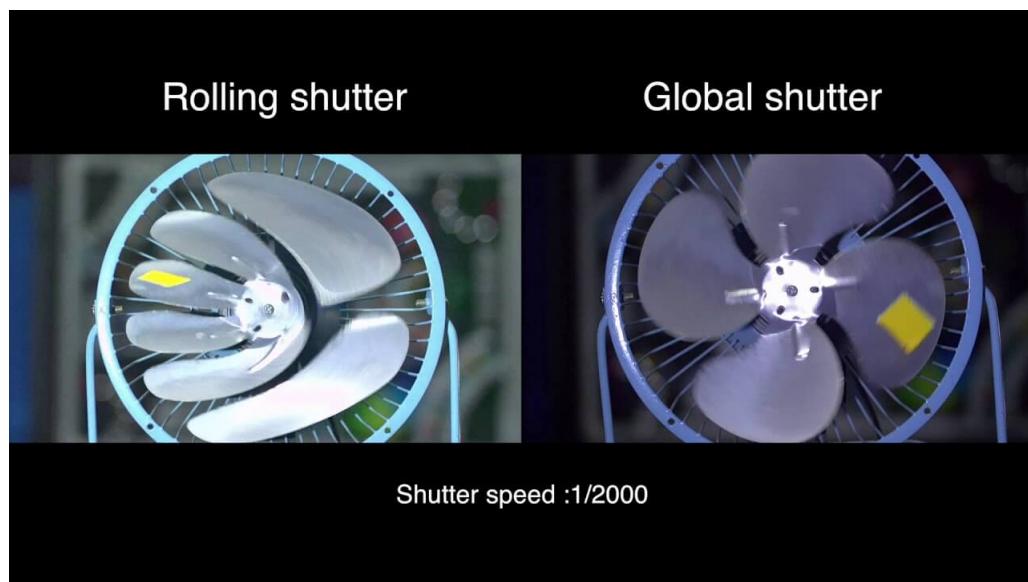


Abb. 2.4: Illustration des Rolling Shutter- Effekts

Quelle: <https://s.studiobinder.com/wp-content/uploads/2022/04/Rolling-Shutter-vs-Global-Shutter.jpg>, zuletzt besucht 17.07.2024

Der Rolling Shutter ist eine kostengünstige und effektive Technologie. Dabei löst ein elektrischer Impuls das Auslesen auf dem Kameratasensor aus. Ein Kameratasensor besteht aus einem zweidimensionalen Array von Photodioden.[33] Während des Ablichtungsprozesses werden beim Rolling Shutter Reihen des Arrays sequenziell vom oberen Bildrand in Richtung des unteren Bildrands ausgelesen. Für statische Szenen ist hier kein Unterschied in

der Fotoqualität zu bemerken. Wenn sich jedoch das Zielobjekt oder auch der Kameratasensor in Bewegung befinden, können sich relativ zur Bewegungsgeschwindigkeit leichte bis schwerwiegende parasitäre Qualitätsmängel im Bild bemerkbar machen.[34] Der Rolling Shutter-Effekt ist dabei eine technologische Limitation. Weil das Auslesen des zweidimensionalen Arrays sequenziell ist, treten hierbei schnell bewegende Objekte unerwünschte Verzerrungen auf. Diese können zwar reduziert werden, wenn die Belichtungszeit bzw. Verschlussdauer minimiert wird, können aber beim Rolling Shutter nicht gänzlich vermieden werden. Der Grund dafür ist, dass bei sehr kurzen Belichtungszeiten nicht genug Photonen vom Kameratasensor erfasst werden können und somit das Bild verdunkelt oder farblich inkorrekt dargestellt wird.

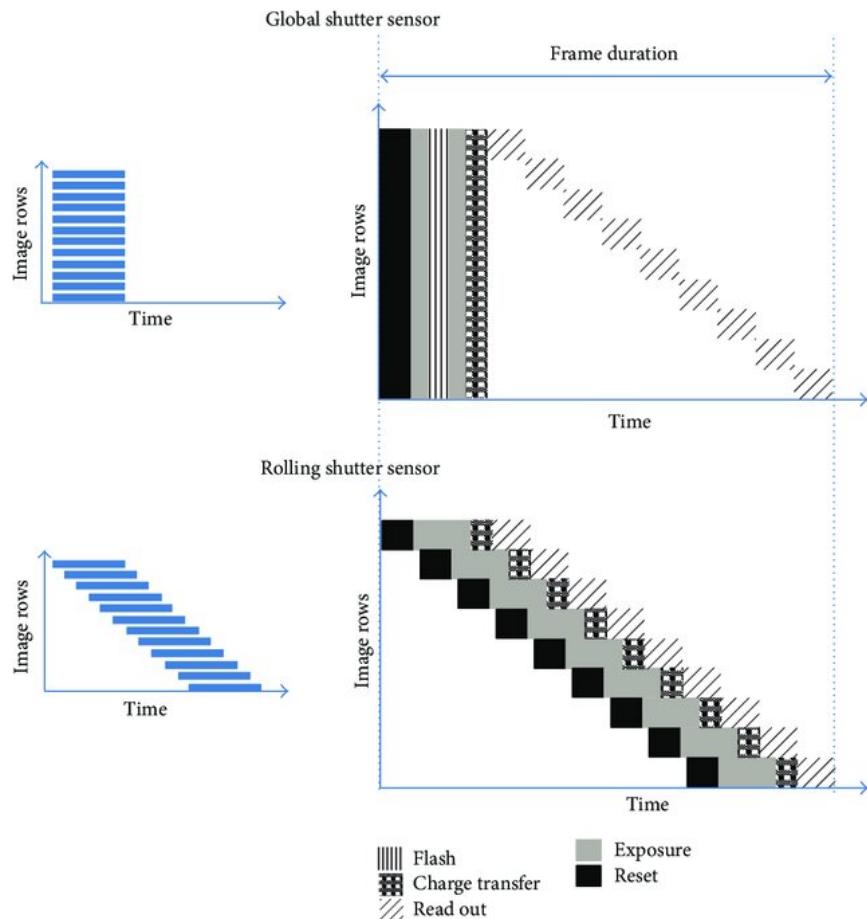


Abb. 2.5: Visualisierung des technischen Unterschieds eines Rolling Shutters und Global Shutters
 Quelle: https://www.researchgate.net/publication/303816203/figure/fig6/AS:668376791785485@1536364853424/Global-shutter-and-rolling-shutter-operation-17-18_W640.jpg, zuletzt besucht 17.07.2024

In Abbildung 2.4 ist ein Vergleich zwischen einer Rolling Shutter- und einer Global Shutter-Kamera zu sehen. Wie der Name suggeriert, werden bei der Global Shutter-Technologie alle

Photodioden auf dem Kamerasensor zum selben Zeitpunkt ausgelesen. Dadurch lässt sich ein schärferes Bild erzeugen. Der Nachteil ist jedoch, dass die Photodioden und entsprechende CMOS-Schaltung bei Global Shutter-Kameras größer ausfallen. Deswegen haben Global Shutter-Systeme bei gleichbleibender Sensorfläche weniger Photodioden, was in einer niedrigeren Auflösung resultiert. Der technische Mechanismus hierzu ist ebenfalls in Abbildung 2.5 illustriert. Hier erkennt man das "Rollen" des Verschlusses.

Eine weitere parasitäre Komponente ist die Bewegungsunschärfe. Diese tritt in beiden Verschlussmechanismen auf und korreliert direkt mit der Belichtungsdauer. Je länger der Kamerasensor belichtet wird, umso mehr entsteht Bewegungsunschärfe. Wie bereits erwähnt, kann man jedoch die Belichtungsdauer nicht indefinit verkürzen.

Unter Berücksichtigung aller Faktoren wurde sich auf die MIPI-CSI-Kamera e-CAM24_CUNX - Color Global Shutter Camera von e-con Systems entschieden.[35] Die e-CAM24 ist in der Lage in verschiedenen Modi zu operieren. Besonders wichtig sind dabei der HD- und Full-HD-Modus. Der HD-Modus besitzt eine Auflösung von 1270x720 und eine Bildwiederholrate von 120. Dieser Modus in Kombination mit dem eingebauten Global Shutter sollte für eine hohe Visual Clarity sorgen. Hier treten keine Rolling Shutter-Effekte auf. Die Bewegungsunschärfe ist reduziert durch die kurze Belichtungszeit und eine hinreichend hohe Auflösung ist vorhanden.[36]

Der Full-HD-Modus hingegen operiert bei einer Auflösung von 1920x1080 und einer Bildwiederholrate von 60. Hier wird die Visual Clarity durch eine höhere Auflösung erzielt und sollte in langsameren Bewegungsabläufen und in dunkleren Szenen bessere Resultate erzielen, weil die Belichtungszeit länger ausfällt.

Außerdem besitzt die Kamera einen Modus mit einem kleineren Seitenverhältnis, einer Auflösung von 1920x1200, einer Bildwiederholrate von 60. Dieser könnte von Relevanz sein, um den toten Winkel unterhalb der Kamera zu verkleinern. Weil die e-CAM24 ein vertikales Sichtfeld von $61,6^\circ$ hat, fällt dieser besonders groß aus. Mit der Gleichung 2.3 und Gleichung 2.4 lässt sich die minimale Objektdistanz berechnen. Somit ergibt sich für β :

$$\begin{aligned}\beta &= \frac{180^\circ - 61,6^\circ}{2} \\ &= 59,2^\circ\end{aligned}$$

Entsprechend ermittelt sich die minimale Objektdistanz wie folgt:

$$\begin{aligned}\text{Gegenkathete} &= \tan(\beta) \cdot \text{Ankathete} \\ &= \tan(59, 2^\circ) \cdot 0,75 \text{ m} \\ &= 1,25 \text{ m}\end{aligned}$$

Durch das große vertikale Sichtfeld ist es möglich, die Kamera in Richtung des Bodens zu rotieren, um den toten Winkel weiter zu verkleinern. Hier erfordert es jedoch experimentelle Justierung, da der Fokus der Kamera fixiert ist. Weil ein Autofokus-Mechanismus wie eine Blackbox fungiert und unvorhersehbare Änderungen vornehmen kann, wurde bei der Wahl der Kamera auf diese Funktion verzichtet. Ein fixierter Fokus hat auch den Vorteil, dass Ergebnisse reproduzierbar bleiben.

2.2.3 Objekterkennung mittels KI

Eine der größten Herausforderungen im kommerziellen KI-Sektor, Robotics- und Automatisierungsbereich, ist das Image Processing. Im Bereich des autonomen Fahrens stehen vorwiegend die Aufgaben Object Prediction, Object Tracking und Object Segmentation im Vordergrund. Die primitivste Aufgabe ist dabei Object Prediction. Hier beschränkt sich die Aufgabe des Maschinenmodells lediglich auf eine boolesche Aussage, ob sich spezifische Assets im Bild befinden und bestimmt deren relative Koordinaten und Bounding Boxen. Diese Form der Inferenz ist besonders performant und eignet sich optimal für Echtzeitanwendungen. Object Tracking ist eine Erweiterung dieses Aufgabenbereichs. Erkannte Objekte erhalten eine Unique ID und werden bildübergreifend verfolgt. Dadurch werden Bewegungsmuster erkannt und können analysiert werden. Object Tracking findet beispielsweise im aktiven Straßenverkehr seinen Nutzen, um das Fahrverhalten anderer Verkehrsteilnehmer zu verfolgen. Mit dieser Information ist es möglich, Präventivmaßnahmen zu implementieren oder auch Sicherheitsmanöver zu gestalten, weil das autonome System besser auf sein Umfeld reagieren kann.

Segmentierung ist eine detailliertere Form der Object Prediction. Dabei werden zusätzlich zu den Koordinaten und Bounding Boxen Masken berechnet. Masken in diesem Kontext sind eine Art Schablone bzw. Ausschnitt der identifizierten Assets. Das Besondere dabei ist, dass diese Masken während der Inferenz auf Pixelebene auf Nachbarschaftsrelationen geprüft werden. Dadurch lassen sich potenziell sehr akkurat Formen bestimmen, welche essenziell für die Spurverfolgung oder das Ermitteln der Straßenführung sind.

Um den Rahmen der Masterarbeit einzuschränken, wurde entschieden, sich auf die Spurverfolgung und Hinderniserkennung zu fokussieren. Außerdem existieren bereits ausgereifte Maschinenmodelle und Frameworks, wie in Abschnitt 1.2 beschrieben.

2.2.3.1 Wahl des Maschinenmodells

Im Computer Vision-Bereich haben sich hauptsächlich Convolutional Neural Networks (kurz CNN) als Architektur für Maschinenmodelle durchgesetzt. Bilder und Videos haben eine enorme Datendimension, weil jedes Pixel in einem Bild als ein Feature interpretiert wird. Das hat negative Auswirkungen auf die Effizienz und den Berechnungsaufwand. Der Vorteil bei CNNs ist das Downampling. Nachdem im Convolutional Layer durch einen Kernel eine Feature Map extrahiert wurde, wird im Pooling Layer die Dimension der Feature Map reduziert. Hierfür führt ein Kernel, ein Average oder Maximum Pooling durch, welches mehrere Features in einer Region zu einem einzelnen Feature aggregiert. Dadurch wird die Auflösung für konsekutive Layer reduziert, weshalb sich der Berechnungsaufwand ebenfalls sinkt. Wie bereits im Abschnitt 1.3.2.5 erwähnt wurde, kann dies zu einer Reduktion der Bildgröße von bis zu 75 % bei einem 2x2 Pooling Kernel pro Pooling Layer führen.

Für eine konkrete Übersicht über Trends und den aktuellen State of the Art im Open Source-Bereich, eignen sich Plattformen wie GitHub[37] und Roboflow.[38] Wohingegen GitHub eine allgemeine Anlaufstelle für Open Source Software ist, steht bei Roboflow das Maschinelle Lernen, speziell Computer Vision, im Vordergrund. Zu den beliebtesten Modellen mit Echtzeitfähigkeit gehören Fast Segment Anything[2] und YOLOv8.[10]



Abb. 2.6: Vergleich von Box-Prompt und Point-Prompt im FastSAM Framework

Quelle: <https://github.com/CASIA-IVA-Lab/FastSAM/blob/main/assets/Overview.png>, zuletzt besucht 12.09.2024

Jedoch unterscheiden sich die Modelle in ihrer Aufgabenstruktur. FastSAM ist darauf ausgelegt, ein Bild in alle vorhandenen Segmente zu unterteilen. Dies kann beim autonomen Fahren besonders nützlich sein. Bei Aufgaben wie der Hinderniserkennung, der Erkennung der Verkehrsführung und Verkehrslage und Spurassistenz bietet FastSAM eine ganzheitliche Lösung. Zwar sind unterschiedliche Aufgaben, wie Point-Prompt oder Box-Prompt, vorhanden, aber diese eignen sich für ein autonomes System nicht. Obwohl man eine statische Bounding Box in der Region of Interest (kurz ROI) platziert, wird in der gesamten

Bounding Box segmentiert. In Abbildung 2.6 werden jeweils beide Tasks illustriert. Diese Einschränkung erschwert Optimierungen und die Spezialisierung von Aufgaben anhand des Modells.



Abb. 2.7: Segmentierung mit YOLO[1]



Abb. 2.8: Segmentierung mit FastSAM[2]

Außerdem besteht das Problem, dass das Modell für FastSAM größer ausfällt. Mit 68 Millionen Parametern ist das Modell vergleichsweise zu YOLOv8 größer. Zusätzlich werden im FastSAM Repository kleinere Modelle nicht angeboten, was die Integration auf einem Jetson Nano erschwert. Dieser besitzt nur 4 GB RAM und laut Angaben verwendet FastSAM ca. 2,6 GB RAM während der Inferenz.[2] FastSAM benutzt zwei sequenzielle Stufen für die Segmentierungsaufgabe, wovon eine Stufe auf YOLO basiert. Das legt nahe, dass YOLO eine bessere Wahl für spezialisierte Anwendungen ist. Im YOLO Repository von ULTRALYTICS bieten die Entwickler mehrere Modellgrößen an. Das hat den essenziellen Vorteil, dass man Präzision und Performance gegeneinander abwiegen kann.

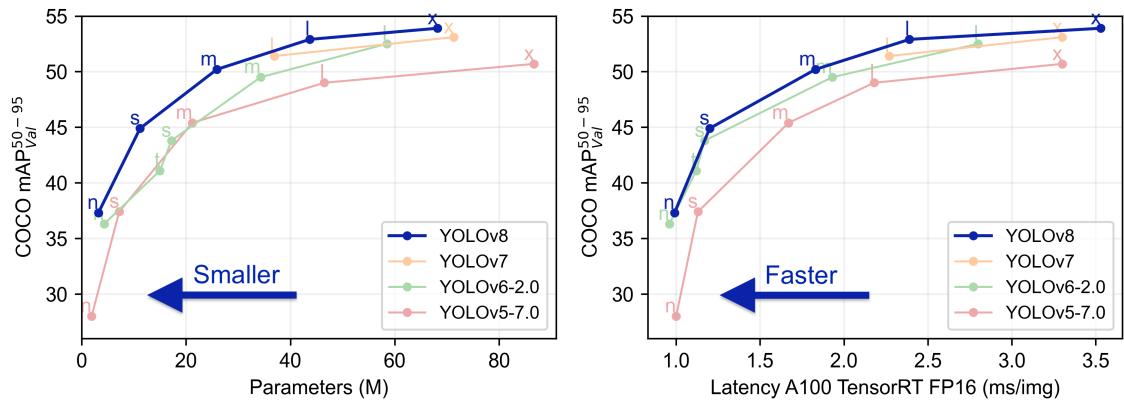


Abb. 2.9: Darstellung der mean Average Precision und Latenz der unterschiedlichen Modellgrößen
Quelle: <https://raw.githubusercontent.com/ultralytics/assets/main/yolov8/yolo-comparison-plots.png>, zuletzt besucht 10.09.2024

In Abbildung 2.9 ist diese Gegenüberstellung illustriert. Was hier in beiden Graphen besonders auffällt, sind die sinkenden Erträge bei steigenden Modellgrößen. Obwohl das größte

Argument	Funktion
source: str	Dateipfad zu einem Order, Bild oder Video. Datenstream mittels einer Kamera möglich.
conf: float	Minimaler Konfidenzschwellenwert, der erreicht werden muss für eine gültige Detektion
iou: float	Schwellwert des Überlappungskoeffizienten. Je niedriger der Wert, umso weniger dürfen je 2 Detektionen sich überlappen.
half: bool	Half-Precision (FP16). Halbiert die Präzision zugunsten der Beschleunigung

Tabelle 2.1: Gekürzte Übersicht der relevanten Inferenzparameter

Modell YOLOv8x-seg die höchste durchschnittliche Präzision von 53,4 % bietet, kostet diese sowohl um Magnituden mehr Systemressourcen als auch Berechnungszeit. Mit 71,8 Millionen Parametern und einer Modellgröße von 140,7 MB ist es wahrscheinlich nicht möglich, dieses Modell zu verwenden. Um eine genauere Analyse der Modellgrößen zu gewährleisten, ist eine technische Realisierung nötig. In Kapitel 2.3 wird dieses Thema näher beleuchtet. Zusätzlich bietet YOLOv8 diverse Inferenzargumente an. Inferenzargumente sind Argumente bzw. Funktionsparameter, die es ermöglichen, das Verhalten des Maschinenmodells zu modifizieren. Auch der Inferenzprozess lässt sich hierdurch bearbeiten. In Tabelle 2.1 sind die wichtigsten Argumente des YOLO-Modells gelistet.

Zusammenfassend ist somit YOLOv8 sowohl eine präzise als auch flexible Lösung für die gewünschte Anwendung. Angesichts dessen wird sich die weitere Ausarbeitung und Integration auf den YOLO-Softwarestack fokussieren.

2.2.3.2 Trainingsdaten

Sofern keine vortrainierten KI-Modelle für eine gegebene Aufgabe benutzt werden, muss ein entsprechender Trainingsdatensatz erstellt werden. Je nach Aufgabe des Modells, haben Datensätze unterschiedliche Kompositionen. Der bekannteste Ansatz bei Object Detection die Verwendung von Bounding Boxen und Klassifizierungen.

Dabei werden zu klassifizierende Objekte in einem Bild mit einer Bounding Box erfasst und entsprechend annotiert. Die Koordinaten für die Bounding Box können je nach Formatierung des Datensatzes unterschiedlich abgespeichert werden. Wie bereits in Abschnitt 1.3.2.6 angeschnitten wurde, ist COCO im Computer Vision-Bereich ein weitverbreiteter Datensatz. Dieser besteht aus einer JSON-Datei und einem Unterordner der annotierten Bilder. Die JSON-Datei speichert sämtliche Metadaten bezüglich des Datensatzes. Für jedes annotierte Bild wird ein Eintrag mit den Bilddimensionen und den korrespondierenden Bounding Boxen gemacht. Hier ist wichtig anzumerken, dass unterschiedliche Com-

puter Vision-Aufgaben unterschiedlich viele Metadaten benötigen. Wohin beispielsweise Object Tracking nur mit Bounding Boxen auskommt, werden für Segmentierungsaufgaben zusätzlich Polygone benötigt. Dies ermöglicht es, komplexe Formen und Strukturen hervorzuheben. In diesem Aspekt tritt die Problematik auf, dass das YOLO-Framework ein hauseigenes Format verwendet.

```

1 {"id": 1234567,
2 "category_id": 1,
3 "iscrowd": 0,
4 "segmentation": [[35.0, 107.8, ..., 452.13, 133.22]],
5 "image_id": 326345,
6 "area": 2398.32710009,
7 "bbox": [35, 80, 475, 307]}
```

Listing 2.1: Beispiel für eine COCO JSON-Datei

Ein YOLO-Datensatz ist zum einen anders aufgebaut, zum anderen werden Metadaten in einem anderen Format abgespeichert. Im Root-Verzeichnis des YOLO-Datensatzes wird je ein Ordner für *images* und *labels* generiert.

	images	18.12.2023 15:31	Dateiordner
	labels	18.12.2023 15:31	Dateiordner

Abb. 2.10: Ordnerhierarchie des YOLO-Datensatzes

Anschließend wird im *labels*-Ordner für jedes Bild im Datensatz eine .txt-Datei erstellt. Dadurch erhält man eine bijektive Projektion, was dazu führt, dass jede .txt-Datei als ein Container für Annotationen interpretiert werden kann. Dabei entspricht eine Zeile einer Annotation.

```

1 2 0.49 0.76 0.54 0.71 0.68 0.7
2 0 0.86 0.78 0.81 0.71 0.68 0.71 0.54 0.71 0.49 0.77 0.86 0.78
3 0 0.13 0.79 0.33 0.72 0.33 0.63 0.15 0.59 0.00 0.59 0.13 0.79
4 0 0.64 0.69 0.81 0.70 0.81 0.67 0.73 0.67 0.59 0.65 0.64 0.69
5 1 0.36 0.65 0.05 0.14 0.36 0.65
```

Listing 2.2: Beispiel für eine YOLO .txt-Datei

Im Beispiel 2.2 fallen zwei Besonderheiten auf. Die Koordinaten werden als eine relative Größe gespeichert. Das hat den Vorteil, dass Bildgrößen unabhängig von bereits erfass-ten Metadaten hoch- oder herunterskaliert werden können, sofern das Seitenverhältnis konstant bleibt. Es ist kurz anzumerken, dass im COCO- und YOLO-Datensatz zwei auf-einanderfolgende Werte eine Koordinate erzeugen.

Die zweite Besonderheit ist die ID zu Beginn jeder Zeile. Diese ID dient zur Zuordnung von Annotationen zu den entsprechenden Klassen. Diese sind in einer *Yet Another Markup Language*-Datei spezifiziert. Die YAML-Datei dient als eine Konfigurationsdatei für den Trainingsprozess mit einem YOLO-Modell.

```
1 train: yolo/train/images
2 val: yolo/valid/images
3 test: yolo/test/images
4 nc: 3
5 names: [ 'meadow', 'obstacle', 'street_main' ]
6 workspace: yolovision
7 project: yolo_vision
8 version: 2
```

Listing 2.3: Beispiel für eine YAML-Datei

Wie man aus dem Beispiel 2.3 entnehmen kann, werden drei Ordnerpfade hinterlegt. Diese Methode wird Data Splitting genannt und ist ein integraler Teil des Trainingsprozesses. Um eine hohe Präzision zu erzielen, werden Datensätze in Subsets unterteilt. Es existieren insgesamt drei unterschiedliche Batches, welche in die Aufgabenbereiche Training, Validierung und Test aufgeteilt sind. 60 bis 80 % des gesamten Datensatzes werden für das Training allokiert. Die restlichen 20 bis 40 % werden zwischen Validierung und Test verteilt. Die Hauptaufgabe des Trainingsdatensatzes ist das Antrainieren und Klassifizieren der annotierten Daten. Hier läuft man jedoch in das Problem des Overfittings. Overfitting ist ein mathematisches Problem, das auftritt, wenn sich ein Modell zu sehr an vorgegebene Daten hält und somit die Qualität der Prädiktion reduziert. Im Anwendungsfall der Generalisierung ist dieser Effekt nicht gewünscht. Besonders im Fall des autonomen Fahrens kann Overfitting dazu führen, dass parallel laufende Straßen oder auch Bürgersteige falsch erkannt werden können. Um Overfitting entgegenzuwirken, benutzt man einen Validierungsdatensatz. Rein funktional verhält sich dieser ähnlich wie der Testdatensatz. Beide werden aus dem reinen Lernprozess ausgeschlossen und werden als eine Performanzmetrik verwendet. Was jedoch Validierung- und Testdatensätze unterscheidet, ist, dass die Ergebnisse über den Validierungsdatensatz während des Trainings verwendet werden, um Hyperparameter zu konfigurieren und die am besten abschneidenden Modelle zu bestimmen. Dadurch können inkrementelle Verbesserungen erzielt werden.

2.2.4 Steuer- und Reglereinheit

Zwar lassen sich mittels Objekterkennung bereits rudimentäre Fahr-, Spur- und Parkassistenten integrieren. Um jedoch eine höhere Autonomie zu bewerkstelligen, benötigt ein autonomes Fahrzeug Regelkreise und Steuerelemente. Diese können mit Sensorwerten, Be-

rechnungen oder physikalischen Modellen gekoppelt werden, um ein Fahrzeug in Echtzeit zu steuern. Im Fall dieser Ausarbeitung soll die Lenkerausrichtung mithilfe eines Regelkreises gesteuert werden.

2.2.4.1 PID-Regelkreis

Ein beliebter Regelkreis ist der **Proportional-Integral-Differential-Regler** (kurz **PID-Regler**). Dieser besteht aus drei Anteilen.[39]

Der **Proportional-Regler** berechnet die Stellgröße $y(t)$ abhängig von der Regeldifferenz des Soll- und Istwerts $e(t)$.

$$x(t) = K_p \cdot e(t) \quad (2.9)$$

K_p ist der lineare Proportionalitätsfaktor, welcher den Anteil der Regeldifferenz in der Stellgröße variieren kann. Der Vorteil dieses Reglers ist die Simplizität und die sofortige Reaktion der Stellgröße, weil diese nur abhängig von der momentanen Regeldifferenz ist. Ein großer Nachteil des **P**-Reglers ist die mögliche Übersteuerung und Oszillation der Steuergröße.

Der **Integral-Regler** im Gegensatz zum **P**-Regler kann Stellgrößen nicht sofort berechnen, genauer gesagt benötigt eine gewisse Vorlaufzeit. Die Integration der Regeldifferenz über die Zeit liefert bei diesem Regler die Stellgröße. Hierdurch lässt sich über das vergangene Verhalten der Eingangsgröße die Stellgröße modulieren.

$$y(t) = \frac{1}{T_N} \int_0^t e(\tau) d\tau \quad (2.10)$$

Die Nachstellzeit T_N wird verwendet, um sowohl die Gewichtung der Stellgröße einzustellen als auch den Gradienten zu bestimmen. $e(\tau)$ ist dabei das zeitliche Verhalten der Regelabweichung. Zwar ist der **I**-Regler ein sehr akkurate Regler, jedoch werden durch die Nachstellzeit die Echtzeitanforderungen aus Abschnitt 2.1.2 verletzt.

Der **Differential-Regler** kann nur in Konjunktion mit anderen Reglern verwendet werden, weil dieser nicht auf Regeldifferenzen reagiert. Stattdessen berechnet der **D**-Regler die Steigung der Änderung im System.

$$z(t) = T_V \frac{d}{dt} e(t) \quad (2.11)$$

T_V ist die Vorhaltzeit und ist, wie K_p , ein Anteilsfaktor, der bestimmt, wie groß der Einfluss des **D**-Reglers ausfallen soll. Mit der Vorhaltezeit wird vorgegeben, wie lange die berechnete Steigung angenommen werden soll. Der zusammengefasste Ausdruck für einen **PID-Regler** ist folglich:

$$r(t) = x(t) + y(t) + z(t) \quad (2.12)$$

$$= K_p \cdot e(t) + \frac{1}{T_N} \int_0^t e(\tau) d\tau + T_V \frac{d}{dt} e(t) \quad (2.13)$$

Prinzipiell eignet sich der PID-Regler als die Steuereinheit für autonome Fahrzeuge. Dieser wurde im Zuge voriger Arbeiten bereits implementiert. Jedoch hat sich während der Experimente und Auswertung herauskristallisiert, dass zu viele dynamische Größen, Variablen und externe parasitäre Einflüsse das autonome Fahrzeug oszillieren lassen und seine Trajektorie destabilisieren. In der Arbeit von Hunziker[19] wurden bereits Simulationen und erste Schritte zur Implementierung eines neuen Reglers eingeführt. Dieser ist der Model Predictive Controller.

2.2.4.2 Model Predictive Controller

Das Konzept des Model Predictive Controls (kurz MPC-Reglers) basiert auf der Simulation von Abläufen in physikalischen Modellen. Dabei werden, je nach Steuerungsaufgabe, Modelle erstellt, die versuchen, den realen Anwendungsfällen möglichst akkurat zu replizieren. Diese Herangehensweise erlaubt es, externe Sensordaten in Modelle einzuspeisen und Prädiktionen für Steuerelemente zu generieren.

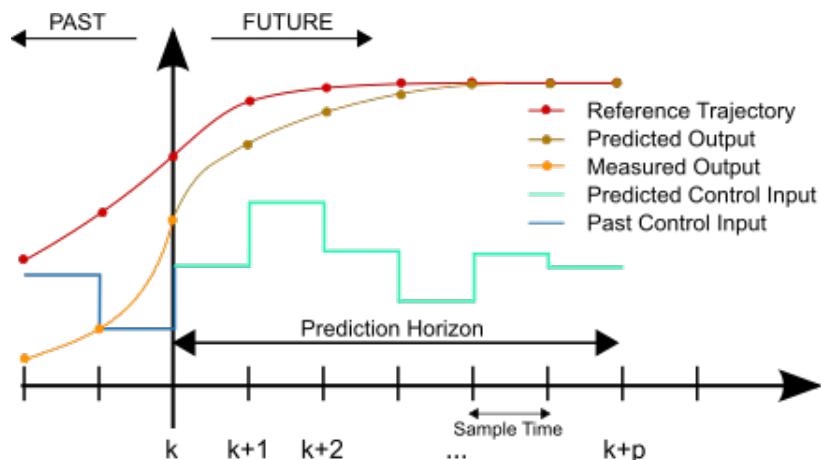


Abb. 2.11: Illustration der zeitdiskreten Verhältnisse der gemessenen, vorhergesagten und idealen Kurven eines MPC-Reglers

Quelle: https://upload.wikimedia.org/wikipedia/commons/1/11/MPC_scheme_basic.svg, zuletzt besucht 15.09.2024

Dabei muss ein Optimierungsproblem über die Kostenfunktionen aller relevanten Parameter formuliert werden. Es handelt sich beim regulären MPC-Regler um ein zeitdiskretes und lineares Gleichungssystem. Was den MPC-Regler besonders effektiv macht, ist die

Fähigkeit, vergangene Systemzustände und aktuelle Geschwindigkeitsänderungen, ähnlich wie der **ID**-Regler zu kontextualisieren. Es ist auch durch die Simulationen möglich, eine Referenzkurve und Prognosen für in der Zukunft liegende Systemzustände zu erstellen. Mit diesen Eigenschaften ist der MPC-Regler besonders resistent gegen Oszillation.

Für eine akkurate Simulation wird der dreidimensionale Raum, inklusive der Zeit, als weitere Dimension verwendet. Aufgrund der höheren Dimensionierung und Parametrisierung eines solchen Modells ist hier eine echtzeitfähige Simulation nur schwer möglich. Der funktionale Nutzen des MPC-Reglers ist jedoch in der Echtzeit verankert. Eine Abstrahierung ist dafür nötig. Die Bewegung von Straßenverkehrsmitteln lässt sich auf eine planare, zweidimensionale Ebene abstrahieren. Wie bereits in Abschnitt 1.3.4 erwähnt, hat die Pitch-Achse nur bedingt Einfluss auf die Straßenlage von Fahrrädern. Eine weitere Abstrahierung erreicht man, indem die Blickperspektive geändert wird. Wenn man die Kameraperspektive, wie es in Abschnitt 2.2.2 geschildert, beibehält, ist eine Vorwärtsbewegung nicht ersichtlich. Daher eignet sich eine Vogelperspektive auf die planare Ebene der Straßenführung.

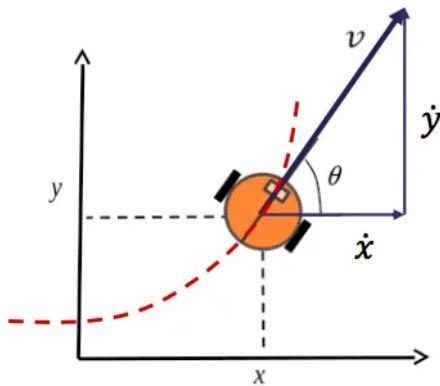


Abb. 2.12: Vogelperspektive der planaren Straßenebene

Quelle: <https://dingyan89.medium.com/simple-understanding-of-kinematic-bicycle-model-81cac6420357>, zuletzt besucht 10.09.2024

Ein effektives Modell hierfür ist das Einspurmodell. Das Einspurmodell wurde bereits 1940 von Rieckert und Schunck entwickelt.[40] Rieckert et al. haben dabei das Wirken von Kräften, wie der Zentrifugalkraft, Seitenwinde und parasitären Kräften, auf ein Fahrzeug formalisiert. Dabei wird angenommen, dass bei konstanter Geschwindigkeit und gegebener Fahrtrichtung, oder auch Kreisbahn, ein Einlenkwinkel bestimmt werden kann. Mit diesen Eigenschaften lassen sich bereits Kurvenverhalten simulieren. Wie in Abbildung 2.12 illustriert, wird die Fahrbahn als eine planare Ebene interpretiert und in ein kartesisches Koordinatensystem projiziert. Dadurch lässt sich das Verhalten und die Position eines jeden Fahrzeugs in einen Zustand parsen. Das Einspurmodell verwendet hierfür ein Quadrupel. Dieser ist definiert als: $[x, y, \Theta, \delta]$. Während x und y absolute Standortkoordinaten sind, handelt es sich bei Θ und δ um Richtungsvektoren. Θ beschreibt die aktuelle Bewegungs-

richtung des Fahrzeugs und gibt somit auch die Richtung des Geschwindigkeitsvektors an. δ ist die Winkeldifferenz zwischen der Lenkerausrichtung relativ zur Bewegungsrichtung Θ . Um mit dem Modell zu interagieren, wird ein Eingabetupel mit folgenden Eigenschaften definiert: $[v, \varphi]$. Mit v lässt sich die aktuelle Fahrtgeschwindigkeit im Modell festlegen. Während δ die aktuelle Lenkerausrichtung im Modell ist, gibt φ die neue Lenkerausrichtung für die neue Iteration des Modells an.

Um das Modell kinematisch modellieren zu können, benötigt man einen Schwerpunkt.

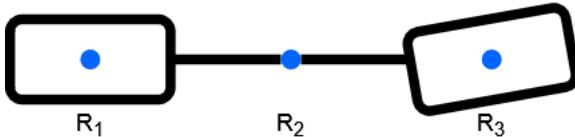


Abb. 2.13: Mögliche Schwerpunkte des Einspurmodells

Hierfür eignen sich die drei Punkte aus Abbildung 2.13 entlang der Längsachse. R_1 und R_3 sind jeweils die Kontaktpunkte der Hinter- und Vorderräder mit dem Boden. R_1 eignet sich sehr gut, weil das Fahrrad, wie in Abschnitt 1.3.4 beschrieben, einen Radnabenmotor besitzt. Fahrzeuge mit einem Hinterradantrieb haben besondere Eigenschaften in der Kurvenlage. Weil die Kraftübertragung rein über die Hinterachse geschieht, ist ein Ausbrechen des Hecks und Oversteer (dt. Übersteuern) zu beobachten.[41] Jedoch treten diese Probleme bei höheren Geschwindigkeiten und extremen Manövern auf. Die Geschwindigkeit des Fahrrads ist zwischen 10 und $20 \frac{km}{h}$ begrenzt. Des Weiteren ist eine maximale Einlenkung durch den BeagleBoneBlack gegeben, welche diese Szenarien vermeidet.

Für eine balancierte Simulation eignet sich R_2 . Hier befindet sich der Massenschwerpunkt. Nachdem ein Schwerpunkt gewählt wurde, lässt sich mithilfe eines Momentanpols das Problem visualisieren, geometrisch und trigonometrisch lösen.

Um das Einspurmodell berechnen zu können, müssen zunächst α , β , R und L bestimmt werden. In Abbildung 2.14 wurden bereits alle Korrelationen eingezeichnet. Wichtig ist anzumerken, dass H nicht die Länge des Fahrrads ist, sondern des Radstandes. Diese wurden bereits in Abbildung 2.13 illustriert und erläutert. Außerdem wird hier die Annahme vorausgesetzt, dass R_2 äquidistant zu R_1 und R_3 ist. Somit ergeben sich für $\vec{R_2R_1}$ und $\vec{R_2R_3}$ die Distanz $\frac{H}{2}$.

$$\gamma = 90 - \varphi \quad (2.14)$$

$$\alpha = 90 - \gamma \quad (2.15)$$

$$= 90 - (90 - \varphi) \quad (2.16)$$

$$\alpha = \varphi \quad (2.17)$$

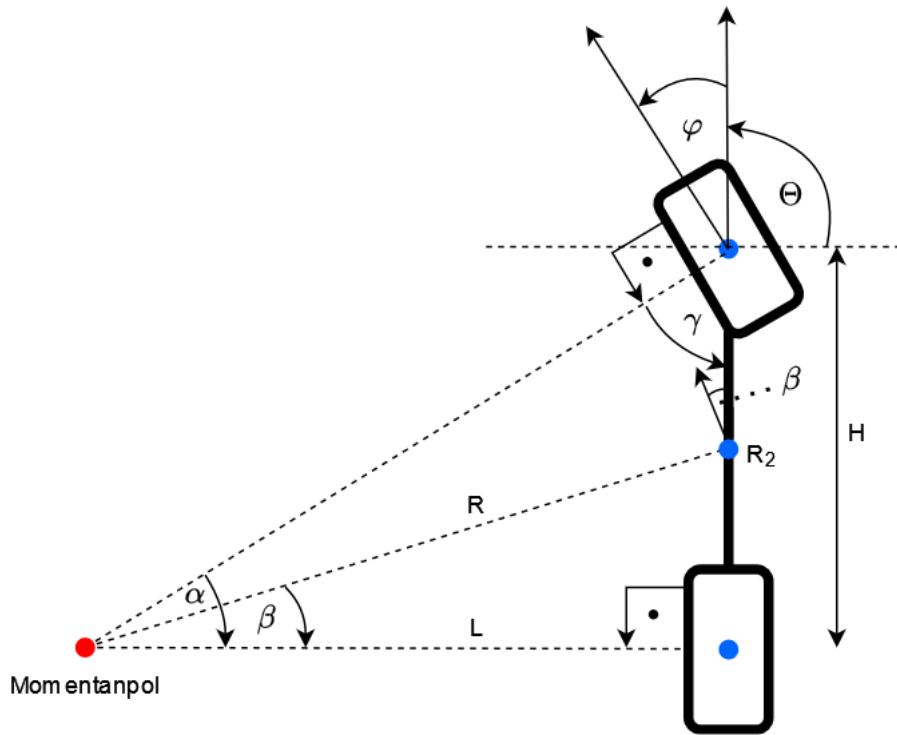


Abb. 2.14: Darstellung der geometrischen Komposition und des Momentanpols

Weil $\alpha = \varphi$ gilt, lässt sich mithilfe des neuen Einlenkwinkels die Winkelgeschwindigkeit ω um den Momentanpol berechnen. Jedoch muss noch der Radius bestimmt werden.

Der Radius R lässt sich ohne weitere Informationen nicht bestimmen. Hierfür sind die Größen L und β notwendig. Um die Länge L zu bestimmen, kann man $\tan(\varphi)$ verwenden.

$$\tan(\varphi) = \frac{H}{L}$$

$$L = \frac{H}{\tan(\varphi)} \quad (2.18)$$

Mit L lässt sich nun β bestimmen. β wird auch Schwimmwinkel genannt und ist die Kraft, die im Schwerpunkt des Fahrrads während einer Rotationsbewegung wirkt. Diese Größe gibt an, wie weit stark das Fahrrad auf seiner Kreisbahn rotiert und in welche Richtung sich das Fahrrad tatsächlich bewegt.

$$\tan(\beta) = \frac{\frac{H}{2}}{L}$$

$$\beta = \arctan\left(\frac{H}{2L}\right) \quad (2.19)$$

Für β lässt sich analog zu α die gleiche Projektion wie in 2.17 anwenden. Somit lässt sich der Radius R mit sin oder auch cos wie folgt berechnen:

$$R = \frac{L}{\cos(\beta)} \quad (2.20)$$

Abschließend kann Θ in Abhängigkeit von R formuliert werden. Weil Θ die neue Fahrtrichtung auf der beschriebenen Kreisbahn ist, kann hier angenommen werden, dass die Winkelgeschwindigkeit ω äquivalent zu Θ ist. Somit ergibt sich für Θ :

$$\begin{aligned} v &= \omega \cdot r \\ \omega &= \frac{v}{r} \\ \Theta &= \omega \\ \Theta &= \frac{v}{R} = \frac{v}{\frac{L}{\cos(\beta)}} = \frac{v \cdot \cos(\beta) \tan(\varphi)}{H} \end{aligned}$$

Das Einspurmodell kann noch mit weiteren Kenngrößen erweitert werden, welche jedoch für eine Echtzeitsimulation den Rahmen der Leistungsfähigkeit sprengen würde. Der Vollständigkeit halber werden hier noch weitere Attribute aufgelistet:

- Die Seitenkräfte an Kontaktpunkten der Reifen
- Die Schräglaufwinkel α_h und α_v
- Die Achsseitenkräfte S_h und S_v

Ohne Beschränkung der Allgemeinheit lässt sich der nächste Zustandsquadrupel des Modells mit folgenden Gleichungen berechnen:

$$\dot{x} = v \cdot \cos(\beta + \Theta) \quad (2.21)$$

$$\dot{y} = v \cdot \sin(\beta + \Theta) \quad (2.22)$$

$$\dot{\Theta} = \frac{v \cdot \cos(\beta) \tan(\varphi)}{H} \quad (2.23)$$

$$\dot{\delta} = \varphi \quad (2.24)$$

Somit ist das Modell und Fundament für eine Simulation gelegt. Die weitere Vorgehensweise im MPC-Regler sieht es vor, das Optimierungsproblem mit diesen Funktionen aufzustellen. Weil jedoch ein regulärer MPC-Regler ein zeitdiskretes und lineares Gleichungssystem unterliegt, müssen die Gleichungen 2.21, 2.22, 2.23 und 2.24 entsprechend angepasst werden.

$$x(i+1) = x(i) + \dot{x} \cdot \Delta t \quad (2.25)$$

$$y(i+1) = y(i) + \dot{y} \cdot \Delta t \quad (2.26)$$

$$\Theta(i+1) = \Theta(i) + \dot{\Theta} \cdot \Delta t \quad (2.27)$$

$$\delta(i+1) = \varphi(i) + \dot{\delta} \cdot \Delta t \quad (2.28)$$

Um die Kostenfunktion, Linearisierung und Gewichtung der einzelnen Parameter genauer nachzuvollziehen, wird empfohlen, die Arbeit von Che Kun Law et al.[42] zu lesen, weil weitere Erläuterungen den Rahmen dieser Arbeit übersteigen. Um das grundlegende Prinzip zu illustrieren, ist Abbildung 2.11 nützlich. Hier werden die einzelnen zeitdiskreten Samples des Systemzustands, die entsprechende Referenzkurve, die Prognosen und die Justierung der Steuergrößen eingezeichnet.

2.3 Technische Realisierung

Die Implementierung von mehrstufigen Prozessabläufen auf eingebetteten Systemen ist nicht trivial. In diesem Abschnitt wird dokumentiert, welche technischen Schritte für jede Komponente abgearbeitet wurden. Gerade, weil der Jetson Nano eine ARM-Prozessorarchitektur besitzt, können hier einige technische Komplikationen auftreten.

2.3.1 Einrichten des Jetson Nano

Der Jetson Nano ist ein älteres Modell in der Produktlinie, weshalb das bereitgestellte Betriebssystem veraltet ist. Dabei handelt es sich um eine modifizierte Ubuntu 18.04-Distribution, die bereits mit vorinstallierten Packages kommt, wie:

- JetPack 4.6.1
- Python 3.6.9
- pytorch 1.10
- CUDA, cuDNN und TensorRT

Diese Zusammensetzung ist jedoch nicht kompatibel für den Anwendungsfall dieser Arbeit. Komponenten des Softwarestacks wie das YOLOv8-Modell benötigen mindestens Python 3.8. Hierfür bietet Q-engineering[43] eine modifizierte Variante des Betriebssystems an. Das hat den Vorteil, dass die oben gelisteten Packages aktuellere Versionen verwendet werden können. Hierdurch lassen sich mit höheren PyTorch-Versionen modernere Hyperparameter einsetzen.

2.3 Technische Realisierung

Der Jetson Nano verwendet einen microSD-Kartenslot als Hauptspeicher. Moderne SD-Karten sind Flash-EEPROM-Speicher, welche geflasht werden muss. Dafür eignen sich Programme wie balenaEtcher. Damit lassen sich einfache Formatierungen des Dateisystems und auch ISO-Dateien flashen.

Hierfür eignet sich bei UNIX-Systemen entweder ein ZFS- oder ext4 File System. Beide File Systems haben ihre Vor- und Nachteile. ext4 ist ein erprobtes und ausgereiftes Dateisystem. Jedoch bietet ZFS eine Snapshot-Funktion und Rollback-Funktionen, die es ermöglichen, bei Korruption oder auch Austausch der zugrunde liegenden Hardware zu erleichtern. Um weitere Kompatibilitätsfragen und unvorhergesehene Probleme zu vermeiden, wird das File System ext4 verwendet. Nachdem das Betriebssystem aufgesetzt wurde, müssen auf dem Jetson Nano der Clang Compiler, torchvision und LibTorch aktualisiert werden. Um diese Schritte genauer nachzuvollziehen, ist die Referenz des Q-engineering Blogs[43] von Vorteil.

Nachdem alle Softwareupdates installiert wurden, ist es möglich, die Treiber für den e-con Systems CUNX-Kamerasensor zu installieren. Hierfür werden die Treiber vom Hersteller bereitgestellt und müssen manuell mithilfe einer Bash-Datei installiert werden. Für genauere Informationen dieses Prozesses ist die bereitgestellte Dokumentation im digitalen Anhang verfügbar, da hier noch weitere technische Zwischenschritte nötig sind. Außerdem befinden sich die Treiber ebenfalls im digitalen Anhang.

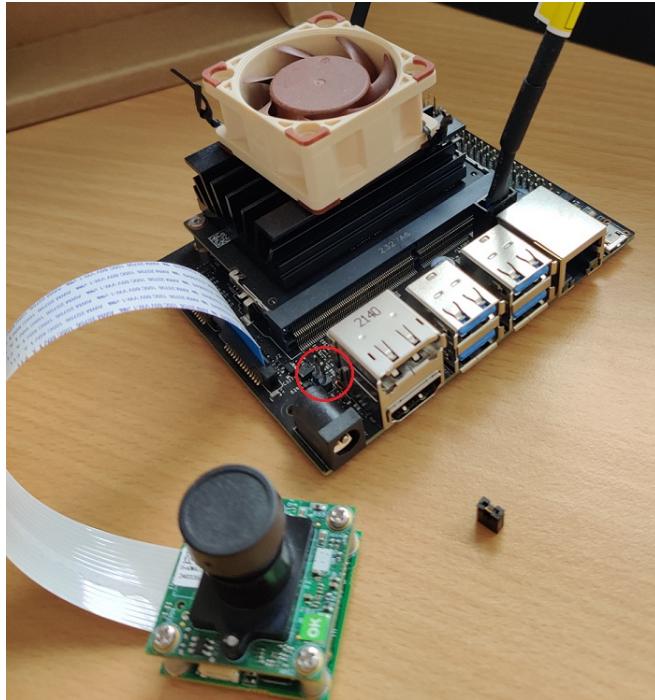


Abb. 2.15: Der zusammengebaute Jetson Nano

Der Jetson Nano besitzt verschiedene Energiemodi. Das System kann entweder über Micro-

USB oder über einen 5,5 x 2,1 mm Barrel Jack mit Strom versorgt werden. Um alle Funktionen des Kamerasensors und den Hochleistungsmodus für die Grafikeinheit zu verwenden, wird eine Versorgung über ein 5 V-, 4 A-Netzteil empfohlen. Hierfür müssen die in Abbildung 2.15 in Rot eingekreisten, Pins J48 mit einem Jumper überbrückt werden.

Bevor man mit dem programmiertechnischen Schritt beginnt, ist es ratsam, die Code Base in eine Versionsverwaltung einzubinden. Hierfür wird GitHub genutzt. Dieser Schritt hat mehrere Vorteile. Zum einen dient das GitHub Repository als eine Backup-Sicherung. Des Weiteren können Continuos Integration/Continuos Deployment-Strategien (kurz CI/CD) verwendet werden. Somit lässt sich die Entwicklungsumgebung separat vom Jetson Nano einrichten. Das erlaubt es, die Softwarereprogrammierung und auch das Trainieren des YOLO-Modells auf einem performanteren System durchzuführen. Um ein Modell über die CPU zu trainieren, ist ein Arbeitsspeicher von über 12 GB empfohlen.[44] Um das entwickelte System zu testen, lässt sich der Jetson Nano mithilfe von git-Command Line Interface (kurz CLI) synchronisieren. Falls in diesem Schritt Fehler auftreten oder eine Feature-Integration fehlschlägt, erlaubt git-cli ebenfalls Rollbacks. Ein Rollback setzt die Code Base auf eine vorige stabile Version zurück.

Relevante Git-Befehle	Funktion
git init	initialize an existing directory as a Git repository
git clone [url]	retrieve an entire repository from a hosted location via URL
git add [file]	add a file as it looks now to your next commit (stage)
git commit -m “[msg]”	commit your staged content as a new commit snapshot
git branch	list your branches
git branch [branch]	create a new branch at the current commit
git checkout [branch]	switch to another branch. Add it to the working directory
git merge [branch]	merge the specified branch’s history into the current one
git fetch [alias]	fetch down all the branches from that Git remote
git push [branch]	Transmit local branch commits to the remote repository
git pull	fetch and merge commits from the tracking remote branch

Tabelle 2.2: Liste aller relevanten Git-Befehle[4]

In Tabelle 2.2 sind die notwendigen Git-Befehle gelistet, die für eine CI/CD-Integration verwendet werden. Dabei sind *git fetch* und *git checkout [branch]* die zwei Befehle, die auf dem Zielsystem verwendet werden. Weil der Jetson Nano auf ARM basiert, werden hier zwei Branches verwendet. Dadurch lassen sich Systemdifferenzen separieren, die auftreten, weil das Entwicklungssystem ein Windowsgerät mit einer x86-Architektur ist.

2.3.2 Trainingsprozess und Datensatz

Die Programmierung der Pipeline ist unterteilt in zwei Abschnitte. Zum einen muss der Programmierbaustein zusammengesetzt werden, um das YOLOv8-Modell zu trainieren und gegebenenfalls Parameter für den Trainingsprozess zu setzen. Außerdem ist in dem Schritt wichtig, die richtige Modellgröße zu wählen. Zum Anderen muss eine Echtzeitanwendung entwickelt werden, die es ermöglicht, Resultate der Segmentierung mit einem MPC-Regler zu koppeln, um Stellgrößen zu berechnen.

Die Verwendung von Arbeitsspeicher während der Inferenz relativ zur Modellgröße ist eine nicht lineare, aber korrelierende Funktion. Tendenziell kann gesagt werden, dass je mehr Parameter ein Modell besitzt, umso mehr Arbeitsspeicher benötigt wird. Zudem steigt der Berechnungsaufwand und entsprechend die Latenz. Nach Jochen Glenn et al.[45] und Abbildung 2.9 eignen sich die Größen Medium und Small des YOLOv8-Modells am besten. Im direkten Vergleich der beiden Modelle fällt auf, dass die mAP für Small 36,8 % und für Medium 40,8 % beträgt. Zusätzlich ist im Small-Modell eine Inferenzdauer von 155,7 ms und für das Medium-Modell eine Inferenzdauer von 317,0 ms angegeben.[46] Mit einer Differenz der mAP von nur 4 % und der Zeitdifferenz von 161,3 ms eignet sich das Small-Medium wesentlich besser und gewährt einen Compute Overhead im Bezug der Arbeitsspeicherallokation.



Abb. 2.16: Beispiel für ein annotiertes Sample mit Roboflow[1]

Der erste Schritt besteht darin, einen Datensatz zu erstellen. Da zu diesem Zeitpunkt der Kamerasensor nicht ausgeliefert ist, kann hier eine Videoaufnahme einer Fahrradfahrt mit einer vergleichbaren Kameraperspektive, Szenerie und Auflösung verwendet werden.[1] Mit

dem Softwarepaket FFmpeg lassen sich diverse Operationen auf Videos ausführen. Um signifikante Samples zu finden, sollten aus unterschiedlichen Szenen Bilder entnommen werden.

```
1 ffmpeg -i test_video.avi -vf fps=1/60 img%03d.jpg
```

Listing 2.4: Ein Beispiel für ein Sampling im Sekundentakt

Nachdem die Samples erstellt wurden, können diese zur Annotation mit Roboflow[38] hochgeladen werden. Für den Annotationsprozess werden insgesamt 3 Klassen annotiert. Obstacle, street_main und meadow decken dabei die grundlegenden Fundamente ab. Mit der Klasse Obstacle sollen andere Fahrradfahrer und Passanten erkannt werden. Diese Klasse soll verwendet werden, um das Fahrrad zum Stillstand zu bringen. Die Klassen street_main und meadow sollen zum Erkennen der Straßenführung dienen. Während des Segmentierungsprozesses werden zu erkannten Objekten Masken erstellt, welche für spätere Schritte von Relevanz sind. Die Maske von street_main spielt dabei eine besondere Rolle in der Erkennung der Fahrtrichtung.

In Abbildung 2.16 fällt auf, dass das Bild in seinen Proportionen leicht verzerrt ist. Das YOLOv8-Modell verwendet eine Standardauflösung von 640x640 für die Bildeingabe. Zwar lässt sich diese Einstellung mit dem Parameter *imgsize* umgehen, jedoch hat eine Vergrößerung negative Auswirkungen auf die Inferenzdauer. Deshalb ist es notwendig, im Preprocessing das Bild auf die entsprechende Auflösung zu reduzieren. Nachdem die Samples das Preprocessing durchlaufen sind, können die entsprechenden Polygone eingezeichnet und annotiert werden. Im Anschluss wird, wie in Abschnitt 2.2.3.2 erläutert, ein Data Split eingestellt.

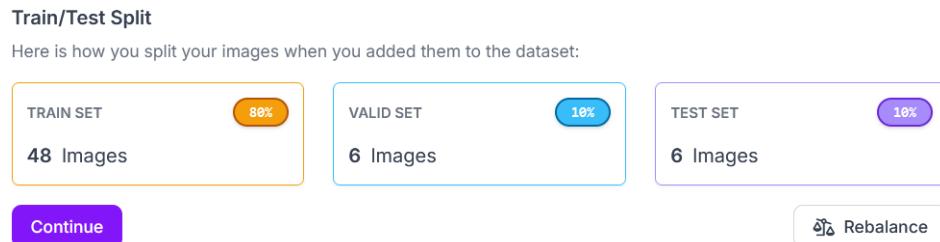


Abb. 2.17: Generierung des Datensatzes mit einem 80:10:10 Data Split

Der Datensatz besteht aus insgesamt 60 Bildern und dient als ein Proof of Concept. Es wird empfohlen, den Testsplit am größten zu halten, weil dieser die größte Auswirkung auf die Präzision des Modells hat. Somit maximiert man den Lerneffekt, ohne die Validierung und das Testen auszuschließen. Deshalb wurde wie in Abbildung 2.17 ein Train-/Validation-/Test-Split von 80/10/10 gewählt.

Zum Abschluss kann der Datensatz heruntergeladen und der Trainingsprozess begonnen werden. Um den Trainingsprozess zu verkürzen, wurden 300 Epochen eingestellt. Nach

Glenn et al.[10] wird für das YOLOv8-Modell eine Epochenspanne von 1000 bis 1100 empfohlen. Mit der bereitgestellten Hardware dauert jedoch der Trainingsdurchlauf in dieser Epochenspanne aufwärts von 12 Stunden. Weil es sich aber bei der Objekterkennung um simple Formen handelt, und das YOLOv8-Modell bereits über den COCO2017-Datensatz vortrainiert ist, können hier Abstriche gemacht werden.

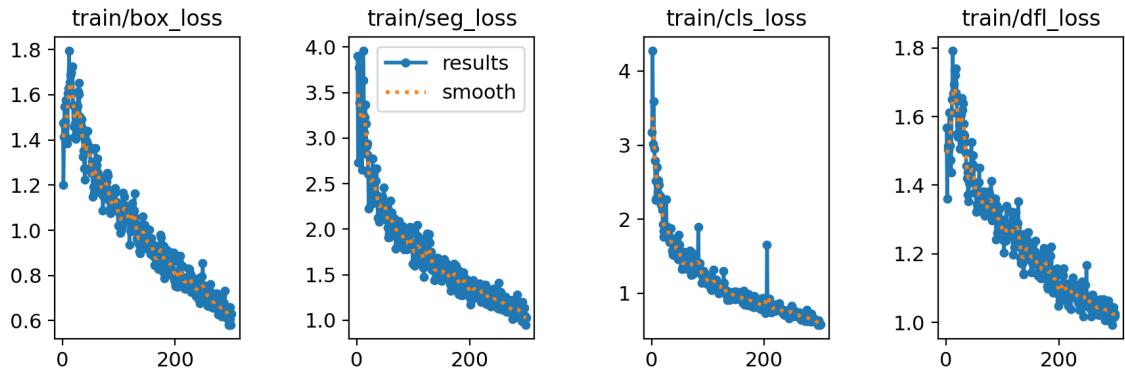


Abb. 2.18: Die Verlustfunktionen des YOLOv8s-seg-Modells

Simple Formen haben den Vorteil, dass sie bereits in frühen Layern erkannt werden. Die Hauptaufgabe des Modells ist, die Straße und entsprechend die Straßenkanten zu erkennen. Hier sollte eine geringe Epochenzahl wie 300 reichen, um den Proof of Concept zu erfüllen. Dies erkennt man auch in Abbildung 2.18. Hier erkennt man, dass alle Verlustfunktionen bereits beginnen zu konvergieren. *seg_loss* und *cls_loss* sind in diesem Anwendungsfällen die wichtigsten Metriken. Hier erkennt man, dass das Modell bereits in der Lage ist, die Klassifizierung von Seitenrändern, Straßen und Menschen zu differenzieren. Außerdem scheint die Segmentierungsmaße sich hinreichend adaptiert zu haben.

2.3.3 Aufbau der Inferenzpipeline

Nachdem das Training des Modells abgeschlossen ist, muss das Modell in eine Pipeline eingebunden werden. Dafür kann mit der Verwendung von Python3 und Open CV die erste Komponente initialisiert werden. Diese ist der Kamerasensor. Open CV erlaubt es, den erzeugten Datenstrom von Kamerasensoren in Echtzeit einzubinden. Dieser Endpunkt wird als Eingabe für das Modell verwendet.

Im nächsten Schritt muss das Modell initialisiert und konfiguriert werden. Der Jetson Nano hat eine auf CUDA-basierte Grafikeinheit. Das hat den Vorteil, dass TensorRT verwendet werden kann. TensorRT ist ein Deep Learning Framework von NVIDIA, welches in der Lage ist, PyTorch und TensorFlow-Modelle in einer optimierten Laufzeitumgebung inferieren zu lassen. Dafür muss das Modell exportiert werden. Während des Exports ist es möglich, weitere Optimierungen vorzunehmen. Hier lässt sich eine automatische INT8-Quantisierung durchführen. Dabei werden die einzelnen Layer des trainierten Modells

2.3 Technische Realisierung

kalibriert und überprüft, ob eine Beschleunigung feststellbar ist, wenn man den Präzisionsstypen zu INT8 umstellt. Wenn dies nicht der Fall ist, wird je nach Performancemetrik entweder FP32 oder FP16 für die Kernels verwendet. Dieser Optimierungsschritt kommt zum Teil auf Kosten der mAP[47]. Hier kann eine Beschleunigung der Inferenz von 100 bis 300 % erwartet werden, mit einem Rückgang der mAP von ca. 5 %. Hier kann auch ein Kompromiss mit Half Precision (FP16) eingegangen werden. Dabei kann eine Beschleunigung von ca. 100 % bei gleichbleibender mAP erwartet werden.

Parameter	Funktion
source	Datenstrom des Kamerasensors
device	CPU oder CUDA
conf	0.75
class	[1, 2]
iou	0.1

Tabelle 2.3: Die verwendeten Parameter für das YOLO-Modell

Nachdem das Modell exportiert wurde, kann das Modell in den Arbeitsspeicher der Grafikeinheit geladen werden. Weil der Datenstrom des Kamerasensors direkt in den Arbeitsspeicher fließt, wird dadurch die Inferenz beschleunigt und der Speicherverbrauch reduziert. Anschließend können die Parameter des Modells eingestellt werden. Um schwache Konfidenzen zu filtern, wird hier eine Mindestkonfidenz von 75 % eingestellt. Zum einen führt dies zu stabilerer Objekterkennung, zum anderen zu einer reduzierten Überlappung von gleichen oder ähnlichen Objekten.



Abb. 2.19: Eine ungefilterte Inferenz[3]

Abbildung 2.19 ist ein Beispiel für diese visuelle Unordnung. Um hier eine effektivere Ordnung innerhalb der identifizierten Objekte zu schaffen, eignet sich der *IoU*-Parameter. Intersection over Union ist im Machine Learning eine gängige Kenngröße, um die Schnitt-

menge zweier Objekte der gleichen Klasse zu berechnen. Diese wird auf 10 % Überschneidung beschränkt. Für jedes Objekt gleicher Klasse wird somit das Objekt mit der höchsten Konfidenz eingezeichnet und alle weiteren Objekte dieser Klasse verworfen, wenn die IoU-Schranke überschritten ist.



Abb. 2.20: Eine gefilterte Ausgabe der Inferenz[3]

Diese Filtereinstellungen haben Vor- und Nachteile. Wenn man Abbildungen 2.19 und 2.20 vergleicht, fällt auf, dass die Straßenerkennung sich auf die Erkennung mit höchster Konfidenz reduziert, aber auch der vorausfahrende Fahrradfahrer nicht mehr erkannt wurde. In Abbildung 2.19 sieht man, dass hier die Konfidenz für den Fahrradfahrer nur bei 45 % liegt. Der Grund dafür ist, dass der COCO-Datensatz zwar 3401 Bilder von Fahrrädern enthält, aber kaum Bilder von Fahrradfahrern im Straßenverkehr besitzt. Außerdem wurden im Trainingsdatensatz aus 2.3.2 Hindernisse als eine Klasse interpretiert und sich Attribute mit Passanten, Hunden oder anderen Fahrzeugen teilen. Das führt zu einer erschwerten Erkennung.

Die Inferenz lässt sich weiter verbessern und beschleunigen. Mithilfe einer Region of Interest (kurz ROI), lässt sich der Fokus auf vordefinierte Regionen im Bild verlegen. Da in dieser Arbeit das autonome Navigieren im Vordergrund steht, lässt sich der ROI auf die untere Bildhälfte bzw. unterhalb des Horizonts begrenzen. Damit halbiert sich die Bildfläche. Zusätzlich lassen sich die vertikalen Seitenränder ebenfalls zuschneiden. Sowohl in experimentellen Durchläufen als auch durch die Heatmap der Bounding Boxen in Abbildung 2.21, hat sich herausgestellt, dass man links und rechts vom Bildrand jeweils 10 % des Bildes entfernen kann.

Dieser Prozess hat vor allem Einfluss auf die Präzision und Konfidenz der Inferenz. Wie in Abschnitt 2.2.2.1 und 2.3.2 bereits erörtert wurde, besitzt das YOLOv8-Modell eine Inputauflösung von 640x640. Dadurch, dass die Bildfläche in einem Preprocess in seinen Dimensionen verkleinert wird, muss das Bild deutlich schwächer herunterskaliert werden.

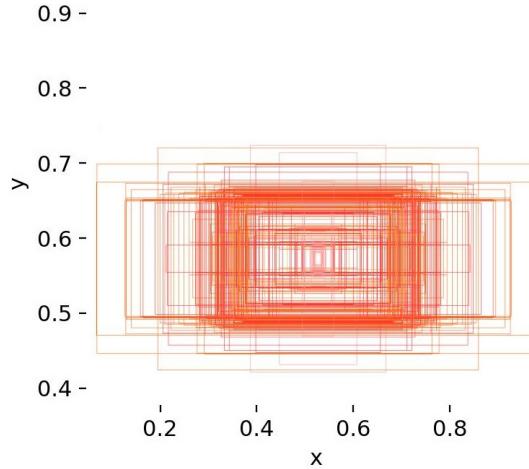


Abb. 2.21: Die aggregierten Bounding Boxen des Trainingsdatensatzes

Das hat den Vorteil, dass das Bild für die Inferenz einen höheren Informationsgehalt bei-hält. Die Auflösung des Datenstroms reduziert sich somit von 1920x1080 auf 1536x640. Damit reduziert sich die Pixelanzahl auf 47,4 %.

Eine weitere Preprocess-Operation ist das Verzerren des Bildes. Kameras bzw. Videoaufnahmen besitzen einen Fluchtpunkt. Der Fluchtpunkt ist der Punkt, in jenem alle parallelen Linien ins Zentrum des Bildes hineinfließen. Somit erscheinen parallele Linien in der realen Welt gekrümmmt und fließen in einen Fluchtpunkt während der Ablichtung. Dieser Effekt wird verstärkt, je weiter ein Objekt vom Kamerasensor entfernt ist. Bei der Straßenerkennung tritt dieses Phänomen ebenfalls auf, wie man in Abbildung 2.20 sieht. Mit einer perspektivischen Verzerrung lässt sich hier teilweise gegensteuern. Mit der Funktion `cv2.getPerspectiveTransform()` lässt sich eine ROI markieren, welche verzerrt werden soll. Dabei gibt es zwei Implementierungsmöglichkeiten. Die erste Option ist das Aufblähen der Pixel im hinteren, näher zum Fluchtpunkt liegenden, Bildbereich. Die zweite Option ist die Komprimierung der Pixel im vorderen, näher zum unteren Bildrand liegenden Bildbereich. Beide Optionen sind valide und haben Vor- und Nachteile.

Beim Aufblähen wird der Bildbereich vergrößert, mit Pixeln, die keinen Informationsgehalt besitzen. Um den Bildbereich zu füllen, werden deshalb nur vorhandene Informationen über einen breiten Raum verteilt. Beim Komprimieren wird der Bildbereich verkleinert, um den vorderen Bildbereich an den hinteren Bildbereich anzupassen. Hier werden zwar Informationen aus dem Bild entfernt, aber der Informationsgehalt der verbleibenden Pixel gleich bleibt. Die resultierenden Ergebnisse der Verzerrung sind in Abbildung 2.22 und Abbildung 2.23 dargestellt. Hier fällt auf, dass die Kompressionsmethode die leeren Räume mit schwarzen Pixeln padded. Dies könnte eventuell zu Problemen während der Inferenz führen. Deshalb werden beide Varianten implementiert und in Abschnitt 3 ausgewertet.

Wenn das Preprocessing abgeschlossen ist, kann mit der Inferenz begonnen werden. Die



Abb. 2.22: Ein mit Kompression verzerrtes Bild[3]



Abb. 2.23: Ein mit Aufblähen verzerrtes Bild[3]

Segmentierung liefert als Ausgabe ein *Results*-Objekt. In *Results* sind primär die erstellten Masken, Konfidenzen und Performanzmetriken von Relevanz. Eine Maske ist ein zweidimensionaler *ndarray*, welcher die Informationen der Segmentierung eines Objekts speichert. Dabei besitzt das *ndarray* dieselbe Größe wie das Bild und wird mit Nullen initialisiert. Anschließend wird während der Inferenz jeder klassifizierte Pixel in der korrespondierenden Stelle im *ndarray* auf Eins gesetzt.

Die erstellte Maske der Klasse *street_main* wird in den nächsten Schritten als Grundlage für die Straßenerkennung und zur Steuerung des MPC-Reglers verwendet. Weil eine Maske nicht mathematisch quantifizierbar für einen Regler ist, müssen hier weitere Postprocessing-Schritte durchgeführt werden. In der realen Welt lassen sich Straßenverläufe oftmals als eine Funktion darstellen. Analog lässt sich diese Herangehensweise auch hier verwenden. Eine effiziente Methode ist die Spline-Interpolation. Diese benötigt gut gewählte Stützpunkte und versucht eine Kurve zu approximieren, die durch alle gewählten Stützpunkte verläuft. Hier werden drei Sets mit je drei Punkten erstellt. Zwei davon repräsentieren den linken und rechten Straßenrand. Das dritte Set ist dabei in der Mitte der Straße platziert.

Die Pixelkoordinaten lassen sich aus der Maske ableiten. Weil die Maske die gleiche Dimension wie das Eingabebild besitzt, können mit arithmetischen Mitteln innerhalb des

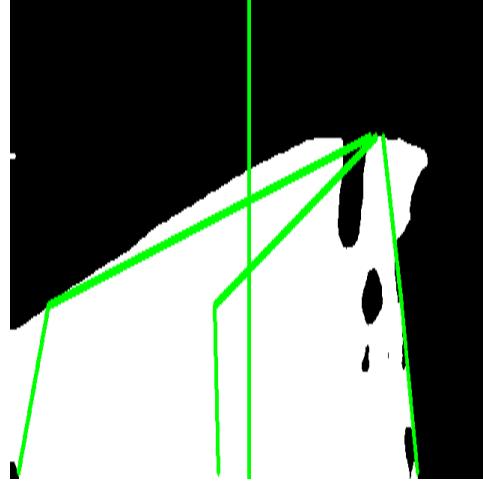


Abb. 2.24: Ein Beispiel der eingezeichneten Stützpunkte und -linien

ndarrays und der Verwendung der *nonzero*-Maske die Koordinaten der Stützpunkte ermittelt werden. Die *nonzero*-Maske hat die besondere Eigenschaft, dass die Maske nur aus den segmentierten Pixeln besteht. Damit lässt sich die äußere Form und Beschaffenheit der Maske abtasten. Die Funktionen *np.count_nonzero()* und *np.argmax()*, in Kombination mit Slicing-Operationen in Python, lassen sich die gewünschten Pixelkoordinaten berechnen.

Die Maske lässt sich in drei relevante Regionen unterteilen. Der vordere Bildbereich enthält Informationen darüber, wo sich das Fahrrad relativ zur Straßenbreite befindet. Der zentrale Bildbereich enthält Informationen zum unmittelbaren Straßenverlauf. Der letzte Bildbereich befindet sich in der Nähe des Fluchtpunkts und enthält Informationen, wo sich der Horizont befindet und wie sich der Straßenverlauf entwickelt. Die Platzierung der Stützpunkte innerhalb eines Sets orientiert sich an diesen Regionen und den Randfällen wie der Straßenseite oder der Straßenmitte. In Abbildung 2.24 sind diese Stützpunkte eingezeichnet. Hier kann man bereits erkennen, dass die Trajektorie der Straße erkennbar wird.

Mit diesen Informationen lässt sich der MPC-Regler steuern und einbinden. Im GitHub Repository *PythonRobotics* von Atsuhi Sakai[48] befindet sich ein MPC-Regler und visueller Simulator, der mit einem Einspurmodell ähnlich zum Abschnitt 2.2.4.2 implementiert ist. Dieser eignet sich, um die Lenkersteuerung und auch die Geschwindigkeit zu regulieren. Zur Auffrischung werden kurz die einzelnen Komponenten des MPC-Reglers zusammengefasst.

Der MPC-Regler verhält sich ähnlich wie ein endlicher deterministischer Automat. Der Zustand des MPC-Reglers lässt sich als Quadrupel $[x, y, \Theta, \delta]$ ausdrücken. Dabei sind x, y die Positionskoordinaten, Θ der Geschwindigkeitsvektor und δ die Winkeldifferenz zwischen dem Geschwindigkeitsvektor und der Lenkerausrichtung, also der Einlenkwinkel. Des Weiteren besitzt der MPC-Regler die zwei Eingabegrößen $[v, \varphi]$. v ist die aktuelle Ge-

schwindigkeit und φ der neue Einlenkwinkel. Diese Parameter werden verwendet, um im Einspurmodell die Simulation des Fahrverhaltens durchzuführen. Zusätzlich benötigt der MPC-Regler noch eine Referenz- bzw. Prognosekurve des Straßenverlaufs. Diese wird mit den aus dem Postprocessing entstandenen Pixelkoordinaten und der Spline-Interpolation entnommen. Folglich ist der MPC-Regler vollständig und kann in Betrieb genommen werden. Dafür muss zunächst die Simulation initialisiert werden. Die Klasse $State(x, y, yaw, v)$ wird dabei verwendet, um den oben beschriebenen Zustand darzustellen. Anschließend wird mit der ersten Eingabe des *lines*-Dictionary, welches die Stützpunkte enthält, die Referenzkurve berechnet. Hier wird aus *scipy.interpolate*-Modul die *CubicSpline()*-Klasse verwendet.

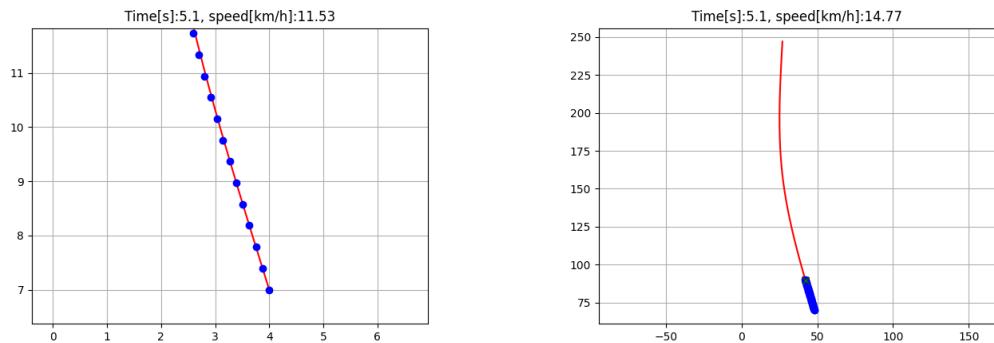


Abb. 2.25: Eine iterative Simulation des MPC-Reglers

Abb. 2.26: Die Originalgröße des Graphen

Somit ist der MPC-Regler vollständig integriert und kann die Steuerung des Fahrrads übernehmen. Für Abbildung 2.25 wurde eine Simulation mit einer Tickrate von 100 ms gewählt. In Rot ist die Spline-Interpolation der Kurve zu sehen. Die blauen Punkte sind die simulierten Zustände des Fahrrads und repräsentieren den gefahrenen Weg. Hier ist anzumerken, dass Abbildung 2.25 stark vergrößert wurde, damit die einzelnen Zustände des MPC-Reglers sichtbar sind. Dadurch erscheint die Referenzkurve linear, aber ist in Wirklichkeit gekrümmt. Dies wird in Abbildung 2.26 illustriert.

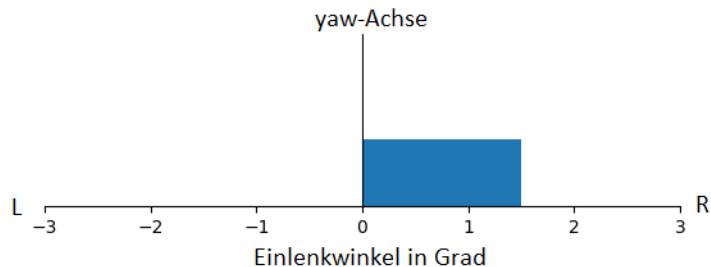


Abb. 2.27: Illustration der Einlenkgröße, die durch den MPC-Regler bestimmt wurde

In Abbildung 2.27 wird der Yaw-Winkel des Lenkers ausgegeben. Dieser wird in der nächs-

2.3 Technische Realisierung

ten Iteration verwendet. Außerdem ist diese Ausgabe auch die neue Steuergröße für das Fahrrad. Dies ist der letzte Schritt der Pipeline und ist vollständig funktionsfähig.

3 Ergebnisse und Diskussion

Mit dem Abschluss der theoretischen Konzeptionierung und der praktischen Umsetzung können die Ergebnisse zusammengetragen werden. Im Fokus der Arbeit standen drei Themen. Zum einen sollte die Durchführbarkeit und Integration einer Machine Learning-Pipeline geprüft werden, die es ermöglicht, Straßen, Passanten und Straßenverläufe zu inferieren. Zum anderen sollte nach einer Möglichkeit gesucht werden, diese Pipeline auf einer mobilen Plattform einzusetzen, die idealerweise diesen Prozess in Echtzeit bewältigt. Zuletzt sollte eine moderne Steuer- und Regeleinheit implementiert werden, die mit den erzeugten Ausgaben der Inferenz qualitative Stellgrößen berechnen kann.

3.1 Inferenzpipeline mit YOLO

Wie in jedem Machine Learning-Projekt zugrunde liegend, ist der Datensatz eine der wichtigsten Komponenten. Obwohl der Datensatz nur aus 60 Bildern besteht, ist es möglich, den Straßenverlauf mit guter Konfidenz zu verfolgen. In der Arbeit wurden zwei Testvideos verwendet. Video 1[1] diente zur Erhebung des Trainingsdatensatzes. Video 2[3] diente als ein Testversuch der vollständigen Pipeline. Hier hat sich eine durchschnittliche Konfidenz von 94,39 % über 100 Bilder ergeben. Dabei wurde eine randomisierte Szene im Video gewählt.

Weil die große Anzahl an Bildern das Ergebnis verfälschen und gewisse Bias kaschieren kann, sollte ebenfalls der Konfidenzverlauf über die Szene hinweg geprüft werden. In Abbildung 3.1 erkennt man beispielsweise, dass die Konfidenz zwischenzeitlich auf 84 % sinkt. Bei Betrachtung des Bildmaterials fällt auf, dass hier Verschmutzungen und Straßenschäden aufzufinden sind. Das lässt suggerieren, dass das Modell zwar in der Lage ist, die Straße mit hoher Konfidenz zu verfolgen, aber anfällig auf Szenenwechsel und unbekannte Beschaffenheiten der Straße ist. Dieser Test wurde ebenfalls in einer Szene mit einer anderen Belichtung getestet und ein ähnliches Verhalten konnte beobachtet werden.

Des Weiteren wurde getestet, inwiefern eine verzerrte Kameraperspektive sich auf die Straßenerkennung auswirkt. Hier hat sich herausgestellt, dass in der aufgeblähten Kameraperspektive die durchschnittliche Konfidenz auf 88,43 % gesunken ist.

Das erklärt sich dadurch, dass der Trainingsdatensatz mit einer unveränderten Kameraperspektive trainiert wurde, und die Verzerrung im Preprocess der Pipeline stattfindet.

3.1 Inferenzpipeline mit YOLO

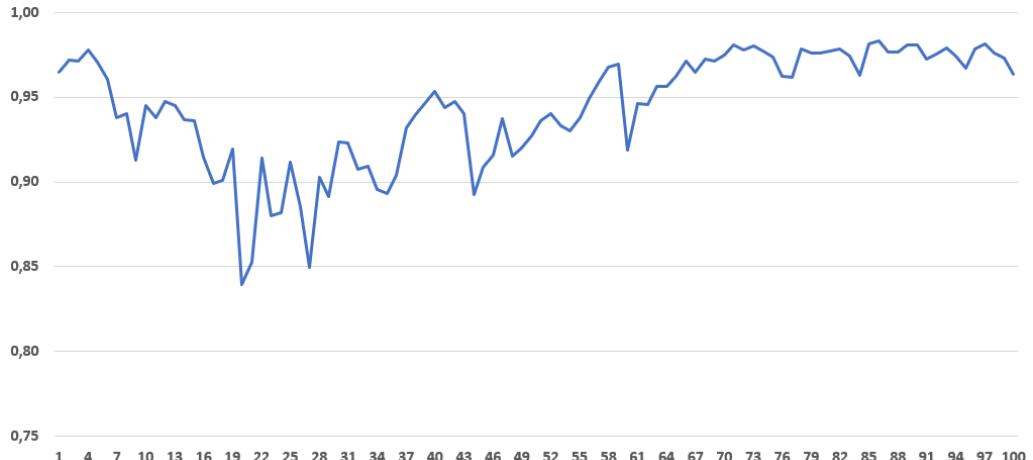


Abb. 3.1: Illustration des Konfidenzverlaufs des Objekts *street_main*

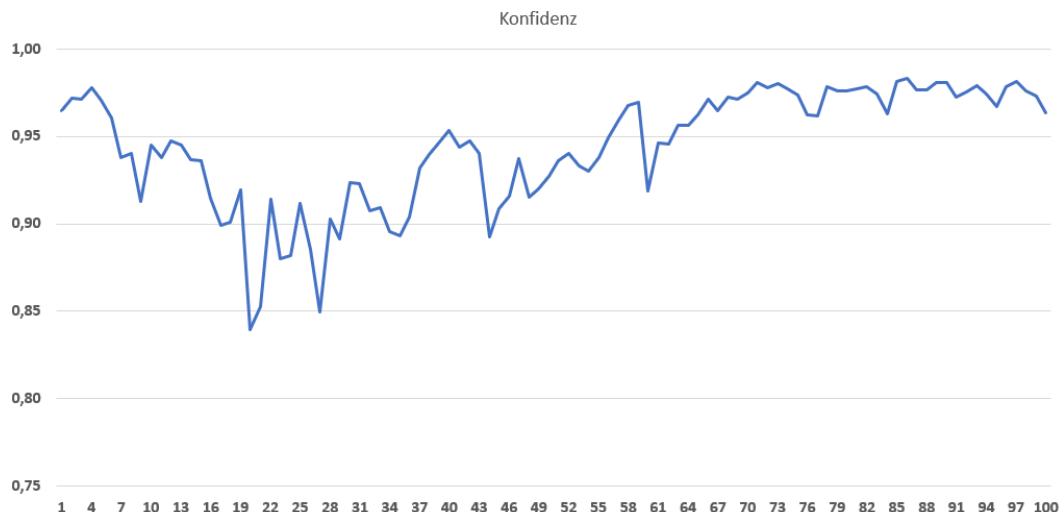


Abb. 3.2: Der Konfidenzverlauf mit einer aufgeblähten Kameraperspektive

Zwar hat man hier einen Verlust von 5,96 %, aber dieser Tradeoff zahlt sich aus. In der verzerrten Kameraperspektive ist die erstellte Maske eine akkurate Darstellung der realen Straßenführung. Sie wirkt dem Fluchtpunkt entgegen, was dazu führt, dass auch die Spline-Interpolation eine akkurate Repräsentation der Referenzkurve liefert. Hier ist außerdem anzumerken, dass die disproportionale Bildauflösung ausgenutzt werden kann. Wie in Abschnitt 2.3.3 erwähnt, hat das zugeschnittene Bild nach dem Preprocess eine Auflösung von 1536x640. Weil die YOLOv8-Eingabe auf 640x640 begrenzt ist, wird hier die Breite des Bildes um ein Vielfaches herunterskaliert. Dem wird mit der Aufblähung ebenfalls entgegengewirkt.

Die Verzerrung mittels Kompression hat keine ertragreichen Ergebnisse geliefert. Durch die extreme Veränderung des Szenenaufbaus, wie es in Abbildung 2.22 zu sehen ist, sinkt

3.1 Inferenzpipeline mit YOLO

die Konfidenz im Testdurchlauf konstant unter 25 %. Dieser Ansatz sollte eigentlich besser geeignet sein, weil der Informationsgehalt gewahrt wird. Um den Kompressionsansatz auszureifen, können weitere Optimierungen durchgeführt werden. Weil die ROI im unteren Bildrand sich verkleinert, können beispielsweise im Preprocess die Seitenränder des Bildes weiter zugeschnitten werden.



Abb. 3.3: Ein möglicher Zuschnitt für den Preprocess[3]

Ein möglicher Zuschnitt ist in Abbildung 3.3 zu sehen. Hier wurde die Bildbreite um weitere 20 % verkleinert, was zu einer Gesamtbreite von 1228 Pixeln führt. Dadurch wird der Effekt der Herunterskalierung weiter gedämpft. Zusätzlich sollte ebenfalls ein neuer Trainingsdatensatz erstellt werden, um dem YOLOv8-Modell die neue Bildkomposition anzutrainieren.

Ein weiterer Aspekt der Segmentierungspipeline ist die Echtzeitfähigkeit. Dafür wurden bei gleichbleibender Konfiguration der Pipeline mehrere Testdurchläufe durchgeführt und die Ergebnisse aggregiert.

Preprocess (ms)	Inference (ms)	Postprocess (ms)	Total (ms)
2,5	84,1	1,5	88,1
3,2	83,3	0,1	86,6
3,6	82,7	1,3	87,6
3,3	83,9	0,2	87,4

Hier wurde eine durchschnittliche Gesamtdauer des Prozesses von 87,4 ms festgestellt. Trotz der Verwendung der TensorRT-Engine konnte die obere Schranke von 30 ms für echtzeitkritische Systeme nicht erreicht werden. YOLOv8 und TensorRT bieten noch weitere Optimierungsschritte an. Beispielsweise kann mit NVIDIA Deepstream SDK das trainierte Modell weiter konfiguriert werden. Während des Exports zu einer TensorRT-Engine lassen sich Strides, Präzisionen, ROIs und Layer an die KI-Anwendung anpassen.

3.2 MPC-Regler

Die Frage, ob ein Regelkreis KI-gestützt Stellgrößen berechnen kann, wurde beantwortet. Die Stützpunkte, die innerhalb der Maske lokalisiert wurden, haben die nötige Signifikanz für die Spline-Interpolation. Dabei ist die Referenzkurve ein wichtiger Bestandteil des MPC-Reglers, weil sie sich unmittelbar auf die Kostenfunktionen auswirkt. Wie bereits in Abbildung 2.27 gezeigt, lässt sich der Lenkwinkel zur Laufzeit bestimmen und entsprechend auf das Fahrrad übertragen.

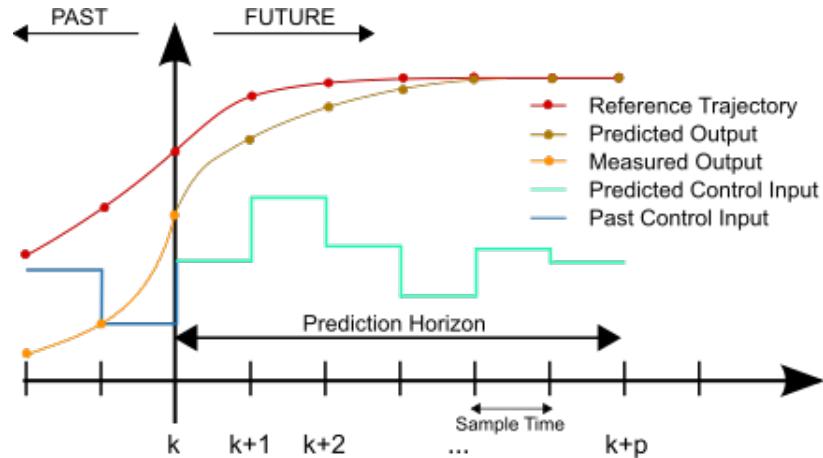


Abb. 3.4: Illustration der zeitdiskreten Verhältnisse der gemessenen, vorhergesagten und idealen Kurven eines MPC-Reglers

Quelle: https://upload.wikimedia.org/wikipedia/commons/1/11/MPC_scheme_basic.svg, zuletzt besucht 15.09.2024

Der MPC-Regler hat jedoch einen Nachteil. Die Simulation sollte idealerweise dauerhaft laufen und in gewissen Zeitabständen eine Stellgröße produzieren, so wie es in Abbildung 3.4 dargestellt ist. Das ist jedoch im Anwendungsfall des autonomen Fahrens nur begrenzt möglich. Mit jeder Iteration des Inferenzprozesses wird eine neue Referenzkurve berechnet. Hier bleiben nur zwei Optionen offen. Die erste Option ist, dass der MPC-Regler mit jeder Iteration neu initialisiert werden muss. Diese Herangehensweise ist akkurate, weil die Referenzkurve sich zu jeder Iteration an die reale Repräsentation der Straße hält. Die zweite Option ist, dass die Referenzkurven eine Verfallszeit erhalten und mehrere Iterationen überdauern. Mit dieser Variante umgeht man die Neuinitialisierungen, aber hat eventuell eine verfälschte bzw. veraltete Darstellung des aktuellen, realen Straßenverlaufs. Was diese Variante weiter bekräftigen könnte, ist die langsame Fahrgeschwindigkeit von $15 \frac{km}{h}$. Dadurch sind rapide Wechsel der Straßenverläufe minimiert.

Mit der aktuellen Konfiguration des MPC-Reglers ist es nur möglich, in der Mitte der Fahrbahn zu fahren. Diese Eigenschaft ist abhängig von der Wahl der Stützpunkte. Mit den Stützpunkten, wie in Abbildung 2.24, ist es aber möglich, die Referenzkurve zu ver-

schieben. Für eine Rechtsverschiebung der Referenzkurve kann man unter anderem das arithmetische Mittel der Pixelkoordinaten des rechten Straßenrands und der Straßenmitte berechnen.

3.3 Limitationen

Durch die Größe und die unterschiedlichen Disziplinen der Problemstellung sind Einschränkungen zustande gekommen. NVIDIA hat etwa den Softwaresupport für den Jetson Nano eingestellt, weswegen die aktuellen Versionen von JetPack, CUDAtoolkit, cuDNN, TensorRT und Python3 nicht zur Verfügung standen. Hier mussten alternative Wege gefunden werden, um den Softwarestack zu aktualisieren. Hier war es zumindest möglich, Updates für Ubuntu, Python3 und PyTorch zu finden. Vor allem war das Python3 Update essenziell, weil YOLOv8 eine Mindestanforderung von Python 3.8 besitzt.

Eine weitere Limitation sind die bereitgestellten Treiber von e-con Systems für den Kamererasensor. Diese sind nur über NVIDIA SDK Manager verfügbar. Deshalb mussten die Treiber im privaten E-Mail-Austausch zur Verfügung gestellt werden. Das hat den Entwicklungsprozess signifikant verlangsamt. Außerdem sind beim Jetson Nano im USB-Betrieb die Kameramodi eingeschränkt. Hier konnten nur 1920x1080 bei 15 FPS erreicht werden. Weil keine mobile Energiequelle und Netzteil für den Barrel Jack zur Verfügung standen, konnten die Tests in der Praxis nicht repliziert werden.

Den MPC-Regler zu implementieren, war eine Herausforderung. Weil der MPC-Regler eine Simulation und Kostenminimierung vereint, gab es hierzu keine vorgefertigten Bibliotheken oder Module. In MATLAB existieren MPC-Module und Einspurmodelle in Konjunktion, die es ermöglichen, in der MATLAB Runtime einen MPC-Regler zu simulieren. Jedoch existiert bisher noch keine MATLAB Runtime für ARM-basierte Rechnersysteme. Deshalb musste das Einspurmodell, die Spline-Interpolation und Teile der MPC-Reglung in Python implementiert werden, bis der MPC-Regler von PythonRobotics[48] eingebunden wurde.

3.4 Zukünftige Arbeit

Über den Zeitraum der Masterarbeit hat sich der State of the Art bereits weiterentwickelt. Zum Beispiel hat META im Juli 2024 das Modell SAM2[49] veröffentlicht. Hier wird behauptet, dass der Segmentierungsprozess um das Sechsfache beschleunigt wurde. Außerdem hat ULTRALYTICS im Oktober 2024 das Modell YOLO11[50] veröffentlicht. Diese Modelle könnten den Inferenzprozess beschleunigen und bessere Resultate liefern. Zusätzlich können die Aufgaben des Inferenzprozesses erweitert werden. Das aktuelle YOLOv8-Modell kann bereits Hindernisse erkennen, aber kann bisher auf diese Erkennungen nicht reagieren. Die Entwicklung eines Ausweichmanövers wäre eine dieser Aufgaben.

Im Bereich der eingebetteten Systeme gab es in diesem Zeitraum ebenfalls Fortschritte. Hier hat RaspberryPi ein AI-Kit für den RaspberryPi 5[51] veröffentlicht. In Anbetracht, dass die Arbeit von Hunziker[19] und der Sensor Fusion auf einem RaspberryPi basiert, könnte eine Kombination beider Arbeiten ein interessantes Forschungsthema sein.

Die Limitationen des Jetson Nano sind bisher nicht vollständig ausgeschöpft. Es ist möglich, noch einen zweiten Kamerasensor anzubringen, welcher aufschlussreiche Informationen über die Bildtiefe und Objektdistanzen liefern kann. Ein weiterer Ansatz wäre auch das Verbleiben bei einem Kamerasensor und das Bestimmen eines Gradienten für die Bildtiefe. Diese Informationen wären eine Bereicherung für die Präzision der Referenzkurve im MPC-Regler. Mit einem Gradienten könnte man die Referenzkurve normalisieren.

Eine Problematik, die in dieser Masterarbeit bestehen bleibt, ist die Rotation des Kamerasensors um die Roll-Achse. Aus zeitlichen Restriktionen wurde diese Thematik entfernt. Hier kann ein Gimbal verwendet werden, um die Videoaufnahme zu stabilisieren.

3.5 Abstract

Im Zuge dieser Arbeit sollte mit einer Computer Vision-Pipeline eine Steuereinheit für ein autonomes Fahrrad konstruiert werden. Diese Pipeline integriert ein KI-Modell zur Segmentierung von relevanten Objekten im Datenstrom einer Kamera. Aus diesen Informationen wurde mithilfe von Stützpunkten und Spline-Interpolation die Straßenführung quantifiziert. Das Ergebnis der Spline-Interpolation wurde mit einem Model Predictive Control-Regler verbunden. Hiermit ist man in der Lage ein Einspurmodell zu simulieren und entsprechende Stellgrößen zu berechnen. Die Pipeline sollte auf einem ARM-basierten, eingebetteten System ausführbar sein. Dazu wurde ein NVIDIA Jetson Nano verwendet. Dieser ist in der Lage, das Inferieren, das Simulieren und das Berechnen der Stellgrößen innerhalb von 100 ms abzuschließen. Das verwendete KI-Modell ist YOLOv8 von ULTRALYTICS, welches in eine TensorRT Runtime migriert wurde.

Literaturverzeichnis

- [1] Amazing Bike Rides, “Virtual bike ride to Presque Isle Park in the UP on Lake Superior.” <https://www.youtube.com/watch?v=bglPXcqRZTQ>, zuletzt besucht 08.10.2024, 2023.
- [2] X. Zhao, W. Ding, Y. An, Y. Du, T. Yu, M. Li, M. Tang, and J. Wang, “Fast segment anything,” 2023.
- [3] Amazing Bike Rides, “25 Minute Virtual Bike Ride — Whatcom Falls Park — Washington — Indoor Cycling Workout.” <https://www.youtube.com/watch?v=Co1Awk9py10>, zuletzt besucht 08.10.2024, 2021.
- [4] GitHubEducation, “GIT CHEAT SHEET.” <https://education.github.com/git-cheat-sheet-education.pdf>, zuletzt besucht 25.09.2024, 2024.
- [5] W. J. Dally, S. W. Keckler, and D. B. Kirk, “Evolution of the graphics processing unit (gpu),” *IEEE Micro*, vol. 41, no. 6, pp. 42–51, 2021.
- [6] Tesla, Inc., “Tesla Vision Update: Replacing Ultrasonic Sensors with Tesla Vision.” <https://www.raspberrypi.com/documentation/>, zuletzt besucht 07.07.2024, 2024.
- [7] Elon Musk, “Twitter/X.” <https://x.com/elonmusk/status/1447588987317547014?lang=de>, zuletzt besucht 07.07.2024, 2021.
- [8] Honda Motor Co., Ltd., “Honda Riding Assist .” <https://www.youtube.com/watch?v=VH60-R8MOKo>, zuletzt besucht 07.07.2024, 2017.
- [9] NVIDIA Corporation, “Jetson Modules.” <https://developer.nvidia.com/embedded/jetson-modules>, zuletzt besucht 04.07.2024, 2023.
- [10] G. Jocher, A. Chaurasia, and J. Qiu, “Ultralytics YOLO.” <https://github.com/ultralytics/ultralytics>, Jan. 2023.
- [11] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, “Microsoft COCO: Common Objects in Context,” 2014.
- [12] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

- [13] J. Nagi, A. Giusti, F. Nagi, L. M. Gambardella, and G. A. Di Caro in *Online feature extraction for the incremental learning of gestures in human-swarm interaction*, 2014.
- [14] Microsoft, Amazon, Facebook, “Open Neural Network Exchange.” <https://onnx.ai/>, zuletzt besucht 10.08.2024, August 2024.
- [15] Intel Corporation, “OpenVINO.” https://github.com/openvino_toolkit/openvino, zuletzt besucht 10.08.2024, August 2024.
- [16] NVIDIA Corporation, “NVIDIA TensorRT.” <https://developer.nvidia.com/tensorrt>, zuletzt besucht 10.08.2024, August 2024.
- [17] S. Chacon and B. Straub, *Pro git*. Apress, 2014.
- [18] O. Team. <https://pypi.org/project/opencv-python/>, zuletzt besucht 20.08.2024, June 2024.
- [19] Alex Hunziker, *Autonomous Steering of an Electric Bicycle Based on Sensor Fusion Using Model Predictive Control*. Johann-Wolfgang-Goethe Universität Frankfurt am Main, 2019.
- [20] R. P. Ltd., “Raspberry Pi Documentation.” <https://www.raspberrypi.com/documentation/>, zuletzt besucht 11.07.2024, 2024.
- [21] F. P. Aileen, Alexander Dennis Suwardi, “WiFi Signal Strength Degradation Over Different Building Materials,” *JURNAL EMACS*, vol. 3, pp. 109–113, 2021.
- [22] U. Brinkschulte, “Grundlagen für eingebettete Systeme,” April 2021.
- [23] U. Brinkschulte, “Echtzeitprogrammierung,” April 2021.
- [24] The Python Software Foundation, “The Python Profilers.” <https://docs.python.org/3/library/profile.html>, zuletzt besucht 26.06.2024.
- [25] e-con Systems India Pvt Ltd., “MIPI camera vs USB camera – a detailed comparison.” <https://www.e-consystems.com/blog/camera/technology/mipi-camera-vs-usb-camera-a-detailed-comparison/>, zuletzt besucht 28.06.2024, January 2023.
- [26] USB Implementers Forum, Inc., “USB 2.0 Specification.” <https://www.usb.org/document-library/usb-20-specification>, zuletzt besucht 26.06.2024, 2024.
- [27] Hewlett-Packard Development Company, “USB 3.0 Technology.” <https://web.archive.org/web/20140103080959/http://h20195.www2.hp.com/v2/GetPDF.aspx%2F4AA4-2724ENW.pdf>, zuletzt besucht 26.06.2024, 2012.
- [28] D. A. Patterson and C. H. Sequin, “RISC I: a reduced instruction set VLSI computer,” in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, (New York, NY, USA), p. 216–230, Association for Computing Machinery, 1998.

- [29] T. N. Rahman, N. Khan, and Z. I. Zaman, “Redefining Computing: Rise of ARM from consumer to Cloud for energy efficiency,” *World Journal of Advanced Research and Reviews*, vol. 21, p. 817–835, Jan. 2024.
- [30] LANOC Reviews, “AMD Ryzen 9 7950X3D - CPU Performance.” <https://lanoc.org/review/cpus/8673-amd-ryzen-9-7950x3d?start=2>, zuletzt besucht 04.07.2024, 2023.
- [31] NVIDIA Corporation, “JetPack SDK.” <https://developer.nvidia.com/embedded/jetpack>, zuletzt besucht 08.07.2024, 2024.
- [32] Wikipedia, “Erdradius.” <https://de.wikipedia.org/wiki/Erdradius>, zuletzt besucht 17.07.2024, 2024.
- [33] M. J. Hußmann, “Wissen: Wie funktioniert der Global Shutter?” <https://www.fotomagazin.de/test-technik/wissen/wissen-wie-funktioniert-der-global-shutter/>, zuletzt besucht 18.07.2024, 2024.
- [34] L. Thithanhnhan, N.-T. Le, and Y. M. Jang, “OCC-ID: New Broadcasting Service-Based Cloud Model and Image Sensor Communications,” *International Journal of Distributed Sensor Networks*, vol. 2016, pp. 1–10, 06 2016.
- [35] e-con Systems India Pvt Ltd., “e-CAM24 CUNX - Color Global shutter Camera for NVIDIA® Jetson Xavier™ NX / TX2 NX / Nano.” <https://www.e-consystems.com/nvidia-cameras/jetson-xavier-nx-cameras/full-hd-xavier-nx-global-shutter-camera.asp>, zuletzt besucht 18.07.2024, 2024.
- [36] e-con Systems India Pvt Ltd., *Datasheet e-CAM24_CUNX*, June 2023.
- [37] GitHub Inc. <https://github.com/>, zuletzt besucht 27.07.2024, 2024.
- [38] Dwyer, Nelson, and Hansen, “Roboflow (Version 1.0) [Software].” <https://roboflow.com>, zuletzt besucht 27.07.2024, 2024.
- [39] Studyflix GmbH, “PID Regler.” <https://studyflix.de/informatik/pid-regler-1450>, zuletzt besucht 25.09.2024, 2024.
- [40] P. Riekert and T. E. Schunck, “Zur fahrmechanik des gummibereiften kraftfahrzeugs,” *Ingenieur-Archiv*, vol. 11, pp. 210–224, Jun 1940.
- [41] B. Mashadi, N. Ebrahimi, and J. Marzbanrad, “Effect of front-wheel drive or rear-wheel drive on the limit handling behaviour of passenger vehicles,” *Proceedings of The Institution of Mechanical Engineers Part D-journal of Automobile Engineering - PROC INST MECH ENG D-J AUTO*, vol. 221, pp. 393–403, 04 2007.
- [42] C. K. Law, D. Dalal, and S. Shearow, “Robust model predictive control for autonomous vehicles/self driving cars,” 2018.
- [43] Q-Engineering, “Install PyTorch on Jetson Nano..” <https://qengineering.eu/install-pytorch-on-jetson-nano.html>, zuletzt besucht 12.04.2024, 2023.

- [44] Glenn Jocher, “Hardware requirements..” <https://github.com/ultralytics/ultralytics/issues/4106#issuecomment-1663166017>, zuletzt besucht 25.09.2024, 2023.
- [45] Glenn Jocher, “Model Selection.” https://docs.ultralytics.com/yolov5/tutorials/tips_for_best_training_results/#training-settings, zuletzt besucht 05.10.2024, 2024.
- [46] Glenn Jocher, “Performance Metrics.” <https://docs.ultralytics.com/models/yolov8/#supported-tasks-and-modes>, zuletzt besucht 05.10.2024, 2024.
- [47] Glenn Jocher, “TensorRT Export für YOLOv8 Modelle.” <https://docs.ultralytics.com/de/integrations/tensorrt/#tensorrt>, zuletzt besucht 05.10.2024, 2024.
- [48] Atsushi Sakai, “Model predictive speed and steering control.” https://github.com/AtsushiSakai/PythonRobotics/tree/master/PathTracking/model_predictive_speed_and_steer_control, zuletzt besucht 05.10.2024, 2024.
- [49] Meta Platforms Inc., “SAM 2: Segment Anything in Images and Videos.” <https://ai.meta.com/research/publications/sam-2-segment-anything-in-images-and-videos/>, zuletzt besucht 08.10.2024, 2024.
- [50] Jocher, Glenn and Chaurasia, Ayush and Qiu, Jing and ULTRALYTICS, “Ultralytics YOLO11.” <https://docs.ultralytics.com/de/models/yolo11/>, zuletzt besucht 08.10.2024, 2024.
- [51] Raspberry Pi, “Raspberry Pi AI Kit.” <https://www.raspberrypi.com/products/ai-kit/>, zuletzt besucht 08.10.2024, 2024.

Abkürzungsverzeichnis

ARM	Advanced RISC Machines
BBB	Beagle Bone Black
CI/CD	Continuos Integration/ Continuos Deployment
CLI	Command Line Interface
CNN	Convolutional Neural Network
COCO	Common Objects in Context
FPS	Frames Per Second
GPU	Graphics Processing Unit bzw. Grafikbeschleunigungseinheit
IoU	Intersection over Union
KI	Künstliche Intelligenz
mAP	mean Average Precision
MIPI-CSI	Mobile Industry Processor Interface-Camera Serial Interface
PID	Proportional-Integral-Differential
RISC	Reduced Instruction Set Computation
ROI	Region of Interest
SAM	Segment Anything
SoC	System on Chip
USB	Universal Serial Bus
YOLO	You Only Look Once