Frankfurt University of Applied Sciences

Fachbereich 2 Informatik und Ingenieurwissenschaften

Entwicklung und Implementierung praxisorientierter KI-Beispiele auf dem Raspberry Pi 5 und dem AI-HAT+

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Vorgelegt von:

Hai Anh Tran

Studiengang: Informatik (B.Sc.)

Matrikelnummer: 1347788

Referent:	Prof. Dr. Christian Baun
Korreferent:	Prof. Dr. Thomas Gabel

Begonnen am:	20.11.2025
Beendet am:	22.01.2026

Eidesstattliche Erklärung

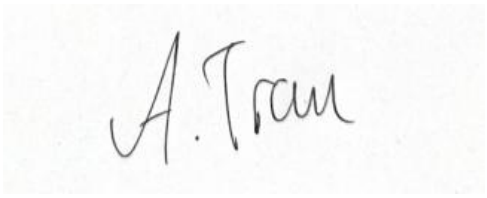
Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen und Abbildungen sind von mir erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit wurde noch nicht in gleicher oder ähnlicher Form bei keiner anderen Prüfungsbehörde eingereicht.

Unterschrift



Zusammenfassung

Diese Arbeit untersucht die Entwicklung und Implementierung praxisorientierter KI-Beispiele auf dem Raspberry Pi 5 in Kombination mit dem AI-HAT+. Das Ziel dieser Arbeit ist es zu zeigen, wie sich moderne KI-Verfahren mit begrenzten Ressourcen einsetzen lassen und welche Herausforderungen dabei in Bezug auf Leistung, Echtzeitfähigkeit und Entwicklungsaufwand auftreten.

Auf einer Plattform, bestehend aus dem Raspberry Pi 5, dem Raspberry Pi Camera Module 2 und dem KI-Beschleuniger AI-HAT+, wurden mehrere Anwendungen umgesetzt: eine Posenerkennung auf Basis von Körperposen, eine Anwendung zur Fahrzeugdetektion und -zählung sowie eine Anwendung zur automatischen Maskierung von Gesichtern. Für diese Beispiele werden passende KI-Modelle ausgewählt und mithilfe von Python, GStreamer und Hailo-Werkzeugen in eine Videoverarbeitungskette integriert.

Die Arbeit beschreibt den vollständigen Entwicklungsprozess der Auswahl der Modelle, bis hin zur Implementierung der Datenverarbeitungsketten und der Evaluation der Prototypen hinsichtlich Genauigkeit, Bildrate und Latenz. Die Ergebnisse zeigen, dass der Raspberry Pi 5 in Verbindung mit dem AI-HAT+ eine leistungsfähige und zugleich kostengünstige Plattform für praxisorientierte KI-Anwendungen darstellt.

Abstract

This thesis researches the development and implementation of practical AI examples on the Raspberry Pi 5 in combination with the AI-HAT+. The aim of this work is to demonstrate how modern AI methods can be used with limited resources and which challenges arise in terms of performance, real-time capability and implementation effort.

The platform consists of the Raspberry Pi 5, the Raspberry Pi Camera Module 2 and the AI accelerator AI-HAT+. Several applications were implemented on this platform: an application for pose detection, for vehicle detection and counting and for automatic blurring of faces. Suitable AI models are selected and integrated into real-time video pipelines using Python, GStreamer and the Hailo tools.

The thesis describes the complete development process, from the selection of the models through to the implementation of the data processing pipelines and the evaluation of the prototypes with respect to accuracy, frame rate, and latency. The results show that the Raspberry Pi 5 in combination with the AI-HAT+ represents a powerful yet cost-effective platform for practical AI applications.

Inhaltsverzeichnis

1. Einführung 11
1.1 Motivation und Hintergrund 11
1.2 Problemstellung und Forschungsfrage 12
1.3 Zielsetzung der Arbeit 12
2. Grundlagen 13
2.1 Künstliche Intelligenz 13
2.2 Maschinelles Lernen 13
2.3 Neuronale Netze und Deep Learning 15
2.4 Training und Inferenz 16
2.5 Edge-KI 16
3. Aktueller Stand von Edge-KI-Plattformen 18
3.1 Bewertungskriterien 18
3.2 Bestehende Edge-KI-Plattformen 19
3.2.1 NVIDIA Jetson 19
3.2.2 Google Coral / Edge TPU 20
3.2.3 Raspberry Pi mit AI-HAT+ 21
3.3 Untersuchung der Plattformen anhand der Kriterien 22
3.4 Verwendete Hardwareplattform 24
3.4.1 Raspberry Pi 5 24
3.4.2 AI-HAT+ 25
3.4.3 Kamera und weitere Peripherie 25
4. Anforderungen und Design der KI-Beispiele 26
4.1 Überblick über die umgesetzten KI-Anwendungen 26
4.2 Funktionale Anforderungen 27
4.2.1 Posenerkennung 27
4.2.2 Fahrzeugzählung 27
4.2.3 Gesichtsmaskierung 28
4.3 Nicht-funktionale Anforderungen 28
4.4 Zielsetzung des Systemdesign 29
4.5 Konzeption der gemeinsamen Systemarchitektur 30
4.6 Auswahl der KI-Modelle 33
4.7 Annahmen und Abgrenzungen 34
5. Implementierung 35

5.1 Entwicklungsumgebung und verwendete Technologien 35
5.1.1 GStreamer als Videopipeline 35
5.1.2 Hailo-Softwareumgebung 36
5.1.3 Python-Bibliotheken für Bildverarbeitung und Numerik 37
5.2 Gemeinsame Basisimplementierung 38
5.3 Implementierung der Gestenerkennung 43
5.4 Implementierung Fahrzeugzählung 47
5.5 Implementierung der Gesichtsmaskierung 49
5.5.1 Abgrenzung der Gesichtsmaskierung gegenüber den anderen Anwendungen 49
5.5.2 Implementierung 51
5.6 Zusammenfassung 54
6. Evaluation 55
6.1 Zielsetzung der Evaluation 55
6.2 Versuchsaufbau 55
6.3 Bewertungsmetriken 56
6.4 Ergebnisse der Posenerkennung 58
6.5 Ergebnisse der Fahrzeugzählung 60
6.6 Ergebnisse der Gesichtsmaskierung 62
6.7 Diskussion und Bewertung der Ergebnisse 64
7. Zusammenfassung und Ausblick 66
7.1 Zusammenfassung der Arbeit 66
7.2 Weiterentwicklungsmöglichkeiten und offene Fragen 67

Abbildungsverzeichnis

Abbildung 1: Aufbau von neuronalen Netzen 15
Abbildung 2: NVIDIA Jetson Orin Nano Developer Kit [..... 19
Abbildung 3: Google Coral Dev Board 20
Abbildung 4: Raspberry Pi 5 mit AI-HAT+ 21
Abbildung 5: Aktivitätsdiagramm der Videoverarbeitungskette 32
Abbildung 6: GStreamer, Elemente, Pads, Pipeline 35
Abbildung 7: Aktivitätsdiagramm Ablauf der Videopipeline 42
Abbildung 8: Architekturdiagramm der Gesichtsmaskierung Pipeline 50
Abbildung 9: Videobild Posenerkennung 59
Abbildung 10: Videobild Fahrzeugzählung 61
Abbildung 11: Videobild Gesichtsmaskierung 63

Tabellenverzeichnis

Tabelle 1: Überblick Edge-KI-Plattformen 23
Tabelle 2: Metriken des Testlaufs für Posenerkennung 58
Tabelle 3: Metriken des Testlaufs für Fahrzeugzählung 60
Tabelle 4: Metriken des Testlaufs für Gesichtsmaskierung 62

Quell- und Pseudocodeverzeichnis

Codeblock 1: Quellcode Pipeline bauen 38
Codeblock 2: Quellcode Hailo-Elemente holen und konfigurieren 39
Codeblock 3: Quellcode Callback-Funktion 40
Codeblock 4: Quellcode identity-Element holen und anhängen 40
Codeblock 5: Quellcode Main-Funktion 41
Codeblock 6: Quellcode Umrechnung der Gelenkpunkte in Bildpixel 44
Codeblock 7: Quellcode HANDS_UP Regeln 44
Codeblock 8: Quellcode T_POSE Regeln 45
Codeblock 9: Quellcode ARMS_DOWN Regeln 46
Codeblock 10: Quellcode Callback-Funktion Posenerkennung 46
Codeblock 11: Pseudocode Callback-Funktion Fahrzeugzählung 48
Codeblock 12: Pseudocode Callback-Funktion on_preview_sample 51
Codeblock 13: Pseudocode worker_loop Thread 52
Codeblock 14: Quellcode map_net_to_preview_letterbox Funktion 53
Codeblock 15: Pseudocode Bildmanipulation Funktion 53

Abkürzungsverzeichnis

AI	Artificial Intelligence
ca.	circa
CNNs	Convolutional Neural Networks
CPU	Central Processing Unit
CSI	Camera Serial Interface
CV	Computer Vision
GPIO	General Purpose Input/Output
GPU	Graphics Processing Unit
HAT	Hardware Attached on Top
HDMI	High-Definition Multimedia Interface
HEF	Hailo Executable Format
ID	Identification
KI	Künstliche Intelligenz
MB	Megabyte
ML	Machine Learning
ms.	milliseconds
NPU	Neural Processing Unit
OS	Operating System
PCIe	Peripheral Component Interconnect Express
RAM	Random Memory Access
ROI	Region Of Interest
RT	Runtime
SDK	Software Development Kit
TOPS	Tera Operation Per Second
TPU	Tensor Processing Unit
USB	Universal Serial Bus
z. B.	zum Beispiel

1. Einführung

Im ersten Kapitel wird das Thema der Arbeit eingeführt und in einen übergeordneten Kontext eingeordnet. Zunächst werden Motivation und Hintergrund von Künstlicher Intelligenz auf eingebetteten Systemen erläutert. Anschließend werden die Problemstellung und die zentrale Forschungsfrage formuliert, bevor die Zielsetzung der Arbeit beschrieben wird.

1.1 Motivation und Hintergrund

In den vergangenen Jahren hat der Einsatz von Künstlicher Intelligenz im Alltag deutlich an Bedeutung gewonnen. KI-basierte Systeme sind längst nicht mehr nur in großen Rechenzentren zu finden, sondern begegnen uns in Smartphones, Haushaltsgeräten, Fahrzeugen und zunehmend auch in eingebetteten Systemen. Als „Edge-KI“ werden Systeme bezeichnet, die Daten direkt am Gerät verarbeiten und eine Echtzeitanwendung ermöglichen, ohne einen Umweg über die Cloud.

Parallel dazu hat sich der Markt für kompakte, leistungsfähige Einplatinencomputer weiterentwickelt. Der Raspberry Pi 5 verfügt über eine leistungsstarke Hardwarebasis, die in Kombination mit spezialisierten KI-Beschleunigern, wie dem AI-HAT+, ein attraktives Gesamtpaket für die Umsetzung von praxisnahen KI-Anwendungen bietet.

Ein wesentlicher Vorteil solcher Edge-KI-Plattformen ist die Möglichkeit, komplexe Aufgaben wie Bildverarbeitung oder Objekterkennung auf kostengünstiger Hardware auszuführen. Wo früher leistungsstarke Desktop-Rechner oder Cloud-Server erforderlich waren, genügen heute kompakte Module, die sich problemlos in bestehende Systeme integrieren lassen. Der AI-HAT+ erweitert den Raspberry Pi 5 dabei um einen dedizierten KI-Beschleuniger, der speziell für KI-Aufgaben optimiert ist und gleichzeitig einen geringen Energieverbrauch ermöglicht. Dadurch werden Anwendungen realisierbar, die sowohl interaktiv als auch echtzeitfähig sind.

1.2 Problemstellung und Forschungsfrage

Im Rahmen dieser Forschung stellt sich die Frage, wie sich eine praxisnahe Kombination aus Einplatinenrechner und KI-Beschleuniger für typische Bildverarbeitungsaufgaben eignet. Es existieren zwar zahlreiche Beispielanwendungen, jedoch konzentrieren sie sich häufig auf einzelne Szenarien und nicht auf den kompletten Entwicklungsprozess. Für den Raspberry Pi 5 mit dem AI-HAT+ ist bisher begrenzt dokumentiert, wie sich mehrere verschiedene Anwendungen auf einer gemeinsamen Grundlage realisieren und im Hinblick auf Bildrate und Latenz bewerten lassen.

Vor diesem Hintergrund ergibt sich die zentrale Frage dieser Arbeit: Inwieweit eignet sich die Kombination aus Raspberry Pi 5 und AI-HAT+ als Plattform für die Entwicklung von praxisorientierten KI-Anwendungen?

1.3 Zielsetzung der Arbeit

Im Rahmen dieser Arbeit werden auf Basis des Raspberry Pi 5 und des AI-HAT+ mehrere praxisorientierte KI-Beispiele entwickelt und implementiert. Diese Beispiele decken unterschiedliche Anwendungsfelder ab. Von Mensch-Maschine-Interaktion über Datenschutz bis hin zur einfachen Verkehrsüberwachung.

Das Ziel der Arbeit ist es, den gesamten Entwicklungsprozess dieser Anwendungen nachvollziehbar darzustellen. Der Prozess fängt bei der Auswahl geeigneter Modelle und Frameworks an und geht weiter zur Umsetzung der Videoverarbeitungskette, bis hin zur Evaluation hinsichtlich Bildrate und Latenz. Damit soll aufgezeigt werden, welches Potenzial der Raspberry Pi 5 in Verbindung mit dem AI-HAT+ als Plattform für praxisnahe KI-Beispiele bietet und welche Chancen und Grenzen sich beim Einsatz solcher Systeme ergeben.

2. Grundlagen

In diesem Kapitel werden die theoretischen und technischen Grundlagen vorgestellt, die erforderlich sind, um die im Rahmen dieser Arbeit entwickelten KI-Anwendungen zu verstehen. Als Erstes werden zentrale Begriffe aus dem Bereich Künstliche Intelligenz und Maschinelles Lernen erläutert. Daraufhin wird auf neuronale Netze eingegangen und der Unterschied zwischen Training und Inferenz wird erklärt. Abschließend wird das Konzept der Edge-KI vorgestellt, welches für den Einsatz des Raspberry Pi 5 in Kombination mit dem AI-HAT+ eine zentrale Rolle spielt.

2.1 Künstliche Intelligenz

Unter Künstlicher Intelligenz (KI) werden Verfahren verstanden, die es Computersystemen ermöglichen, Aufgaben zu lösen, welche typischerweise menschliche Intelligenz erfordern würden. Anwendungen und Geräte, die mit KI ausgestattet sind, können unter anderem Bilder und Sprachen erkennen, sowie Entscheidungen treffen und aus Erfahrungen lernen [1].

In der Praxis wird der Begriff KI häufig als Oberbegriff für verschiedene Methoden verwendet. Diese reichen von einfachen regelbasierten Systemen bis hin zu komplexen lernenden Systemen. In dieser Arbeit steht insbesondere der Bereich des maschinellen Lernens im Vordergrund, bei dem Systeme nicht für einen expliziten Einzelfall programmiert werden, sondern selbstständig Muster in Daten erkennen und daraus ein Modell ableiten [2].

Im Bereich der Künstlichen Intelligenz wird zwischen „schwacher KI“ und „starker KI“ unterschieden. „Starke KI“ ist in der Lage, jede intellektuelle Aufgabe zu lösen. Im Gegensatz dazu löst „schwache KI“ konkrete Aufgaben. [3]

Die in dieser Arbeit entwickelten Anwendungen basieren auf lernenden Modellen und werden dem Bereich der „schwachen KI“ zugeordnet.

2.2 Maschinelles Lernen

Maschinelles Lernen (ML) ist ein Teilgebiet der KI und befasst sich mit Algorithmen, die aus Beispieldaten lernen, anstatt feste Regeln vorzugeben. Daraus ergibt sich ein Modell. Das

System erhält dabei Trainingsdaten, die aus Eingaben (z. B. Bildern) und oft auch zugehörigen Zielwerten (z. B. Klassenbeschriftungen wie „Auto“ oder „Person“) bestehen. Im Trainingsprozess werden interne Parameter der Modelle angepasst, damit es neue, bisher unbekannte Daten möglichst korrekt verarbeiten kann [2].

Je nach Art der verfügbaren Daten und Zielsetzung wird zwischen verschiedenen Lernparadigmen unterschieden:

Überwachtes Lernen:
Beim überwachten Lernen werden Modelle mit gekennzeichneten Datensätzen trainiert, zum Beispiel ein Bild mit der Beschriftung „Fahrzeug“. Das Modell passt während des Trainings seine Gewichtungen so an, dass es Daten klassifizieren oder Ergebnisse möglichst präzise vorhersagen kann [2]. Typische Verfahren sind Klassifikation, wie die Erkennung einer bestimmten Körperpose, und Regression, also die Vorhersage eines kontinuierlichen Werts.

Unüberwachtes Lernen:
Bei dem unüberwachten Lernen liegen nur Eingabedaten ohne explizite Zielwerte vor. Das Modell versucht, Strukturen oder Cluster (Gruppierungen) in den Daten zu finden, die nicht offensichtlich sind. Typische Aufgaben sind das Gruppieren von ähnlichen Daten (Clustering), Assoziationen in den Daten zu finden und die Dimensionsreduktion, also Daten zu komprimieren und zu visualisieren [2].

Bestärkendes Lernen:
Ein Agent lernt durch Interaktion mit einer Umgebung und erhält Belohnungen oder Strafen. Der Agent wählt Aktionen, bekommt eine Rückmeldung in Form von einer Belohnung oder einer Strafe und passt sein Verhalten an. Das Ziel dabei ist nicht die Fehler-Minimierung, sondern die maximale Belohnung [2]. Typische Anwendungsfälle sind in der Robotertechnik oder in Videospielen zu finden.

In dieser Arbeit wurden Modelle verwendet, die im Rahmen des überwachten Lernens trainiert worden sind. Das eigentliche Training der Modelle findet dabei nicht auf dem Raspberry Pi 5 statt, sondern wurde zuvor auf leistungsstärkeren Systemen durchgeführt. Auf dem Raspberry Pi wird anschließend nur noch die sogenannte Inferenz ausgeführt (siehe Abschnitt 2.4).

2.3 Neuronale Netze und Deep Learning

Viele moderne KI-Anwendungen basieren auf künstlichen neuronalen Netzen. Neuronale Netze sind Modelle des maschinellen Lernens, die aus vielen einfachen „Neuronen“ bestehen. Sie orientieren sich grob an der Arbeitsweise biologischer Nervenzellen und führen einfache Rechenoperationen durch. Durch das Zusammenschalten vieler solcher Neuronen entstehen mehrere Schichten, die zusammen ein leistungsfähiges Modell darstellen und komplexe Muster erkennen können [4].

Neuronale Netze bilden laut der Quelle [4] die Basis von Deep Learning. Die Eingabeschicht nimmt Daten auf, während mehrere versteckte Schichten die Daten verarbeiten. Im Anschluss liefert die Ausgabeschicht das Ergebnis. Jede Verbindung hat ein Gewicht, um abzuwägen, wie wichtig ein Eingangssignal ist. Zusätzlich hat jeder Knoten einen Schwellenwert, um abzustimmen, wann er anspringt. Wenn das Ergebnis über dem Schwellenwert liegt, wird der Knoten aktiviert und das Signal wird an die nächste Schicht weitergeleitet. Somit können neuronale Netze hierarchische Merkmale aus Rohdaten lernen. Im Kontext der Bildverarbeitung werden häufig Convolutional Neural Networks eingesetzt (CNNs). Diese nutzen Faltungsoperationen (Convolutions), um Bildstrukturen wie Kanten, Formen oder Texturen zu erkennen. Typische Aufgaben, die mit CNN-basierten Modellen gelöst werden, sind Bildklassifizierung und Objekterkennung.

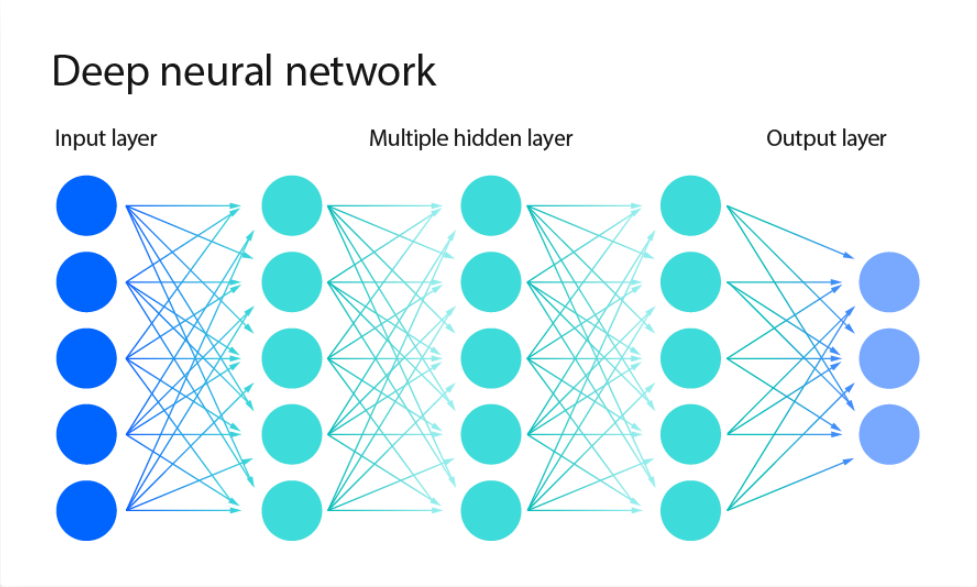


Abbildung 1: Aufbau von neuronalen Netzen [4]

2.4 Training und Inferenz

Im Lebenszyklus eines neuronalen Netzes wird zwischen zwei Phasen unterschieden: Training und Inferenz.

Training:
In der Trainingsphase wird das Modell mit großen Mengen von beschrifteten Daten trainiert. Dabei werden die Parameter des Netzes so angepasst, dass der Fehler zwischen den Modellvorhersagen und den Zielwerten minimiert wird [5]. Dieser Prozess ist sehr rechenintensiv und wird in der Regel auf leistungsfähiger Hardware durchgeführt.

Inferenz:
In der Inferenzphase ist das Training abgeschlossen. Das Modell wird mit neuen Eingabedaten versorgt und berechnet die entsprechenden Ausgaben. Ein Beispiel wäre ein Bild einer Person als Eingabe und die Erkennung (markierter Bildbereich) sowie die Klassifizierung der Person als Ausgabe. Hier steht insbesondere die Effizienz im Vordergrund. Die Berechnungen müssen schnell genug sein, um Echtzeit-Anforderungen zu erfüllen, und gleichzeitig auf der verfügbaren Hardware ausführbar bleiben [5].

Die vorliegende Arbeit konzentriert sich ausschließlich auf die Inferenzphase. Die verwendeten Modelle werden in einem bereits trainierten Zustand bereitgestellt, typischerweise in einem geeigneten Format für den AI-HAT+. Der AI-HAT+ übernimmt dabei einen Großteil der Rechenarbeit der Inferenz.

2.5 Edge-KI

Ein zentrales Konzept dieser Arbeit ist die Edge-KI. Sie bezeichnet das Ausführen der KI-Algorithmen direkt auf dem Endgerät selbst oder in unmittelbarer Nähe der Datenquelle. Somit wird eine Echtzeit-Datenverarbeitung und -analyse ermöglicht, ohne eine Abhängigkeit von einer Cloud-Infrastruktur [6].

Edge-KI bietet mehrere Vorteile laut Quelle [6]:

- Geringe Latenz:
  Da die Verarbeitung lokal erfolgt, entfallen Netzwerklaufzeiten. Dies ist besonders wichtig für Anwendungen mit Echtzeitanforderungen, zum Beispiel bei einer Fahrzeugerkennung bei selbstfahrenden Autos.

- Datenschutz:
  Sensible Daten müssen nicht in ein anderes Netzwerk übertragen werden. Dies reduziert das Risiko von Datenschutzverletzungen, da die Informationen direkt auf dem Gerät verarbeitet werden.

- Echtzeitanalyse:
  Die Anwendungen können ohne stabile Internetverbindung betrieben werden, was in vielen Szenarien (z. B. mobile Systeme, abgelegene Orte) von Vorteil ist.

Edge-KI stehen auch Herausforderungen gegenüber [6, 7]:

- Viele Edge-Geräte verfügen im Vergleich zu Cloud-Servern noch deutlich weniger Rechenleistung, Speicher und Energie. KI-Modelle müssen daher speziell optimiert und angepasst werden, damit sie auf der Zielhardware benutzt werden können.

- Edge-KI eignet sich für lokale Echtzeitaufgaben direkt auf dem Gerät. Für rechenintensive Aufgaben wie Training oder Datenaggregation, wird jedoch häufig weiterhin die Cloud benötigt. Zudem ist die Verwaltung vieler verteilter Edge-Geräte im großen Maßstab aufwendig.

Der Raspberry Pi 5 in Kombination mit dem AI-HAT+ ist ein typisches Beispiel einer Edge-KI-Plattform. Die in dieser Arbeit entwickelten Anwendungen demonstrieren, wie sich komplexe KI-Funktionen direkt auf dem Endgerät realisieren lassen, trotz begrenzter Ressourcen.

3. Aktueller Stand von Edge-KI-Plattformen

Wie in den vorherigen Abschnitten genannt, hat sich der Bereich der Edge-KI weiterentwickelt. Für typische KI-Aufgaben wie Objekterkennung existieren inzwischen zahlreiche Hard- und Softwarelösungen, die sich in Leistungsfähigkeit, Kosten und Entwicklungsaufwand unterscheiden. Das Ziel dieses Kapitels ist es, den verwendeten Raspberry Pi 5 in Kombination mit dem KI-Beschleuniger AI-HAT+ in diesen Kontext einzuordnen. Dazu werden Kriterien definiert und anschließend bestehende Lösungen vorgestellt. Die Plattformen werden anhand dieser Kriterien verglichen. Auf dieser Grundlage lässt sich begründen, warum der Raspberry Pi mit dem AI-HAT für die angestrebten KI-Beispiele geeignet ist und wo weiterhin Bedarf für eigene Entwicklungen und Untersuchungen besteht.

3.1 Bewertungskriterien

Ein zentrales Kriterium ist die **Funktionalität**, also ob und in welchem Umfang typische Bildverarbeitungsaufgaben wie Objekterkennung oder Segmentierung unterstützt werden. Dazu sollten vorgefertigte Modelle oder Beispielanwendungen existieren, um die Funktionalität der Plattform zu bestätigen.

Eng damit verbunden ist die **Leistungsfähigkeit** im Hinblick auf Echtzeitbetrieb. Es ist relevant, ob eine Plattform die Bildraten im Bereich von mehreren zehn Bildern pro Sekunde mit möglichst geringer Verzögerung erreichen kann. Die Hersteller der Plattformen geben die Rechenleistung von KI-Beschleunigern häufig in „TOPS“ an, also in „Tera-Operationen pro Sekunde“. Dieser Wert dient in dieser Arbeit als grobe Orientierung für die Rechenleistung.

Ein weiteres Kriterium sind die **Anschaffungs- und Betriebskosten**. Da sich die Arbeit an einer typischen Lehr- und Entwicklungsumgebung orientiert, spielt die Verfügbarkeit als auch die Kosten eine Rolle.

Schließlich ist der **Entwicklungsaufwand** von Bedeutung: Eine Lösung ist für den Rahmen einer Bachelorarbeit nur dann praktikabel, wenn sie über eine nachvollziehbare Dokumentation, Beispielprojekte und ein nutzbares Software-Entwicklungspaket (SDK) verfügt.

3.2 Bestehende Edge-KI-Plattformen

Als repräsentative Beispiele aktueller Edge-KI-Lösungen werden im Folgenden drei Ansätze betrachtet: NVIDIA Jetson, Google Coral mit Edge TPU sowie der Raspberry Pi 5 in Kombination mit dem AI-HAT+.

3.2.1 NVIDIA Jetson

Plattformen der NVIDIA-Jetson-Reihe sind kompakte Rechnersysteme, die für rechenintensive Bild- und KI-Anwendungen ausgelegt sind [8]. Eine Charakteristik der Jetson-Systeme ist die Integration der Rechenkomponenten und Beschleunigereinheiten auf dem Gerät. Die Systeme basieren auf einem Chip, der Prozessor, Grafikeinheit und weitere Komponenten auf einem System kombiniert [9]. Für KI-Anwendungen ist der integrierte Grafikprozessor der Jetson-Systeme relevant, um die Rechenoperationen der neuronalen Netze besser parallelisieren zu können [10]. Aus der Sicht der Softwareentwicklung bieten Jetson-Systeme eine Vielzahl an Bibliotheken und Werkzeugen an, die die Ausführung neuronaler Netze auf der GPU beschleunigen. Diese Werkzeuge umfassen sowohl allgemeine Rechenbibliotheken als auch Optimierungswerkzeuge für neuronale Netze [11]. Der Entwicklungsaufwand kann bei den Systemen sinken, da viele Modelle ohne grundlegende Umformung lauffähig sind. In der Praxis werden Jetson-Plattformen oft benutzt, wenn eine hohe Leistung oder anspruchsvollere Modelle benötigt werden.

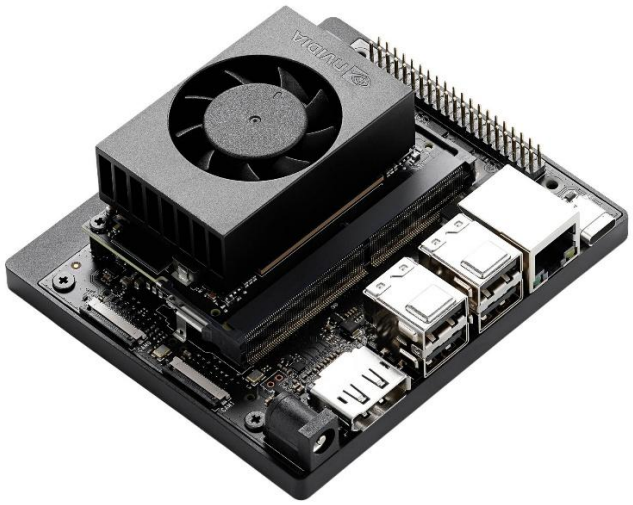


Abbildung 2: NVIDIA Jetson Orin Nano Developer Kit [41]

3.2.2 Google Coral / Edge TPU

Google-Coral-Plattformen enthalten als Zusatzhardware einen KI-Beschleuniger an einem Einplatinenrechner. Die Kernkomponente ist die Edge TPU: ein spezialisierter KI-Chip für die Ausführung neuronaler Netze [12]. Der Anschluss erfolgt dabei meist über die PCIe-Schnittstelle. PCI Express ist eine interne Schnittstelle, die für hohe Datenraten ausgelegt ist und sich besonders für den Transfer von großen Datenmengen eignet. Das zentrale Merkmal dieses Ansatzes ist eine starke Spezialisierung auf die effiziente Inferenz. Für die Edge TPU gelten jedoch strikte Anforderungen an die Modellform: Häufig müssen Modelle in einem kompakten Format vorliegen (TensorFlow Lite) und als vollständig quantisierte Ganzzahlmodelle bereitgestellt werden, damit sie vollständig auf dem Beschleuniger ausgeführt werden können [13]. Laut Coral reduziert die Quantisierung die Zahlenpräzision (von Fließkommazahl zu Ganzzahlen) im Modell und senkt dadurch den Speicherbedarf und den Rechenaufwand, was zu einer verbesserten Geschwindigkeit und Energieeffizienz führt [13].

Abbildung 3: Google Coral Dev Board [42]

3.2.3 Raspberry Pi mit AI-HAT+

Der in dieser Arbeit verwendete Raspberry Pi 5 ist ein weiterer Einplatinenrechner, der durch Zusatzhardware für KI-Aufgaben erweitert werden kann. Der AI-HAT+ ist dabei eine Aufsteckplatine, die einen dedizierten KI-Beschleuniger enthält. Technisch gesehen handelt es sich hierbei um einen Hailo-basierten Beschleuniger, der über die PCIe-Schnittstelle angebunden wird [14].

In einer typischen KI-Anwendung übernimmt der Raspberry Pi die Aufgaben der Datenerfassung und Systemsteuerung, wie den Zugriff auf die Kamera und das Dateisystem. Der AI-HAT+ führt dabei die rechenintensive Inferenz aus. Daraus ergibt sich eine Arbeitsaufteilung, die die CPU des Raspberry Pi entlastet. Wie bei den Coral-Plattformen müssen die Modelle in einem speziellen Format (HEF-Modelle) für den KI-Beschleuniger vorbereitet werden. Dazu gehört die Anpassung der Rechenoperationen sowie die Quantisierung. Im Abschnitt 3.4 werden der Raspberry Pi 5 und AI-HAT+ im Einzelnen betrachtet.

Abbildung 4: Raspberry Pi 5 mit AI-HAT+ [14]

3.3 Untersuchung der Plattformen anhand der Kriterien

Die im Abschnitt 3.1 definierten Kriterien werden im Folgenden auf die drei betrachteten Plattformansätze angewendet, um den Raspberry Pi 5 mit AI HAT+ in den Stand der Technik einzuordnen.

In Hinsicht auf die **Funktionalität** unterstützen grundsätzlich alle drei Ansätze typische Bildverarbeitungsaufgaben wie Objekterkennung oder Segmentierung. Unterschiede ergeben sich jedoch bei der Umsetzbarkeit. Bei Jetson-Systemen ist die Anzahl von lauffähigen Modellen hoch, da die Beschleunigung über die integrierte GPU erfolgt und es wenige Vorgaben an Modellformate gibt. Bei externen KI-Beschleunigern wie Coral und AI-HAT+ ist die Funktionalität stärker davon abhängig, ob ein Modell in der geforderten Form bereitgestellt werden kann. Dazu müssen die im Modell verwendeten Rechenoperationen vom jeweiligen Beschleuniger unterstützt werden. Somit ist die Kompatibilität der konkreten Modelle ein entscheidender Punkt.

Für die **Echtzeitfähigkeit** gilt, dass Jetson-Systeme in der Regel eine hohe Rechenleistung erbringen, die für anspruchsvollere Modelle ein klarer Vorteil ist. Gleichzeitig sind hierfür höhere Anforderungen an Energieversorgung und Wärmeabfuhr [15] zu berücksichtigen, was die Einordnung als „kompaktes“ und „leichtgewichtiges“ Lehr- und Entwicklungssystem relativiert. Bei Coral- und AI-HAT+-basierten Lösungen hängt die erreichbare Echtzeitfähigkeit davon ab, ob das Modell vollständig auf dem KI-Beschleuniger ausgeführt werden kann. Insbesondere Arbeitsschritte wie Bildskalierung, Filterung oder die Ausgabe von Ergebnissen können die Gesamtlatenz beeinflussen.

Das Kriterium **Anschaffungs- und Betriebskosten** ist für den Kontext einer Lehr- und Entwicklungsumgebung relevant, da es die Verfügbarkeit und den Einsatz in typischen Projektszenarien beeinflusst. Da die Jetson-Systeme leistungsorientiert sind und als Gesamtsystem eingeordnet werden, liegen sie meist in höheren Anschaffungsklassen. Die Einplatinenrechner sind als Basis weit verbreitet und bieten damit eine eher kostengünstige Lehr- und Entwicklungsumgebung.

Ergänzend dazu wird der **Entwicklungsaufwand** minimiert, wenn nachvollziehbare Dokumentation, Beispielprojekte und Software-Entwicklungspakete verfügbar sind. Die Jetson-Systeme verfügen über eine Anzahl von umfangreichen Entwicklungswerkzeugen, jedoch erfordern sie dadurch auch eine stärkere Einarbeitung in die Systemumgebung. Bei Coral-Systemen ergibt sich der Aufwand aus den Modellvorgaben und möglichen Prüfungen

für die Modellkompatibilität. Der AI-HAT+ ist in die Raspberry-Pi-Umgebung integriert und es stehen Dokumentationen und Beispielprojekte zur Verfügung. Allerdings werden Werkzeuge benötigt, um Modelle in ein geeignetes Ausführungsformat zu überführen.

Die Tabelle 1 gibt einen zusammenfassenden Überblick über die drei betrachteten Edge-KI-Plattformen und ordnet sie den Kriterien zu. Herstellerzahlen der TOPS [16, 17, 14] können nur als grobe Orientierung verwendet werden, da sie von der Modellstruktur und Rechenart abhängen.

Plattform	**Funktionalität**	**Echtzeitfähigkeit**	**Anschaffungs-/Betriebskosten**	**Entwicklungsaufwand**
NVIDIA Jetson (Orin Nano)	Breites Spektrum für Bild-verarbeitung	Hohe Leistung, höherer Energiebedarf, bis zu 67 TOPS	höhere Anschaffung, höherer Betrieb,	Viele Bibliotheken, hoher Einstieg
Google Coral/ Edge TPU	Effiziente Bildmodelle	Schnell, nur bei Kompatibilität, 4 TOPS	niedriger, Einplatinenrechner-basiert	Modellhürden
Raspberry Pi 5 mit AI-HAT+	Echtzeit-Inferenz auf dem Pi	Hoch, CPU-Vor/Nachverarbei-tung, 13/26 TOPS	niedriger, Einplatinenrechner-basiert	Pi-Integration, Werkzeugkette erforderlich

Tabelle 1: Überblick Edge-KI-Plattformen

Zusammenfassend verfügen die drei betrachteten Ansätze alle über die Kapazität, um typische KI-Aufgaben zu bewältigen. Dabei unterscheiden sie sich primär im Systemkonzept. Die Jetson-Systeme sind als integrierte Gesamtsysteme konzipiert und bieten eine hohe Rechenleistung. Jedoch erfordern sie auch eine abgestimmte Systemumgebung, um effizient zu funktionieren. Zwar sind sowohl Coral- als auch AI-HAT-Systeme modellabhängig, jedoch unterscheiden sie sich in der Art der Abhängigkeit. Bei Coral ergeben sich Einschränkungen aus strikten Modellvorgaben, während bei dem AI-HAT+ das erforderliche Ausführungsformat im Vordergrund steht. Für die vorliegende Arbeit ist dabei entscheidend, dass der Raspberry Pi 5 eine verbreitete und nachvollziehbare Basis für Kamera- und Videodatenverarbeitung bietet und der AI HAT+ die rechenintensive Inferenz auslagert. Durch die Nutzung vorgefertigter

Modelle kann der Entwicklungsfokus auf die Umsetzung gelegt werden sowie auf die Evaluation der Anwendungen. Damit eignet sich der Raspberry Pi 5 mit AI-HAT+, um praxisnahe KI-Beispiele zu implementieren und die Grenzen unter realistischen Bedingungen zu untersuchen.

3.4 Verwendete Hardwareplattform

In diesem Abschnitt wird die in dieser Arbeit verwendete Hardwareplattform vorgestellt. Der Raspberry Pi 5 in Kombination mit dem KI-Beschleuniger AI-HAT+ bilden die Basis für alle entwickelten Anwendungen. Ergänzend kommen eine Kamera sowie weitere Peripheriegeräte zum Einsatz, um Videoeingaben zu erfassen und die Ergebnisse der KI-Verarbeitung darzustellen.

3.4.1 Raspberry Pi 5

Der Raspberry Pi 5 ist die fünfte Generation der weitverbreiteten Einplatinencomputer der Raspberry-Pi-Familie. Er ist für einen kostengünstigen, zugleich aber leistungsfähigen Einsatz in Lehrumgebungen, Prototypen und eingebetteten Anwendungen konzipiert [18].

Zentrale Merkmale [19, 20], die für diese Arbeit relevant sind, sind insbesondere:

- eine mehrkernfähige CPU, die die Ausführung der Steuerlogik, der Videoverarbeitung sowie der Ansteuerung des KI-Beschleunigers übernimmt
- ausreichend Hauptspeicher, um Videodaten, Puffer und die zur Inferenz benötigten Datenstrukturen im Arbeitsspeicher zu halten
- eine leistungsfähige Videoeinheit (GPU) zur Beschleunigung von Grafik- und Videoaufgaben
- standardisierte Schnittstellen wie HDMI, GPIO-Pins sowie CSI- oder USB-Schnittstellen für den Anschluss von Kameras und anderen Geräten

Der Raspberry Pi 5 übernimmt in dieser Arbeit die Rolle der zentralen Steuereinheit. Er initialisiert die Kamera, konfiguriert die Datenverarbeitungskette, übergibt Bilddaten an den AI-HAT+ und führt die Nachverarbeitung der Inferenzresultate durch.

3.4.2 AI-HAT+

Der AI-HAT+ ist ein Aufsteckmodul (HAT) für den Raspberry Pi 5, welcher einen „Neural-Network-Accelerator“ (NPU) integriert. Dieser ist darauf ausgelegt, neuronale Netze effizient auszuführen, und übernimmt einen Großteil der rechenintensiven Operationen, die bei der Inferenz von Deep-Learning-Modellen anfallen. „Die NPU erlaubt es, die beschleunigten KI-Modelle lokal auszuführen, sodass keine Daten zur Verarbeitung an einen Cloud-Server übertragen werden müssen.“ (eigene Übersetzung) [14].

Je nach Variante des Hailo-Chips bietet das System 13 TOPS (Hailo-8L) oder 26 TOPS (Hailo-8). Die Kommunikation zwischen Raspberry Pi 5 und AI-HAT+ erfolgt über die definierte PCIe-Schnittstelle des HATs [14].

Der Vorteil des AI-HAT+ liegt insbesondere in der deutlich höheren Inferenzleistung im Vergleich zu einer reinen CPU-Inferenz. Dadurch werden Bildraten im Echtzeitbereich ermöglicht, die für Anwendungen wie Personenerkennung erforderlich sind. Der AI-HAT+ übernimmt in dieser Arbeit den Inferenzteil der Anwendungen.

3.4.3 Kamera und weitere Peripherie

Für die Erfassung der Bilddaten wird das Raspberry Pi Camera Module 2 verwendet. Diese ist mit dem Raspberry Pi 5 verbunden. Wichtig ist dabei, dass die Kamera eine ausreichende Bildauflösung und Bildrate liefert, um die im Rahmen dieser Arbeit untersuchten Anwendungen sinnvoll zu betreiben. Typischerweise wird mit der Auflösung von 720p und Bildraten zwischen 15 und 30 Bildern pro Sekunde gearbeitet.

Neben der Kamera kommen weitere Peripheriegeräte zum Einsatz, unter anderem:

- einen Monitor zur Darstellung des Videobildes und der KI-Ergebnisse
- Eingabegeräte wie Tastatur und Maus zur Bedienung des Systems

Die Gesamtplattform aus Raspberry Pi 5, AI-HAT+, Kamera und Peripherie bildet eine kompakte Edge-KI-Plattform, ohne dass eine externe Cloud-Infrastruktur angebunden werden muss. Sie dient in dieser Arbeit als Referenzplattform für die Untersuchung, wie praxisorientierte KI-Anwendungen unter realistischen Randbedingungen auf einer ressourcenbegrenzten Hardware umgesetzt werden können.

4. Anforderungen und Design der KI-Beispiele

In diesem Kapitel werden die Anforderungen der entwickelten KI-Anwendungen beschrieben und das darauf aufbauende Design der Systemarchitektur vorgestellt. Im Fokus stehen drei KI-Beispiele, die auf Bild- bzw. Videodaten operieren und in Echtzeit ausgeführt werden sollen. Die Architektur soll dabei die begrenzten Hardware-Ressourcen der Plattform berücksichtigen und gleichzeitig genügend Flexibilität für die verschiedenen Anwendungsfälle bieten.

4.1 Überblick über die umgesetzten KI-Anwendungen

Im Rahmen dieser Arbeit werden folgende KI-Anwendungen realisiert:

Posenerkennung:
In dieser Anwendung wird die Körperhaltung einer Person im Kamerabild analysiert. Ein Modell zur Schätzung von Körperposen erkennt sogenannte „Schlüsselpunkte“. Diese Punkte sind Positionen von bestimmten Gelenkpunkten wie Schultern, Ellenbogen oder Handgelenken. Aus diesen Angaben lassen sich diskrete Posen wie „Arme gehoben“ oder „Arme unten“ mithilfe einfacher Regeln ableiten. Diese Anwendung eignet sich insbesondere zur Demonstration von Mensch-Maschine-Interaktion über Körperbewegungen.

Gesichtsmaskierung:
Diese Anwendung detektiert das Gesicht im Kamerabild und maskiert es durch Verpixelung oder Unschärfe der entsprechenden Bildbereiche in Echtzeit. Ein Gesichtsdetektionsmodell liefert den Bildbereich des erkannten Gesichts. Auf dieser Basis wird entschieden, welche Bildbereiche maskiert werden sollen. Sie adressiert den Aspekt des Datenschutzes und zeigt, wie KI-basierte Erkennung mit anschließender Bildmanipulation kombiniert werden kann.

Fahrzeugzählung:
In dieser Anwendung werden Fahrzeuge im Bild erkannt und gezählt, sobald sie eine im Bild definierte virtuelle Linie überqueren. Ein Objekterkennungsmodell erkennt Fahrzeuge im Bild und markiert die Position der Bildbereiche. Aus dieser Position wird abgeleitet, ob sich der Mittelpunkt einer Fahrzeugbox bewegt hat und ob sie sich von einer Seite der Linie auf die

andere bewegt hat. Die Anwendung demonstriert ein einfaches Szenario der Verkehrsüberwachung und eignet sich, um die Kombination aus Objektverfolgung und Ereignislogik („Linienkreuzung“) zu zeigen.

Die drei Beispiele decken damit unterschiedliche Anwendungsfelder ab: Mensch-Maschine-Interaktion, Datenschutz und Videoanalyse im Verkehrsbereich. Gleichzeitig nutzen sie ähnliche technische Bausteine wie Kamera, Inferenz, Nachverarbeitung, die eine gemeinsame Systemarchitektur ermöglichen.

4.2 Funktionale Anforderungen

Im Folgenden werden die funktionalen Anforderungen an die einzelnen Anwendungen beschrieben. Sie legen fest, welche Funktionen die Prototypen bereitstellen sollen.

4.2.1 Posenerkennung

Die Anwendung soll einen Videostream erfassen können über eine angeschlossene Kamera. Dabei soll es möglich sein, mindestens eine Person zu erkennen und die Körpergeste über Schlüsselpunkte zu berechnen. Durch vordefinierte Regeln soll die Anwendung die Körperpose erkennen und die erkannte Pose im Videobild oder im Terminal in Echtzeit anzeigen. Dabei liegt die Verarbeitung eines einzelnen Benutzers im Vordergrund, während der Umgang mit mehreren Personen optional ist.

4.2.2 Fahrzeugzählung

Die Anwendung soll Fahrzeuge im Videostream erkennen können, mithilfe eines Objekterkennungsmodells. Es soll eine virtuelle Linie im Bild definiert werden, welche relativ zur Fahrstrecke der Fahrzeuge liegt (horizontal oder vertikal). Die Position der erkannten Fahrzeuge soll verfolgt werden und ein Zähler wird erhöht, sobald ein Fahrzeug die Linie in einer definierten Richtung überquert.

4.2.3 Gesichtsmaskierung

Die Anwendung soll Gesichter im Videostream erkennen können. Um erkannte Gesichter deutlich sichtbar zu machen, wird eine Box um das erkannte Objekt gezeichnet. Danach folgt die Maskierung der Gesichter durch Verpixelung oder Weichzeichnen der entsprechenden Bildbereiche. Die Darstellung des maskierten Videostreams erfolgt dabei in Echtzeit. Die Anwendung soll dabei mindestens ein Gesicht erkennen und maskieren können.

4.3 Nicht-funktionale Anforderungen

Neben den funktionalen Anforderungen spielen nicht-funktionale Anforderungen eine wichtige Rolle. Sie legen grob fest, wie gut die Anwendungen ihre Aufgaben erfüllen sollen. Genauere Zielwerte für Bildrate oder Latenz werden in Kapitel 6 „Evaluation“ festgelegt (Siehe 6.3).

Die folgenden Anforderungen gelten für alle Anwendungen:

Echtzeitfähigkeit und Latenz:
Alle Programme sollen reibungslos funktionieren und mindestens 15 Bilder pro Sekunde anzeigen. Die Bildrate sollte bei normaler Belastung konstant bleiben, ohne dass es zu starken Einbrüchen kommt. Die Verzögerung zwischen Kameraaufnahme und Darstellung der Ergebnisse soll minimal sein. Der Nutzer soll die Resultate in Echtzeit wahrnehmen, ohne dass sich Verzögerungen oder Inkonsistenzen bemerkbar machen.

Ressourcennutzung:
Es sollte sichergestellt werden, dass die CPU- und Speicherauslastung in einem Bereich bleibt, der für einen dauerhaften Betrieb ohne Instabilität sorgt. Dabei soll der Raspberry Pi nach längerer Zeit nicht überhitzen oder in eine CPU-Drosselung geraten. Ressourcen bleiben bei mehreren Prozessen ordnungsgemäß verteilt und blockieren nicht das System.

Robustheit:
Die Anwendungen sollen über eine längere Zeit stabil funktionieren, ohne dass sie abstürzen. Außerdem sollen sie in alltäglichen Situationen (normale Beleuchtung, mäßige Bewegung, unterschiedliche Hintergründe) verlässliche Ergebnisse erzielen. Darüber hinaus sollte die Erkennung gegenüber Verdeckungen, leichtem Wackeln und unterschiedlichen Distanzen tolerant sein.

Bedienbarkeit:

Alle Programme sollen mit nur wenigen Schritten gestartet werden können, etwa mit einem Skript oder einem Befehl. Die Ergebnisse werden verständlich dargestellt, durch gezeichnete Boxen, Schlüsselpunkte und Statusanzeigen im Videobild oder im Terminal.

Datenschutz:

Die Daten sollen lokal verarbeitet werden und keine Cloudanbindung haben. Eine Speicherung von Bildern ist nur bewusst möglich und klar gekennzeichnet. Insbesondere sollen die Ergebnisse der Maskierung zuverlässig sein.

4.4 Zielsetzung des Systemdesign

Das Systemdesign verfolgt drei zentrale Ziele, die im Folgenden erläutert werden:

Echtzeitverarbeitung des Videostreams:

Als Erstes soll der Videostream der Kamera in Echtzeit verarbeitet werden können, sodass der Benutzer eine direkte Rückmeldung seiner Bewegung oder der Ereignisse im Video erhält. Dazu ist ein durchgängiger „Pipeline“-Ansatz erforderlich, der Bildaufnahme, Vorverarbeitung, Inferenz, Nachverarbeitung und Darstellung sinnvoll verbindet.

Auslagerung der KI-Inferenz auf den AI-HAT+:

Als Zweites wird es angestrebt, dass die rechenintensive Ausführung der neuronalen Netze auf den AI-HAT+ ausgelagert wird. Der Raspberry Pi 5 dient lediglich als Steuereinheit und übernimmt Vor- und Nachverarbeitungsaufgaben, sowie die Darstellung der Ergebnisse. Dadurch sollen die Ressourcen der CPU geschont werden, um höhere Bildraten zu ermöglichen.

Gemeinsame Architektur für alle Anwendungen:

Das dritte Ziel ist es, die Architektur so zu gestalten, dass alle Anwendungen auf einer ähnlichen Datenverarbeitungskette basieren, welche aus Kameraeingang, Vorverarbeitung, Inferenz, Nachverarbeitung und Darstellung besteht. Die Unterschiede zwischen den Anwendungen sollen in der Auswahl des Modells und der Auswertung des Modells liegen, nicht in den verschiedenen Strukturen. Dadurch werden die Entwicklung, Vergleichbarkeit und Wiederverwendung von Komponenten erleichtert.

4.5 Konzeption der gemeinsamen Systemarchitektur

Obwohl die drei Anwendungen unterschiedliche Aufgaben lösen, sollen sie auf einer gemeinsamen Systemarchitektur basieren. Wie im vorherigen Abschnitt genannt, erleichtert es die Entwicklung, den Vergleich und eine mögliche Wiederverwendung der Komponenten. Der Kern ist eine „Videopipeline“ (Videoverarbeitungskette). Dabei wird jedes einzelne Bild in einer gleichen Abfolge von Verarbeitungsschritten durchlaufen. Auf hoher Abstraktionsebene lässt sich die Architektur folgendermaßen beschreiben:

1. **Videoaufnahme:** Die Kamera liefert einen kontinuierlichen Videostream an den Raspberry Pi. Es werden die aktuellen Bilder des Videostreams ausgelesen, welche die Kamera liefert. Die einzelnen Bilder dienen als Eingang für weitere Verarbeitung.

2. **Vorverarbeitung:** Die Rohbilder liegen zunächst im kameraspezifischen Ausgangsformat vor. Je nach KI-Modell gibt es Anforderungen zur Auflösung und zum Farbraum. Der Schritt der Vorverarbeitung passt die Bilder an diese Anforderungen. Die Bilder werden auf die geeignete Eingangsauflösung skaliert (1280 × 720 Pixel, sofern das Modell keine anderen Vorgaben macht). Als Nächstes folgt die Umwandlung in den benötigten Farbraum (RGB, falls keine Vorgaben existieren). Gegebenenfalls ist auch eine Normalisierung der Pixelwerte nötig.

3. **KI-Inferenz auf dem AI-HAT+:** Das vorbereitete Bild wird nun an den AI-HAT+ übertragen. In diesem Schritt wird das zuvor geladene Modell ausgeführt und das vorbereitete Bild eingegeben. Dabei ist wichtig zu verstehen, dass die Anwendung keine Details über die interne Funktionsweise des AI-HAT+ kennen soll. Die Schnittstelle soll modellunabhängig einheitlich sein. Die Bilder werden analysiert und der AI-HAT+ erzeugt strukturierte Ausgaben wie Klasseninformationen, Schlüsselpunkte oder Rahmenboxen (Bounding Boxes).

4. **Nachverarbeitung:** Die Roh-Ausgaben des Modells werden interpretiert und in nutzbare Informationen umgewandelt. Die Rahmenboxen und Schlüsselpunkte werden in das Bild gezeichnet. Eine Skalierung in die ursprüngliche Auflösung soll hier geschehen.

Gleichzeitig wird hier die anwendungsspezifische Logik eingebaut:

- Gestenerkennung: Die vom Modell gelieferten Schlüsselpunkte enthalten die benötigten Informationen, um anhand von vordefinierten Regeln eine diskrete Pose zu bestimmen.
- Gesichtsmaskierung: Das Modell liefert den Bildbereich des erkannten Gesichts, der anschließend verpixelt oder weichgezeichnet wird.

- Fahrzeugdetektion und -zählung: Das Modell erkennt die verschiedenen Fahrzeuge im Bild. Aus den Positionen wird abgeleitet, ob das Fahrzeug die virtuelle Linie überquert. In dem Fall wird der Zähler erhöht.

5. **Darstellung:** Die Ergebnisse der Nachverarbeitung werden je nach Anwendung im Videobild dargestellt, beispielsweise durch Skelett- und Schlüsselpunktdarstellungen, maskierte Bildbereiche und Rahmenboxen. Zusätzlich werden Textinformationen wie Pose, Zählerstand oder Debug-Ausgaben im Terminal ausgegeben. Das erzeugte Bild wird auf dem Monitor sichtbar und bildet die sichtbare Rückmeldung des Systems.

Die Architektur ist bewusst so ausgelegt, dass Kameraeingang, Vorverarbeitung, Inferenz, Nachverarbeitung und Bildausgabe in allen Anwendungen in ähnlicher Weise realisiert werden. Die Unterschiede ergeben sich hauptsächlich im verwendeten Modell und im jeweiligen Nachverarbeitungsschritt. Dadurch entsteht eine modulare Systemarchitektur, welche die Wiederverwendung von Komponenten ermöglicht.

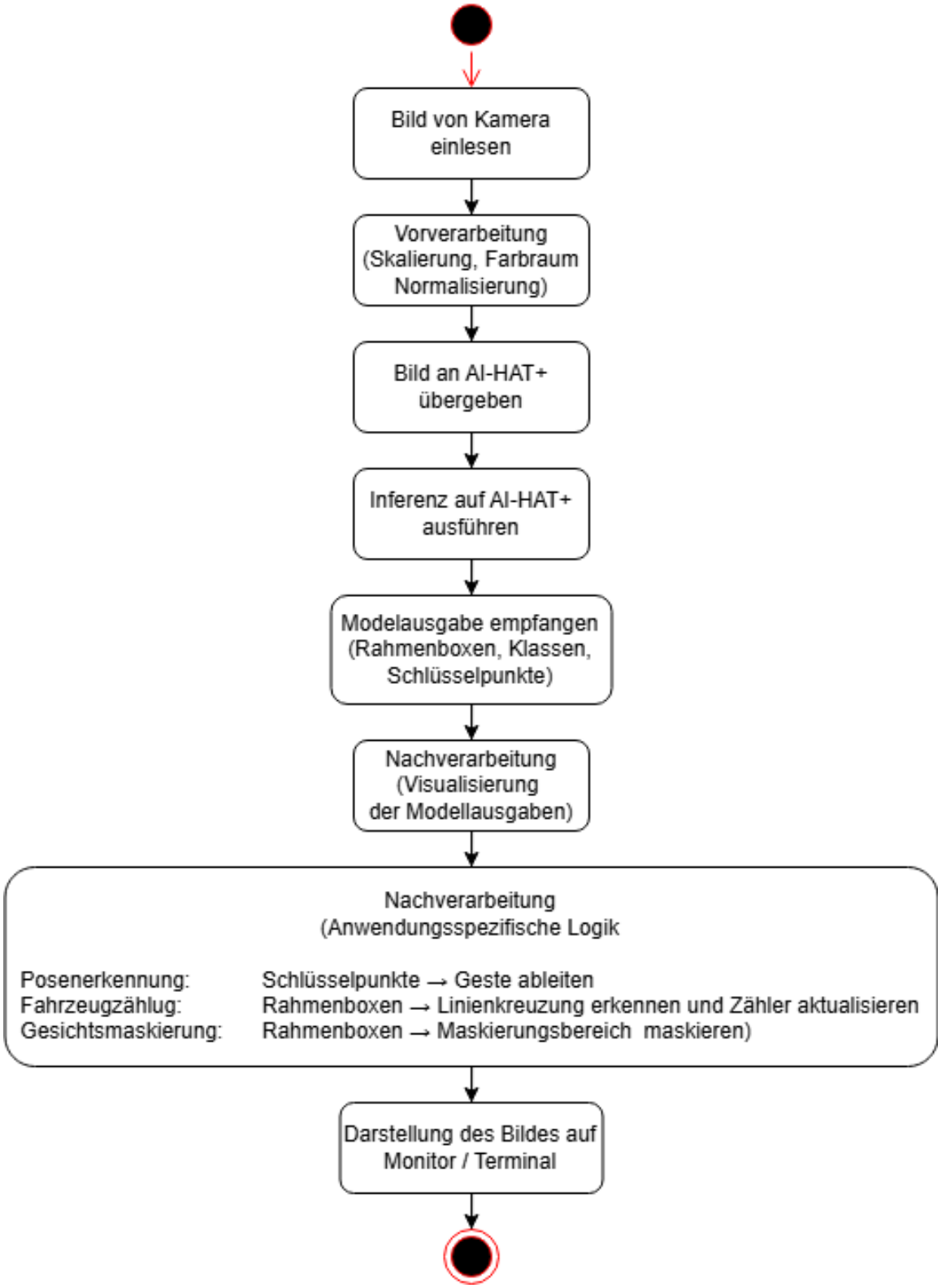


Abbildung 5: Aktivitätsdiagramm der Videoverarbeitungskette

4.6 Auswahl der KI-Modelle

Für die Umsetzung der Anwendungen wurden bewusst vortrainierte Modelle aus dem Hailo-Model-Zoo [21] ausgewählt. Der Model-Zoo bietet eine reichliche Anzahl an vortrainierten Modellen an, die im Zielformat für Hailo-Beschleuniger liegen. Zudem bietet er Umgebungen und Werkzeuge an, um eigene Modelle zu trainieren. Das Training von eigenen Modellen wäre im Rahmen dieser Arbeit zeitlich und organisatorisch nicht sinnvoll gewesen, da die Datenaufbereitung und das Modelltraining zeitintensiv sind und das eigentliche Ziel der Arbeit verschoben hätte. Die Auswahlkriterien waren daher die Verfügbarkeit als vorkompilierte HEF-Datei, die Kompatibilität mit dem AI-HAT+, sowie eine ausreichende Laufzeitperformance für Echtzeitanwendungen.

Für die Posenerkennung wurde das „YOLO8s Pose“-Modell [22] verwendet. Dieser schätzt Körperposen und liefert pro Person mehrere Gelenkpunkte. Das Modell eignet sich besonders gut, da es keine fertigen Klassen ausgibt, sondern nur die Positionen der Körperpunkte. Dadurch bleibt die Logik anpassbar, während das Modell die Schätzung übernimmt.

Die Fahrzeugdetektion und -zählung verwendet das „YOLOv8s“-Objekterkennungsmodell [23]. Dieser kann 80 verschiedene Klassen erkennen und klassifizieren. Da in der Anwendung mehrere Objekte erkannt werden sollen, wie zum Beispiel Autos, Motorräder oder Fahrräder, deckt das Modell die benötigten Klassen ab.

Das Gesichtserkennungsmodell „SCRFD_10G“ [24] wurde für die Gesichtsmaskierung verwendet. Das Modell ist für den Anwendungsfall besser geeignet als ein Gesichtsidentifikationsmodell, da nur die Gesichtsregion benötigt wird. Eine Identität des Gesichtes würde nicht das Datenschutz-Ziel dieser Anwendung verfolgen.

Insgesamt wurde die Modellauswahl so getroffen, dass sich alle drei Anwendungen mit der konzipierten Architektur implementieren lassen. Dabei liefern die Modelle strukturierte Ausgaben, die sich in der Nachverarbeitung weiterverarbeiten lassen. Die Unterschiede der Modelle liegen somit in den Ausgaben und weniger in der Systemintegration, wodurch ein Vergleich in der entwickelten Architektur möglich ist.

4.7 Annahmen und Abgrenzungen

Für die Konzeption der Anwendungen werden einige Annahmen getroffen und bewusste Abgrenzungen vorgenommen:

- Die Anwendungen sind als Prototypen zu verstehen und nicht als produktive Systeme. Bestimmte Aspekte wie umfassende Fehlerbehandlung werden nicht betrachtet.
- Das Training der verwendeten Modelle ist nicht Bestandteil dieser Arbeit. Es wird vorausgesetzt, dass geeignete, vortrainierte Modelle zur Verfügung stehen und in das für den AI-HAT+ notwendige Format vorkompiliert sind (Hailo-Model-Zoo).
- Die Anwendungen werden unter kontrollierten Bedingungen entwickelt und getestet, zum Beispiel mit einer einzelnen Person vor der Kamera oder einem definierten Kamerawinkel bei der Fahrzeugzählung. Extreme Szenarien wie schlechte Beleuchtung, starke Bewegungsunschärfe oder sehr viele Objekte im Bild werden nur eingeschränkt betrachtet.
- Die Arbeit konzentriert sich auf die bildbasierte KI-Verarbeitung. Weitere Sensoren werden nicht einbezogen.

Durch diese Abgrenzungen bleibt der Umfang der Arbeit überschaubar, während die gewählten Beispiele dennoch ein breites Spektrum typischer Edge-KI-Anwendungen abdecken.

5. Implementierung

In diesem Kapitel wird beschrieben, wie das im vorherigen Abschnitt entwickelte Design konkret in Software umgesetzt wurde. Für das Verständnis werden als Erstes die technischen Grundlagen erläutert. Im Anschluss wird auf die gemeinsame Basisimplementierung eingegangen, bevor die Besonderheiten der einzelnen Anwendungen erläutert werden.

5.1 Entwicklungsumgebung und verwendete Technologien

Die Implementierung erfolgt in Python, da diese Sprache eine gute Unterstützung für Bildverarbeitung, Skripting und die Anbindung externer Bibliotheken bietet. Auf der untersten Ebene stellt das Betriebssystem Raspberry Pi OS die notwendigen Gerätetreiber zur Verfügung, um die Kamera und den AI-HAT+ anzusprechen. Die Treiber, Multimedia-Frameworks und Bibliotheken werden im Folgenden besprochen.

5.1.1 GStreamer als Videopipeline

GStreamer ist ein Multimedia-Framework [25], mit dem sich Videodaten in Pipelines verarbeiten lassen. Eine Pipeline besteht aus einer Reihe von Elementen [26, 27], die jeweils eine klar definierte Aufgabe übernehmen. Somit können Datenverarbeitungsschritte miteinander verbunden. Jedes Element besitzt sogenannte „Pads“ [28]. Das sind Eingangs- und Ausgangsanschlüsse (src und sink), über die Daten von einem Element zum nächsten fließen. Somit entsteht eine durchgehende Datenstrecke.

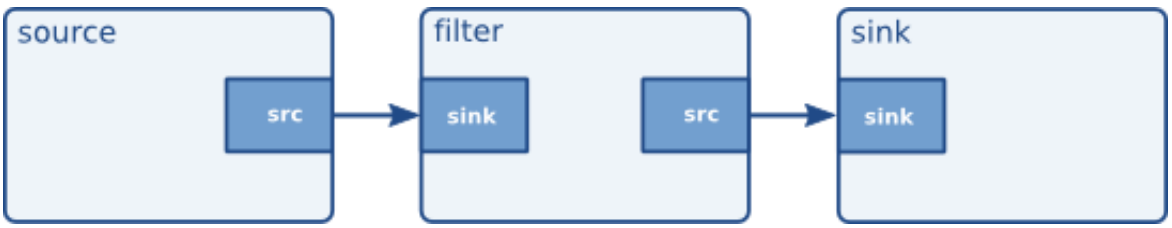


Abbildung 6: GStreamer, Elemente, Pads, Pipeline [26]

Die Pipeline wird in Python aufgebaut und gesteuert. Die einzelnen Elemente werden zu einer festen Reihenfolge miteinander verbunden. Die Kamerabilder fließen kontinuierlich durch

diese Kette. In den Hailo-Referenzanwendungen wird der AI-HAT+ über herstellerspezifische GStreamer-Elemente in die Pipelines integriert [29]. Ein Element (hailonet) übernimmt in der Regel die Inferenz und ein weiteres Element (hailooverlay) kann die Visualisierung der Erkennungsergebnisse unterstützen. Diese Hailo-Elemente fügen sich wie normale GStreamer-Bausteine in die Pipeline ein, sodass Kamera, Inferenz und Ausgabe in einem durchgehenden Datenfluss verbunden werden können.

Für die anwendungsspezifische Logik müssen die Anwendungen Zugriff auf die Modellausgaben haben. Dafür benutzt diese Arbeit das Konzept der „Pad-Probe“ [30] in GStreamer. Eine Pad-Probe ist ein „Abhörpunkt“ an einem Pad eines Elements. Somit kann eine Callback-Funktion an einem bestimmten Pad registriert werden. Typischerweise liegt dies am Ausgangspad des Elements, das die Modellausgaben bereitstellt. Das heißt, dass die Callback-Funktion aufgerufen wird, sobald ein Bild dieses Pad passiert. Somit kann bei jedem Bild die Funktion aufgerufen werden und die benötigten Informationen abrufen.

5.1.2 Hailo-Softwareumgebung

Damit der AI-HAT+ als KI-Beschleuniger genutzt werden kann, wird eine spezifische Hailo-Softwareumgebung benötigt. Im Wesentlichen besteht die Umgebung aus drei Bausteinen:

Der Hailo-Treiber [31] sorgt dafür, dass der AI-HAT+ vom Betriebssystem erkannt wird und als Gerät zur Verfügung steht. Der Treiber übernimmt den Datentransfer zwischen dem Arbeitsspeicher des Raspberry Pi und dem Speicher des Hailo-Chips. In der Implementierung wird der Treiber nicht direkt angesprochen, jedoch ist er die Voraussetzung, damit andere Bibliotheken auf den Beschleuniger zugreifen können.

Die HailoRT-Runtime [32, 33] ist die Laufzeitumgebung, die direkt mit dem Hailo-Chip kommuniziert. Sie lädt die HEF-Modelle in den Chip und führt die Inferenz durch. In den Anwendungen dieser Arbeit wird HailoRT überwiegend indirekt über GStreamer-Elemente genutzt, da die Hailo-Plugins intern auf HailoRT zurückgreifen.

Das Hailo-SDK [34] ist das Entwicklerpaket und stellt verschiedene Werkzeuge bereit (z. B. Modellkonvertierungswerkzeuge). Die Referenzbeispiele [35] aus dem SDK dienten als Vorlage für den Aufbau der eigenen GStreamer-Pipeline.

5.1.3 Python-Bibliotheken für Bildverarbeitung und Numerik

In den Anwendungen kamen mehrere Python-Bibliotheken zum Einsatz.

Open Source Computer Vision Library (OpenCV) [36] ist eine verbreitete Bibliothek für Bildverarbeitung und Computer Vision. Für die entwickelten Anwendungen stellt sie Funktionen zur Skalierung, Filterung, Farbkonvertierung und mehr bereit.

NumPy [37] ist die Standardbibliothek für numerische Berechnungen in Python. Für die Anwendungen stellt sie Datenstrukturen für Vektor- und Matrixoperationen zur Verfügung.

Die Kombination aus GStreamer, Hailo-Softwareumgebung und Python-Bibliotheken bildet die technische Grundlage für die folgenden Abschnitte der Implementierung.

5.2 Gemeinsame Basisimplementierung

In der gemeinsamen Basisimplementierung wird in der Funktion build_pipeline die Pipeline gebaut. Es wird eine Kette aus Kameraquelle (`rpicamsrc` für den Raspberry-Pi-Kamerastack oder alternativ `v4l2src` für Video4Linux2), Konvertierungs- und Skalierungsschritten, einem Hailo-Inferenz-Element und einem Video-Sink erstellt. Dabei werden Breite, Höhe und Bildrate der Frames zentral festgelegt (1280×720 Pixel). Nach dem Aufbau werden die Hailo-spezifischen Elemente (`hailonet`, `hailooverlay` und `hailofilter`) aus der Pipeline geholt und bei Bedarf weiter konfiguriert.

```
FRAME_WIDTH  = 1280
FRAME_HEIGHT = 720
FRAMERATE    = 30

def build_pipeline(app_name: str = "name", use_rpicam: bool = True):


    if use_rpicam:
        source = (
            "rpicamsrc name=src "
            f"! video/x-raw,width={FRAME_WIDTH},height={HEIGHT},"
            f"framerate={FRAMERATE}/1"
        )
    else:
        source = (
            "v4l2src device=/dev/video0 name=src "
            f"! video/x-raw,width={WIDTH},height={HEIGHT},"
            f"framerate={FRAMERATE}/1"
        )

    pipeline_desc = f"""
        {source}
        ! videoconvert
        ! videoscale
        ! video/x-raw,format=RGB
        ! queue
        ! hailonet name=hnet hef-path={hef_path}
        ! queue
        ! hailofilter name= hfilter so-path={so_path}
        ! queue
        ! hailooverlay name=overlay
        ! queue
        ! identity name=identity_callback
        ! autovideosink sync=false
    """

    return Gst.parse_launch(pipeline_desc)
```

Codeblock 1: Quellcode Pipeline bauen

```
def configure_hailo_elements(pipeline):
    hnet    = pipeline.get_by_name("hnet")
    overlay = pipeline.get_by_name("overlay")

    # Beispiel: Eigenschaften könnten hier gesetzt werden
    # hnet.set_property("batch-size", 1)
    # overlay.set_property("draw-scores", True)
```

Codeblock 2: Quellcode Hailo-Elemente holen und konfigurieren

Die eigentliche KI-Inferenz wird mit der Hilfe der GStreamer-Elemente `hailonet` und `hailooverlay` realisiert. Die Elemente, sowie Hailo-Runtime werden als dynamische Bibliotheken (.so-Dateien) von der Hailo-SDK bereitgestellt und in das System eingebunden. Das Element `hailonet` übernimmt das Laden des jeweiligen HEF-Modells in den Hailo-Chip sowie die Ausführung der Inferenz. Die vorverarbeiteten Bilder werden als Eingabe an `hailonet` übergeben. Dort werden sie verarbeitet und die Modell-Ausgaben, wie zum Beispiel Rahmenboxen oder Schlüsselpunkte, werden als Metadaten an den jeweiligen Videobuffer gehängt (`hailofilter`). Nun liest das Element `hailooverlay` die Metadaten aus und zeichnet die entsprechenden Informationen direkt in das Videobild ein, z. B. Rahmen um erkannte Fahrzeuge oder Gesichter.

Am Ende der GStreamer-Pipeline steht ein Video-Sink-Element wie `autovideosink` oder `waylandsink`. Diese Elemente sind dafür zuständig, um den resultierenden Videostream im Anzeigefenster des Raspberry Pi auszugeben. Außerdem wird in allen Anwendungen eine Pad-Probe am Ausgabepad (`src`-Pad) angehangen. Diese Pad-Probe ruft pro Bild eine Rückruf-Funktion (`app_callback`) auf. In dieser Funktion ist der Zugriff auf den aktuellen Videobuffer und die angehängten Metadaten möglich. Die anwendungsspezifische Logik, wie zum Beispiel die Posenklassifikation, wird auf Grundlage dieser Information in der Rückruf-Funktion umgesetzt. Gleichzeitig wird hier eine Funktion eingebaut, um die Metriken wie Bildrate, Latenz und CPU-Auslastung aufzuzeichnen.

```
def app_callback(pad, info, user_data):
    buffer = info.get_buffer()
    if buffer is None:
        return Gst.PadProbeReturn.OK

    detections = []
    try:
        roi = hailo.get_roi_from_buffer(buffer)
        detections = roi.get_objects_typed(hailo.HAILO_DETECTION)
    except Exception:
        detections = []

    #Anwendungsspezifische Logik hier

    #Metriken

    return Gst.PadProbeReturn.OK
```

Codeblock 3: Quellcode Callback-Funktion

Die Pipeline erhält ein `identity`-Element, damit die Pad-Probe sicher platziert werden kann. Das `identity`-Element gibt die Bilder neutral weiter, ohne sie zu verändern. Sie dient dem Zweck, einen stabilen Ankerpunkt für die Pad-Probe bereitzustellen. Da Sink-Elemente von Pipeline zu Pipeline unterschiedlich sein können, wäre es ohne das Element schwieriger, die Probe zuverlässig an derselben Stelle zu platzieren. Nachdem die Pipeline erstellt wurde, wird das `identity`-Element über seinen Namen `identity_callback` gesucht. Anschließend wird das src-Pad ausgelesen und dort eine Pad-Probe registriert, die bei jedem Bild die Funktion `app_callback` ausführt. Im Codeblock 4 wurde dieses Vorgehen implementiert.

```
def build_pipeline(app_name: str = "name", use_rpicam: bool = True):
    (...)

        pipeline = Gst.parse_launch(pipeline_desc)

        identity = pipeline.get_by_name("identity_callback")
        src_pad = identity.get_static_pad("src")
        src_pad.add_probe(Gst.PadProbeType.BUFFER, app_callback,
None)

    return pipeline
```

Codeblock 4: Quellcode identity-Element holen und anhängen

Der eigentliche Programmstart erfolgt über die `main`-Funktion. In der Funktion wird die Videopipeline aufgebaut und eine GLib-Mainloop gestartet, sodass die Pipeline kontinuierlich ausgeführt wird. Die eigentliche Struktur der Pipeline wird in einer eigenen Funktion gekapselt, sodass die drei Anwendungen jeweils als eigenständige Programme umgesetzt sind, jedoch alle dem gleichen grundlegenden Pipeline-Aufbau folgen.

```
import gi
from gi.repository import Gst, GObject
def main():
    Gst.init(None)
    loop = GObject.MainLoop()
    pipeline = build_pipeline()
    pipeline.set_state(Gst.State.PLAYING)

    try:
        loop.run()
    except KeyboardInterrupt:
        pass
    finally:
        pipeline.set_state(Gst.State.NULL)

    (...)

if __name__ == "__main__":
    main()
```

Codeblock 5: Quellcode Main-Funktion

Durch die Kombination aus Hailo-SDK und GStreamer können Kamerabilder eingelesen, auf den AI-HAT+ verarbeitet und das Ergebnis angezeigt werden. In den folgenden Unterabschnitten wird auf die anwendungsspezifischen Unterschiede eingegangen.

Die Abbildung 7 zeigt noch einmal die Pipeline mit ihren Elementen:

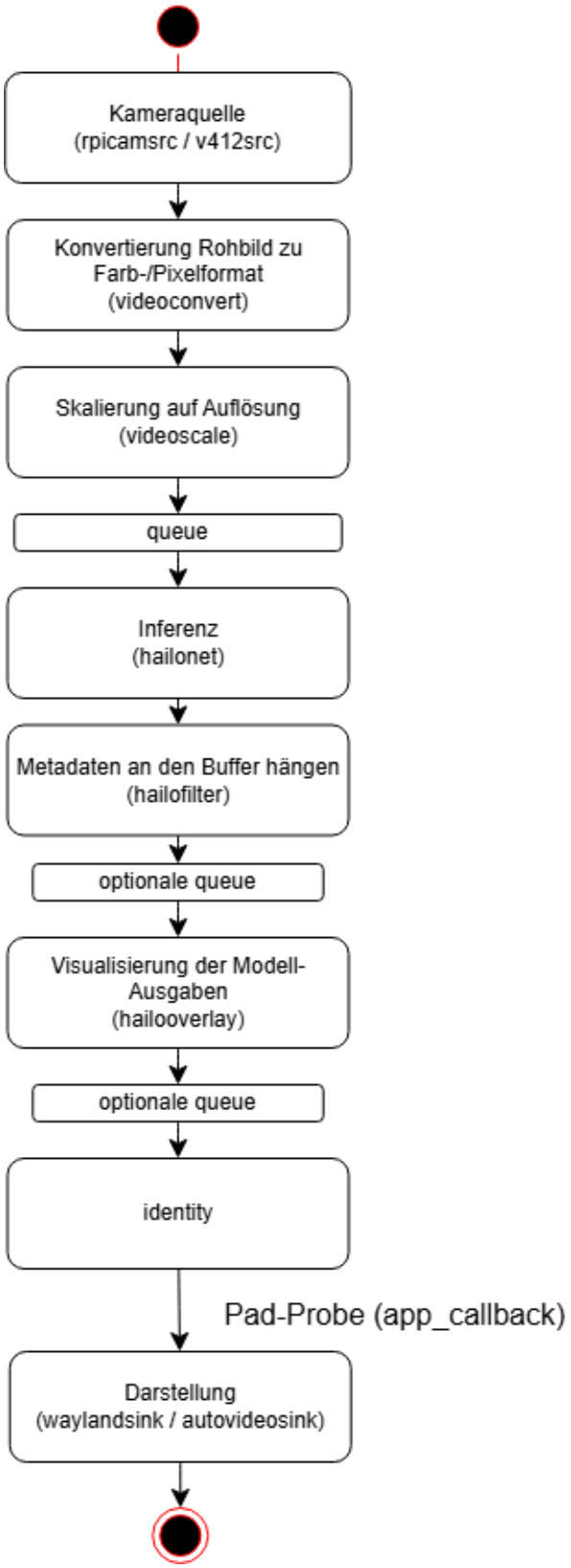


Abbildung 7: Aktivitätsdiagramm Ablauf der Videopipeline

5.3 Implementierung der Gestenerkennung

Die Gestenerkennung auf Basis der Körperpose verwendet das Modell „YOLO8s Pose“, das für jede erkannte Person im Bild eine Menge von Gelenkpunkten liefert. In der Implementierung werden diese Schlüsselpunkte zunächst in Pixelkoordinaten des ursprünglichen Kamerabildes zurückgerechnet. Anschließend werden sie so miteinander verbunden, dass eine einfache Skelettdarstellung entsteht, die im Video visualisiert wird.

Als Erstes werden die Rohdaten von der Kamera geliefert. Das Modell erwartet jedoch eine Eingabegröße von 640×640 Pixel. Das Bild wird proportional skaliert, sodass das Seitenverhältnis gleich bleibt und die fehlenden Bereiche mit Rändern aufgefüllt werden (Padding). Ein „Strecken“ des Bildes würde es verzerren und die Erkennung verschlechtern.

Im `hailonet`-Element wird das Modell geladen und der Hailo-Chip führt die Inferenz aus. Danach dekodiert das `hailofilter`-Element die Tensors und führt die Koordinaten vom Netzraum zurück in den Bildraum mithilfe der Funktion `filter_letterbox` (aus der Nachverarbeitungsbibliothek libyolov8pose_postprocess.so). Die fertigen Metadaten `HAILO_DETECTION` und `HAILO_LANDMARKS` werden an den GStreamer-Buffer angehängt und können mit der Funktion `hailo.get_roi_from_buffer(buffer)` gelesen werden.

Am Ende zeichnet `hailooverlay` noch die Boxen und die Schlüsselpunkte, um ein Skelett zu erhalten. Aus dem GStreamer-Buffer wird nun die beste Person und deren Schlüsselpunkte genommen, um die Pose zu klassifizieren. Da die Punkte noch relativ zur Rahmenbox normalisiert sind, müssen sie in Bildpixel umgerechnet werden:

1. Relativ in Rahmenbox → relativ im Bild

$$x_{relativ\ zum\ Bild} = p.x() \cdot bbox.width() + bbox.xmin()$$

$$y_{relativ\ zum\ Bild} = p.x() \cdot bbox.width() + bbox.xmin()$$

2. Relativ im Bild → Bildpixel

$$x_{Bildpixel} = x_{relativ\ zum\ Bild} \cdot W$$

$$y_{Bildpixel} = y_{relativ\ zum\ Bild} \cdot H$$

```
roi = hailo.get_roi_from_buffer(buffer)
detections = roi.get_objects_typed(hailo.HAILO_DETECTION)

best_kps, best_conf = None, 0.0
for det in detections:
    if det.get_label() != "person":
        continue
    bbox = det.get_bbox()
    conf = det.get_confidence()
    lmarks = det.get_objects_typed(hailo.HAILO_LANDMARKS)
    if not lmarks:
        continue

    pts = lmarks[0].get_points()
    kps_xyc = []
    for p in pts:
        x = (p.x()*bbox.width() + bbox.xmin()) * W
        y = (p.y()*bbox.height()+ bbox.ymin()) * H
        kps_xyc.append((x, y, conf))

    if conf > best_conf:
        best_conf, best_kps = conf, kps_xyc
```

Codeblock 6: Quellcode Umrechnung der Gelenkpunkte in Bildpixel

Um die Posen zu klassifizieren, müssen einige Regeln programmiert werden. Für `HANDS_UP` müssen die Handgelenke deutlich über der Nase und Schulterhöhe liegen. Für die erste Regel benötigen wir die y-Koordinaten der Handgelenke. In Bildkoordinaten liegt der Ursprung oben links. Das heißt, dass für die Handgelenke ein höherer Wert zurückgegeben wird, obwohl sie im Bild tief liegen (neutrale Position). Für die Programmierung der Regeln heißt das, dass der Wert der Handgelenke kleiner sein muss als der Nasen Wert. Dies gilt auch für die zweite Regel, die besagt, dass die Handgelenke über den Schultern sein müssen. Damit sichergestellt werden kann, dass die Handgelenke deutlich über der Nase und den Schultern sind, werden die Koordinaten jeweils um 10% der Bildhöhe erhöht.

```
#lwy:        y-Wert des linken Handgelenks
#rwy:        y-Wert des rechten Handgelenks
#nose_y:     y-Wert der Nase
#mean_sh:    Mittelwert der Schultern

up_margin = 0.10*H
if (lwy < nose_y - up_margin and rwy < nose_y - up_margin) or \
   (lwy < mean_sh - up_margin  and rwy < mean_sh - up_margin):
    return "HANDS_UP"
```

Codeblock 7: Quellcode HANDS_UP Regeln

T_POSE wird erkannt, sobald Handgelenke und Ellenbogen auf Schulterhöhe sind oder weit nach außen gehen. Um sicherzugehen, dass die Handgelenke nicht exakt auf der Schulterhöhe liegen müssen, wird ein vertikales Toleranzband definiert. Das Band lässt Abweichungen von bis zu 18% der Bildhöhe zu. Zudem wird verglichen, ob die Höhe der Handgelenke auf der Höhe der Schultern liegt (innerhalb des Toleranzbands). Um zu schauen, ob die Arme ausgestreckt sind, wird das Maximum aus 60% der Schulterbreite und 15% der Bildbreite genommen. Dieser Wert wird mit den x-Werten der Handgelenke verglichen, um zu schauen, ob sie seitlich weit genug von den Schultern entfernt sind.

```
#lwy/rwy:        y-Wert des linken/rechten Handgelenks
#torso_w:        Breite des Torsos (ungefähre Schulterbreite)
#lsy/rsy:        y-Wert der linken/rechten Schulter
#lsx/rsx:        x-Wert der linken/rechten Schulter

tol_y_level = 0.18*H
out_min_x   = max(0.6*torso_w, 0.15*W)

left_lvl  = abs(lwy - lsy) < tol_y_level
right_lvl = abs(rwy - rsy) < tol_y_level

left_out  = abs(lwx - lsx) > out_min_x
right_out = abs(rwx - rsx) > out_min_x

if left_lvl and right_lvl and (left_out or right_out):
    return "T_POSE"
```

Codeblock 8: Quellcode T_POSE Regeln

Als Letztes wird ARMS_DOWN erkannt, sobald die Handgelenke deutlich unter der Schulterhöhe liegen. Um herauszufinden, ob die Handgelenke unter der Schulterhöhe sind, werden die jeweiligen Handgelenke mit dem Mittelwert der Schultern verglichen. Ein Schwellwert wurde auf 35% der Torsohöhe unterhalb der mittleren Schulterhöhe definiert, da dies die besten Ergebnisse in den Tests erzielte. Die Regel erwies sich trotzdem als zu großzügig, da seitlich ausgestreckte Arme teilweise als ARMS_DOWN erkannt wurden. Daher wurde eine zweite Bedingung ergänzt, die die seitliche Nähe der Arme zum Körper prüft. Ein Handgelenk gilt als nah am Körper, wenn der horizontale Abstand zur jeweiligen Schulter kleiner ist als ein neu definierter Schwellwert, der sich aus dem Maximum aus 50% Schulterbreite und 10% Bildbreite zusammensetzt.

```
below_sh_thr    = mean_sh + 0.35*torso_h
hands_below     = (lwy > below_sh_thr) and (rwy > below_sh_thr)

near_body_max_x= max(0.5*torso_w, 0.10*W)
hands_near      = (abs(lwx - lsx) < near_body_max_x) and (abs(rwx -
rsx) < near_body_max_x)

if hands_below and hands_near:
    return "ARMS_DOWN"
```

Codeblock 9: Quellcode ARMS_DOWN Regeln

Die app_callback-Funktion ist das Herzstück der Anwendung. Sie wird für jedes ankommende Bild aufgerufen und führt die Auswertung aus. Vereinfacht lässt sich der Ablauf so beschreiben: Die Kamera liefert als Erstes ein Videobild. Das Hailo-Modul führt auf dem Bild die Inferenz aus und in der Nachverarbeitung werden die Metadaten wie „Person“ und Schlüsselpunkte der Pose an das Bild angehängt. Nun wird app_callback aufgerufen, liest die Daten aus und klassifiziert die Pose. Im Anschluss werden die letzten Bilder geglättet, um Flackern zu vermeiden. Das Bild läuft weiter durch die Pipeline und hailooverlay zeichnet das Skelett. So bleibt app_callback leichtgewichtig und kann die Daten analysieren, ohne sie selbst zu rendern.

```
def app_callback(pad, info, user_state):
    (...)
    best_kps = extract_best_person_keypoints(detections, W, H)

    pose = classify_pose(best_kps, W, H)
    user_state.pose_hist.append(pose)
    smoothed = Counter(user_state.pose_hist).most_common(1)[0][0]
    user_state.current_pose = smoothed

    if PRINT_EVERY_FRAME:
        print(smoothed)

    return Gst.PadProbeReturn.OK
```

Codeblock 10: Quellcode Callback-Funktion Posenerkennung

5.4 Implementierung Fahrzeugzählung

Die Implementierung des Fahrzeugzählers besteht aus der Objektdetektion mit dem Modell „YOLOv8s“ und der GStreamer-Pipeline. Wie bei den anderen Anwendungen lädt `hailonet` das HEF-Modell. Auf der Modellebene wird damit eine robuste Mehrklassen-Detektion erreicht. Die Anwendung erkennt ausschließlich die Fahrzeugklassen „car“, „bus“, „truck“ und „motorcycle“.

In der GStreamer-Pipeline gelangt das Bild als Erstes zur Farbkonvertierung und Skalierung. Das Modell erwartet dabei ein Seitenverhältnis von 640×640 Pixel im RGB-Farbraum. Auch hier bleibt das Seitenverhältnis bei der Skalierung erhalten und die Ränder werden aufgefüllt (Padding). Das Element `hailonet` lädt das Bild für die Inferenz und danach zu `hailofilter` für die Nachverarbeitung. Die Ausgabe des Modells sind Klassen und die dazugehörigen Box-Koordinaten. Als Nachverarbeitungsbibliothek wird libyolo_hailortpp_postprocess.so genutzt, die auch mit der Funktion `filter_letterbox` die Boxen zurück ins Originalbild rechnet. Da das Modell insgesamt 80 Klassen erkennen kann, wird das Standardvisualisierungselement `hailooverlay` deaktiviert, da ausschließlich Rahmenboxen für Fahrzeuge gezeichnet werden. Das Darstellen der Linie, Boxen und Texte übernimmt `cairooverlay` und `textoverlay` (bereitgestellte GStreamer Elemente).

Ein wesentlicher Punkt für die Zählgenauigkeit ist die Stolperfallen-Logik in der Rückruf-Funktion. Der Rückruf `app_callback` wird pro Bild aufgerufen und ist die zentrale Komponente in der Anwendung. Als Erstes werden die gewünschten Klassen gefiltert. Somit wird sichergegangen, dass ausschließlich die Fahrzeugklassen „`car`“, „`bus`“, „`truck`“ und „`motorcycle`“ erkannt werden. Danach werden Rahmenboxen in Pixelkoordinaten gerechnet und ein Ankerpunkt definiert. Der Ankerpunkt ist wichtig, um die Position des Fahrzeugs zu ermitteln. Anstatt die Mitte der Box zu verwenden, nutzt die Zähllogik die untere Mitte der Box (Bodenkontaktpunkt). Bei mehreren Versuchen stellte sich heraus, dass dieser Ankerpunkt Zählfehler bei hohen Fahrzeugen minimiert und eine stabilere Überquerung der Linie liefert. Danach wird geprüft, auf welcher Seite der Linie sich der Ankerpunkt befindet. Die Punkte des aktuellen Bildes werden mit dem vorherigen Bild verglichen und es wird geschaut, ob ein Seitenwechsel mit derselben Objekt-ID erfolgt ist. Ein Wechsel von links nach rechts wird als „`LR`“ gezählt und umgekehrt als „`RL`“.

```
CALLBACK app_callback(pad, info, state):

    (...)

    carsnow ← 0

    FÜR jede detektion IN dets:
        klasse ← detektion.klasse
        WENN klasse ∉ {car, bus, truck, motorcycle}:
            WEITER

        (x1, y1, x2, y2) ← rechne Rahmenbox in Pixel
        ankerpunkt ← (cx = (x1+x2)/2, cy = y2)

        s_prev ← state.last_side[tid]
        s_now  ← seite_relativ_zur_tripline(cx, cy)

        WENN s_prev existiert UND s_prev ≠ s_now UND
        (jetzt − state.last_cross_ts[tid]) ≥ MIN_CROSS_INTERVAL:
            WENN s_prev < s_now: state.passed_LR += 1
            SONST:               state.passed_RL += 1
            state.last_cross_ts[tid] ← jetzt

        state.last_side[tid] ← s_now
        state.last_seen[tid] ← jetzt
        füge (x1,y1,x2,y2, klasse) zu state.draw_boxes hinzu
        carsnow += 1

    aktualisiere Textoverlay mit:
        "Cars: carsnow | Passed LR: state.passed_LR  RL:
state.passed_RL"

    RETURN OK
```

Codeblock 11: Pseudocode Callback-Funktion Fahrzeugzählung

5.5 Implementierung der Gesichtsmaskierung

Das Ziel dieser Implementierung ist die Echtzeitmaskierung von Gesichtern in einem Videostream. Aufgrund praktischer Probleme bei der Bildmanipulation, musste die GStreamer-Struktur angepasst werden.

5.5.1 Abgrenzung der Gesichtsmaskierung gegenüber den anderen Anwendungen

Während die anderen Anwendungen die Programmlogik über eine Pad-Probe an einem `identity`-Element realisieren, wird in dieser Anwendung bewusst auf eine einzelne Rückruffunktion (`app_callback` pro Bild) verzichtet. Der Grund ist, dass eine Verpixelung eine direkte Veränderung der Bilddaten erfordert und das „Verdecken“ des Gesichts mit `cairooverlay` nicht ausreicht. Allerdings werden Buffer von GStreamern oft als unveränderliche Zero-Copy-Buffer zur Verfügung gestellt, was bei Manipulationen zu Instabilitäten führen kann. In der Praxis traten bei Versuchen, den Buffer direkt zu verändern, Fehler wie „write map requested on non-writable buffer“ auf. Auch extreme Verzögerungen oder niedrige Bildraten traten bei vorherigen Versionen des Prototyps auf.

Der zentrale Entwurfspunkt der neuen Architektur ist die Trennung zwischen der Verarbeitung der Pipeline und der Bildmanipulation. In der überarbeiteten Pipeline wird ein `tee`-Element hinzugefügt. Dieser agiert wie ein Verteiler und trennt den Videostrom in zwei Datenpfade. Ein Pfad dient zur Darstellung des Kamerabildes, während der andere Pfad die KI-Inferenz auf dem Hailo-Chip hat. Dadurch wird verhindert, dass aufwendige Bildoperationen die Videopipeline blockieren. Würde die Maskierung direkt im GStreamer-Rückruf (`app_callback`) ausgeführt werden, könnte es im Video schnell zu Stau, Aussetzern oder „Einfrieren“ kommen.

Um dies zu vermeiden, wird das Kamerabild über ein `appsink`-Element an die Anwendung übergeben und dort in einem eigenen, schreibbaren Speicherbereich bearbeitet. Das Ergebnis wird anschließend über `appsrc` wieder in eine Ausgabe-Pipeline geführt. Die KI-Inferenz über den Hailo-Chip bleibt dabei unverändert und liefert weiterhin die Metadaten, welche über die Pad-Probe ausgelesen werden.

Die Abbildung 8 beschreibt die überarbeitete Pipeline mit allen Elementen:

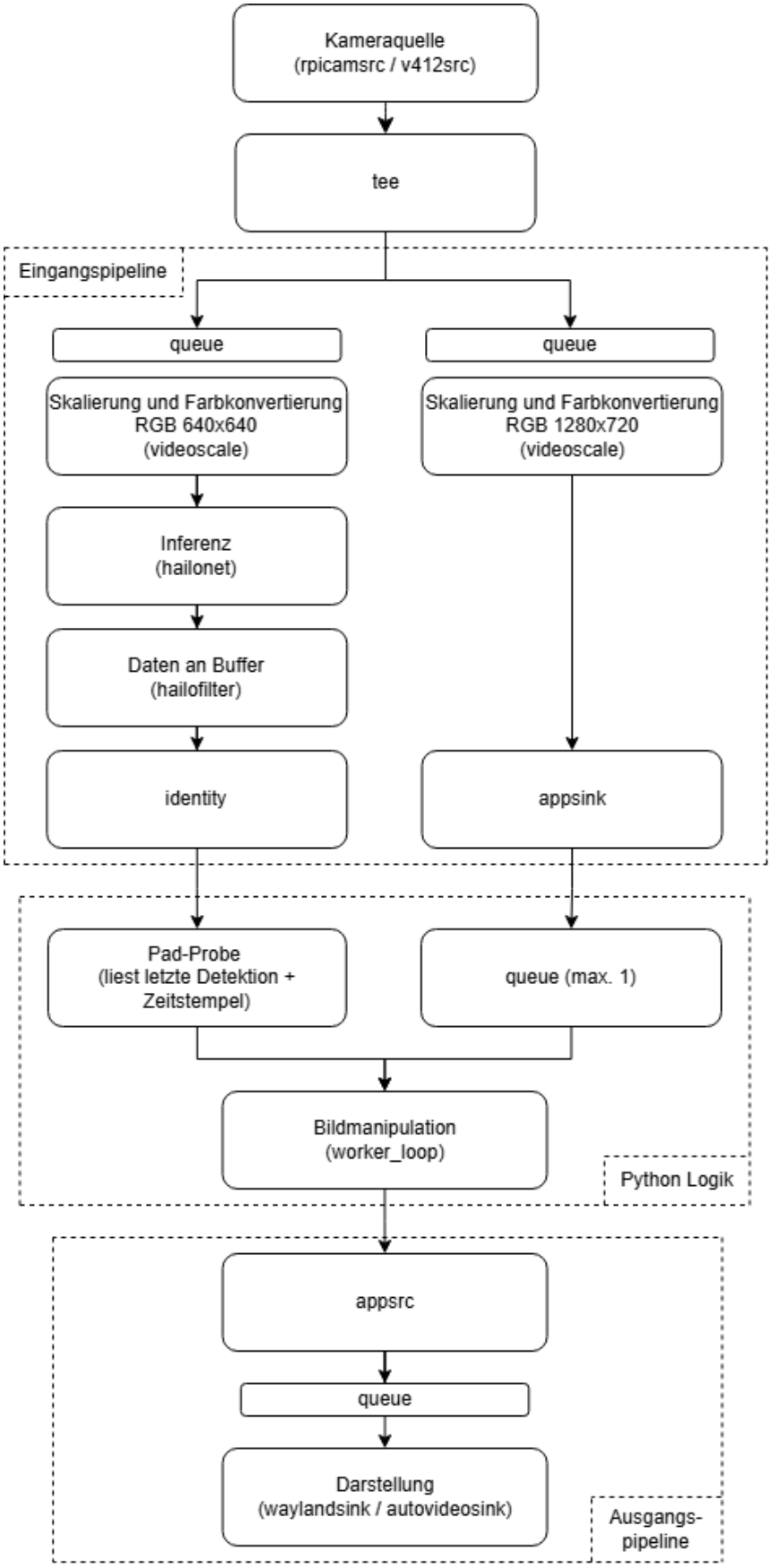


Abbildung 8: Architekturdiagramm der Gesichtsmaskierung Pipeline

5.5.2 Implementierung

Zu Beginn wird die Eingangsgröße des Gesichtsmodells „SCRFD_10g“ bestimmt. Die Gesichtserkennung liefert ihre Rahmenboxen im Koordinatensystem des Modell-Eingangs. Damit die Boxen korrekt auf die von uns vorgegebene Auflösung (1280×720 Pixel) übertragen werden können, liest das Programm die Netzgröße aus dem HEF-Format aus (640×640 Pixel).

Der Anzeige-Zweig der Pipeline führt die Kamerabilder in ein `appsink`-Element, damit die Anwendung die Bilddaten übernehmen kann. Die Bilder werden somit nicht in der Pipeline verarbeitet, sondern an die Anwendung übergeben. Mit der Rückruf-Funktion `on_preview_sample` kann ausschließlich das aktuelle Bild aus dem Buffer ausgelesen werden. Das Bild wird als NumPy-Array in eine kleine Warteschlange gelegt. Wie in der Abgrenzung erwähnt, findet in diesem Rückruf keine aufwendige Bildbearbeitung statt. Somit wird der Rückruf absichtlich so kurz gehalten wie möglich: das Bild übernehmen, das alte Bild verwerfen und ein neues Bild ablegen.

```
CALLBACK on_preview_sample(sample):
    frame ← lese aktuelles RGB-Bild aus sample
    frame ← kopiere Bild in lokales Array

    WENN warteschlange_voll:
        verwerfe altes Bild
    lege frame in Warteschlange

    RETURN OK
```

Codeblock 12: Pseudocode Callback-Funktion on_preview_sample

Der Inferenz-Zweig der Pipeline skaliert das Bild proportional auf die Eingangsgröße des Netzes (640×640) und die fehlenden Bildbereiche werden mit Rändern aufgefüllt (Padding). Anschließend wird es zum `hailonet-` und `hailofilter`-Element weitergeleitet. Die Rahmenboxen der erkannten Gesichter werden als Metadaten an den jeweiligen Buffer gehängt. Wie in den anderen Anwendungen wird eine Pad-Probe an das `scr`-Pad des `identity`-Elements registriert. Die Pad-Probe liest pro Buffer die Metadaten aus und speichert nur die aktuelle Liste der Rahmenboxen mit Zeitstempel. Auch hier gilt das gleiche Prinzip wie im Anzeige-Zweig: Die Pad-Probe soll nur Daten abgreifen und nicht blockieren.

Die tatsächliche Maskierung findet dann nicht im Rückruf statt, sondern im sogenannten Worker-Thread. Dieser funktioniert wie ein zusätzlicher Ausführungsstrang, der parallel zum Hauptprogramm läuft. Der Thread `worker_loop` wartet auf ein neues Kamerabild aus der

Warteschlange. Sobald ein neues Bild vorliegt, werden die gespeicherten Rahmenboxen genommen, und von den Netz-Koordinaten zurück in die Vorschau-Auflösung umgerechnet. Falls die Detektionen älter als 0,7 Sekunden sind, wird die Liste der Rahmenboxen geleert. Um das Gesicht zu maskieren, wird ein für die erkannte Gesichtsbox ein leicht vergrößerter Bereich berechnet und der Bildbereich im NumPy-Array verpixelt. Anschließend wird das bearbeitete Bild in einen neuen GStreamer-Buffer kopiert und über `appsrc` in eine separate Ausgabepipeline eingespeist. Das Ergebnis wird über `waylandsink` oder autovideosink angezeigt.

```
THREAD worker_loop():
    SOLANGE programm_läuft:
        frame ← warte blockierend auf neues frame aus warteschlange

        (dets, timestamp) ← hole zuletzt gespeicherte detektionen
        WENN dets zu älter als 0.7s:
            dets ← leere liste

        FÜR jede detection in dets:
            box_net ← koordinaten im netz-system (net_w, net_h)
            box_preview ← rechne box_net auf 1280x720 um
            box_preview ← vergrößere box leicht
            box_preview ← begrenze box auf bildränder

            verpixle den bildbereich innerhalb box_preview im frame

        schiebe bearbeitetes frame in ausgabe (appsrc)
```

Codeblock 13: Pseudocode worker_loop Thread

Damit die Maskierung des Gesichts an der richtigen Stelle des Bildes erscheint, müssen die Boxen in die Vorschau-Auflösung zurückgerechnet werden. In den anderen Anwendungen passiert dieser Schritt durch die Funktion `filter_letterbox`. Diese Funktion ist jedoch oft in YOLO-Nachverarbeitungsbibliotheken zu finden. Da es sich hierbei um ein SCRFD-Modell handelt, wird die libscrfd.so-Bibliothek genutzt und eine eigene Funktion muss dafür definiert werden. Im Codeblock 14 übernimmt die Funktion `map_net_to_preview_letterbox` das Entfernen der Randbereiche und das Rückskalieren der ursprünglichen Auflösung.

```
def map_net_to_preview_letterbox(x1n, y1n, x2n, y2n, prev_w, prev_h,
net_w, net_h):

    scale = min(net_w / prev_w, net_h / prev_h)
    new_w = prev_w * scale
    new_h = prev_h * scale
    pad_x = (net_w - new_w) / 2.0
    pad_y = (net_h - new_h) / 2.0

    x1 = (x1n - pad_x) / scale
    x2 = (x2n - pad_x) / scale
    y1 = (y1n - pad_y) / scale
    y2 = (y2n - pad_y) / scale
    return x1, y1, x2, y2
```

Codeblock 14: Quellcode map_net_to_preview_letterbox Funktion

Bei der Maskierung des Gesichts handelt es sich um eine Block-Pixelung. Zunächst wird der erkannte Gesichtsbereich („ROI“, also „Region Of Interest“) aus dem Bild ausgeschnitten. Anschließend wird dieser Ausschnitt in gleich große Blöcke unterteilt. Für jeden Block wird eine einheitliche Farbe bestimmt, die sich aus dem Mittelwert der Pixel-Farben im Block ergibt. Die Blöcke werden mit den Farben gefüllt und das Gesicht wird unerkennbar. Im Anschluss wird der Ausschnitt des Gesichtes zurück ins das Originalbild geschrieben. Die Stärke der Verpixelung lässt sich über die Blockgröße steuern.

```
Funktion VERPIXELN(bild, gesichts_box, blockgröße):
    (x1, y1, x2, y2) = gesichts_box

    roi = bild[y1:y2, x1:x2]

    Für y von 0 bis roi.höhe in Schritten von blockgroesse:
        Für x von 0 bis roi.breite in Schritten von blockgroesse:

            block = roi[y : y+blockgroesse, x : x+blockgroesse]
            farbe = MITTELWERT(block)
            block[:] = farbe

    bild[y1:y2, x1:x2] = roi
```

Codeblock 15: Pseudocode Bildmanipulation Funktion

5.6 Zusammenfassung

Die Implementierung setzt das entworfene Systemdesign konsequent um und zeigt, dass auf der gemeinsamen Plattform Raspberry Pi 5 mit AI-HAT+ unterschiedliche KI-Anwendungen realisiert werden können, ohne die Architektur jedes Mal neu zu erfinden. Alle Anwendungen folgen demselben Grundprinzip aus Kamerazugriff, Vorverarbeitung, Inferenz, Nachverarbeitung und Darstellung. Die Unterschiede konzentrieren sich auf das jeweils verwendete Modell und auf die Anwendungslogik in der Nachverarbeitung.

Bei der Anwendung zur Gesichtsmaskierung, trat jedoch ein spezielles Problem auf. Die Bildmanipulation erwies sich als deutlich empfindlicher als die reine Auswertungslogik der anderen Anwendungen. Sie erforderte eine unterschiedliche Pipeline-Architektur, während bei den anderen KI-Beispielen die Verarbeitung direkt über die Rückruf-Funktion erfolgte.

Insgesamt wird es bestätigt, dass es möglich ist, eine ähnliche Grundstruktur zu implementieren. Gleichzeitig wird aber klar, dass bestimmte Aufgaben wie Bildmanipulation eine Anpassung der Pipeline-Architektur erfordern, um eine robuste und flüssige Darstellung zu gewährleisten.

6. Evaluation

In diesem Kapitel wird untersucht, wie leistungsfähig die entwickelten Anwendungen auf der Plattform aus Raspberry Pi 5 und AI-HAT+ sind. Das Ziel der Evaluation ist es, die Leistungsfähigkeit der Systeme im Hinblick auf Echtzeitfähigkeit, Erkennungsqualität und Ressourcenauslastung zu bewerten und die ursprünglichen Ziele aus der Einführung zu überprüfen. Die Betrachtung erfolgt aus quantitativer Sicht, über Bildraten und CPU-Auslastung, sowie aus qualitativer Sicht durch das Beobachten des Verhaltens in typischen Szenarien.

6.1 Zielsetzung der Evaluation

Die Evaluation verfolgt zwei zentrale Fragestellungen. Zum einen soll anhand von definierten Kriterien geprüft werden, ob die entwickelte Plattform in der Lage ist, die KI-Anwendungen in einer flüssig wahrgenommenen Geschwindigkeit auszuführen (Bildrate). Zum anderen soll untersucht werden, ob die Ergebnisse der Modelle in den gewählten Szenarien ausreichend robust und zuverlässig sind (Latenz, CPU-Auslastung), um die jeweiligen Anwendungsfälle sinnvoll abzudecken.

Darüber hinaus dient die Evaluation dazu, Unterschiede zwischen den Anwendungen sichtbar zu machen. Einige Szenarien, wie die Posenerkennung, sind naturgemäß komplexer und ressourcenintensiver als andere. Die gewonnenen Messwerte und Beobachtungen liefern somit auch Hinweise darauf, welche Art von Edge-KI-Anwendungen sich besonders gut für den Einsatz auf dem Raspberry Pi 5 mit AI-HAT+ eignet und wo Grenzen der Plattform erkennbar werden.

6.2 Versuchsaufbau

Die Messungen wurden direkt auf dem Raspberry Pi 5 durchgeführt, auf dem auch die Anwendungen implementiert wurden. Der AI-HAT+ war während aller Tests eingebunden und führte die Inferenz aus. Die Kamera war in fester Position, um reproduzierbare Bedingungen zu schaffen. Die Videobilder wurden von der Kamera aufgenommen.

Für die Posenerkennung fanden die Tests in einem Innenraum mit gleichmäßiger Beleuchtung statt. Die Testperson positionierte sich in variierender Distanz zur Kamera und führte unterschiedliche Körpergesten vor. Der Zweck bestand darin, die vorgegebenen Gesten sowie Grenzfälle wie zum Teil verdeckte Handgelenke oder schnelle Bewegungen hervorzurufen.

Die Erkennung von Fahrzeugen wurde anhand von Videosequenzen untersucht, in denen Fahrzeuge eine festgelegte Linie im Bild überqueren. Je nach Aufbau konnten hierfür entweder reale Aufnahmen oder aufgenommene Videos mit der Kamera abgespielt werden. Entscheidend war, dass Fahrzeuge in verschiedenen Abständen, Geschwindigkeiten und Blickwinkeln die virtuelle Linie überquerten, um das Verhalten des Zählers nachvollziehen zu können.

Die Gesichtsmaskierung wurde ebenfalls in einem Innenraum getestet. Hier lag der Fokus darauf, ob das Gesicht zuverlässig erkannt und maskiert wird, auch wenn sich die Person bewegt oder teilweise seitlich zur Kamera steht.

6.3 Bewertungsmetriken

Für die Bewertung wurden mehrere Kennzahlen herangezogen. Eine zentrale Rolle spielt die Bildrate, also die Anzahl von verarbeiteten Bildern pro Sekunde. Sie gibt einen direkten Hinweis darauf, ob eine Anwendung als „echtzeitnah“ wahrgenommen wird. „Etwa 16 bis 18 Bilder pro Sekunde kann unser Gehirn für ein flüssiges Zusammenspiel der Bilder verarbeiten“ [38], während Kinofilme oder Videoportale bis zu 30 Bilder die Sekunde liefern [38]. Laut der Quelle [39] kann das menschliche Auge 14 bis 16 Bilder pro Sekunde wahrnehmen. Der Zielwert liegt somit bei 15 bis 30 Bildern pro Sekunde bei einer Auflösung von 1280×720 Pixeln, um ein flüssiges Videobild darzustellen.

Ein weiterer Indikator für eine „echtzeitnahe“ Wahrnehmung ist die Latenz der Pipeline. Sie gibt an, wie lange es durchschnittlich dauert, bis das nächste fertige Bild durch die Schleife kommt. Eine steigende Latenz zeigt, dass die Pipeline stockt.

Ergänzend wurde die Auslastung der CPU und des RAM beobachtet, um abzuschätzen, inwieweit der Raspberry Pi 5 noch Reserven für weitere Aufgaben hätte oder bereits an seine Grenzen stößt. Eine CPU-Auslastung von über 80% ist grundsätzlich nicht kritisch, jedoch kann es bei längeren Laufzeiten zu höheren Temperaturen kommen.

Als Letztes wurde noch die CPU-Temperatur gemessen, um zu schauen, ob sich die Temperatur trotz höherer Auslastung in einem normalen Bereich bewegt. Der Raspberry Pi beginnt laut Hersteller ab 80 °C zu drosseln [40], um eine Überhitzung zu vermeiden.

Zur Beurteilung der Erkennungsqualität wurden je nach Anwendung unterschiedliche Kriterien herangezogen. Bei der Posenerkennung wurde betrachtet, ob die vom System ausgegebene Pose mit der tatsächlich gezeigten Pose übereinstimmt und wie stabil die Erkennung bei Bewegungen oder leichten Veränderungen der Position ist. Bei der Gesichtsmaskierung war relevant, ob die Gesichter im Bild erkannt und anonymisiert wurden und wie häufig Fehl- oder Nicht-Erkennungen auftraten. In der Fahrzeugdetektion und -zählung wurden die gezählte Anzahl der Fahrzeuge mit der tatsächlichen Anzahl der Linienüberquerungen verglichen.

Neben diesen quantitativen Metriken flossen auch subjektive Eindrücke in die Bewertung ein, zum Beispiel, ob das System träge wirkt oder ob der Lüfter deutlich lauter wird.

6.4 Ergebnisse der Posenerkennung

In den Tests zur Posenerkennung zeigte sich, dass das System bei einer Auflösung von 1280×720 Pixeln in der Lage ist, den Videostream kontinuierlich zu verarbeiten und die Körperpose der im Vordergrund stehenden Person zu schätzen. Die Bildrate lag im Durchschnitt bei 30 Bildern pro Sekunde. Somit konnten Bewegungen nahezu in Echtzeit nachvollzogen werden. Die Pipeline-Latenz lag konstant bei rund 35 Millisekunden und zeigt, dass die Pipeline stabil ist. Die Auslastung der CPU lag durchschnittlich bei 71%, obwohl der Hailo-Chip die KI-Berechnung übernimmt. Durch Vor- und Nachverarbeitungsschritte muss der Raspberry Pi viel Arbeit leisten und lässt durch die Auslastung wenig Raum für andere Programme. Eine RAM-Nutzung von ca. 270MB bedeutet, dass die Anwendung sehr speicherschonend ist. Die CPU des Raspberry Pi erreicht eine Temperatur von 73°C. Das ist warm, aber im Regelfall noch sicher. Mit 72 °C ist der CPU nah an der Drosselungsgrenze, wodurch der Lüfter deutlich hörbar ist.

Zeit in Minuten	Bilder pro Sekunde	Pipeline-Latenz (ms)	CPU-Temp. (°C)	CPU-Auslastung (%)	RAM (MB)
10	30	34.7	72.15	74.20	271.89
20	30	35.2	71.33	74.10	271.56
30	30	35.0	71.05	74.17	271.28
40	30	34.7	71.05	74.17	271.28
50	30	35.1	71.60	74.12	270.85
60	29.99	35.1	71.71	74.02	270.45

Tabelle 2: Metriken des Testlaufs für Posenerkennung

Die Skelettdarstellung im Videobild machte die Arbeit des Modells gut sichtbar. Die Anwendung lieferte in typischen Situationen sinnvolle Ergebnisse. Deutlich angehobene Arme wurden zuverlässig erkannt, und neutrale Haltungen wurden korrekt von den anderen Posen unterschieden. Schwächen traten hauptsächlich in Randbereichen auf, etwa bei zügigen Bewegungen, seitlichen Perspektiven oder wenn der Körper teilweise aus dem Bild herausragte. In einigen Fällen traten instabile Klassifizierungen oder kurzzeitige Fehlanzeigen auf.

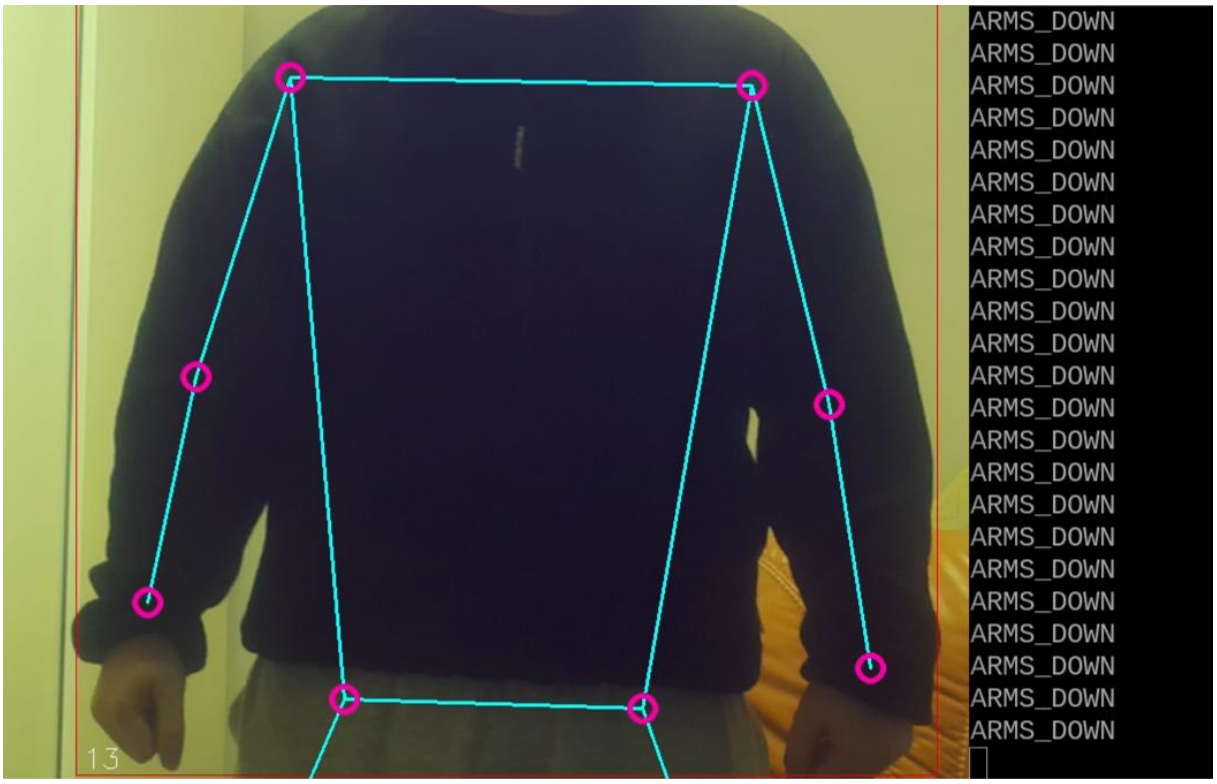


Abbildung 9: Videobild Posenerkennung

6.5 Ergebnisse der Fahrzeugzählung

Bei Tests zur fahrzeugbasierten Detektion und Zählung arbeitete das System stabil im Echtzeitbereich. Die mittlere Bildrate lag konstant bei 30 Bildern pro Sekunde. Die gemessene Pipeline-Latenz betrug etwa 30 Millisekunden über die ganze Testzeit. Damit reagierte die Zähl-Logik sichtbar flüssig. Die Systemlast blieb niedrig. Die CPU-Auslastung bewegte sich zwischen 27-28%. Das deutet darauf hin, dass die rechenintensive Objektdetektion effektiv auf den Hailo-Beschleuniger ausgelagert ist. In der Praxis bleibt somit Raum für ergänzende Aufgaben wie Protokollierung oder Export der Zähldaten. Die CPU-Temperatur lag bei etwa 67°Cund bleibt somit deutlich unter dem Drosselwert von 80°C. Beim Arbeitsspeicher zeigt sich wie bei den anderen Anwendungen ein unauffälliges Bild. Die Auslastung liegt bei rund 252 MB und bleibt konstant. Das spricht gegen unbeabsichtigtes Anwachsen von Puffern und lässt genügend RAM für Erweiterungen.

Zeit in Minuten	Bilder pro Sekunde	Pipeline-Latenz (ms)	CPU-Temp. (°C)	CPU-Auslastung (%)	RAM (MB)
10	30.00	31.1	66.65	26.70	256.28
20	30.00	30.4	67.20	26.50	253.95
30	30.00	30.6	66.83	26.40	253.17
40	29.99	29.7	66.65	26.35	252.76
50	30.00	30.2	66.54	26.30	252.39
60	30.00	31.0	66.56	26.37	252.14

Tabelle 3: Metriken des Testlaufs für Fahrzeugzählung

Die Darstellung der Boxen und der Linie macht die Arbeitsweise des Systems nachvollziehbar. In typischen Verkehrsszenen mit moderater Perspektive und stabilem Bild liefert die Anwendung robuste Ergebnisse. Die vorbeifahrenden Fahrzeuge werden zuverlässig erkannt und der Zähler inkrementiert, sobald ein erkanntes Fahrzeug die Linie überquert. Einige Schwächen zeigen sich in Randbereichen. Kleine oder weit entfernte Fahrzeuge werden manchmal nicht erkannt und dadurch nicht gezählt. Starke Teilverdeckungen durch das Überholen oder dichtes Auffahren erschweren die Zuordnung der Boxen. Zudem kann es bei extremen Kamerawinkeln oder bei schnellen Spurwechseln zu einem instabilen Seitenwechsel kommen.

Abbildung 10: Videobild Fahrzeugzählung [43]

6.6 Ergebnisse der Gesichtsmaskierung

In den Tests zur Gesichtsmaskierung zeigte sich, dass das System den Videostream, bei einer Auflösung von 1280×720 Pixel, stabil verarbeitet. Während des Tests traten keine Fehler oder erkennbare Abbrüche auf. Die Bildrate lag im Mittel bei 28 Bildern pro Sekunde, wodurch die Maskierung im laufenden Bild weitgehend flüssig wirkte. Die gemessene Pipeline-Latenz blieb im Durchschnitt bei 52 Millisekunden, was für eine Echtzeit-Maskierung ausreichend ist. Die CPU-Auslastung des Raspberry Pi lag im Mittel bei 60%. Ursache hierfür sind auch wieder die Vor- und Nachverarbeitungsschritte, wie Farbkonvertierung, Skalierung und die Maskierung der Bildbereiche. Damit beansprucht die Anwendung einen spürbaren Teil der Rechenleistung, jedoch lässt sie im Vergleich zur Posenerkennung etwas mehr Reserven für weitere Prozesse. Die RAM-Nutzung lag im Durchschnitt bei 158 MB, wodurch die Anwendung auch als speicherschonend zählt. Die CPU-Temperatur erreichte im Mittel etwa 72°C. Damit arbeitet das System im warmen Bereich, bleibt noch unterhalb der Drosselgrenze von 80°C.

Zeit in Minuten	Bilder pro Sekunde	Pipeline-Latenz (ms)	CPU-Temp. (°C)	CPU-Auslastung (%)	RAM (MB)
10	27.80	51.9	73.80	60.20	159.20
20	27.76	51.0	72.97	60.15	159.20
30	27.79	51.9	72.70	60.13	158.46
40	27.82	51.7	72.42	60.17	158.09
50	27.81	51.9	72.26	60.26	157.86
60	27.80	51.9	72.15	60.38	158.08

Tabelle 4: Metriken des Testlaufs für Gesichtsmaskierung

Die Maskierung im Videobild machte die Funktion der Gesichtserkennung sichtbar. Sobald ein Gesicht im Bild lag, wurde der entsprechende Bereich maskiert. In typischen Situationen wurden die Gesichter in Echtzeit erkannt und die Maskierung lag zuverlässig auf den Bildbereichen. Auch bei schnellen Kopfbewegungen oder teilweiser Verdeckung kam es selten zu Fehleranzeigen. Schwächen wurden nur in Momenten erkannt, in denen die Gesichtsregion kurzfristig abgegrenzt wurde.

Abbildung 11: Videobild Gesichtsmaskierung

6.7 Diskussion und Bewertung der Ergebnisse

Der Raspberry Pi 5 in Kombination mit dem AI-HAT+ hat sich im Rahmen dieser Arbeit als grundsätzlich geeignete Plattform für die Umsetzung von praxisorientierten KI-Anwendungen erwiesen. Die Evaluation hat gezeigt, dass in allen drei Anwendungen die Kamerabilder verarbeitet werden konnten und die Inferenz in nahezu Echtzeit durchgeführt werden konnte. Dazu waren die erzielten Bildraten und Reaktionszeiten ausreichend, um die Anwendung aus einer Benutzersicht als flüssig wahrzunehmen. Damit erfüllt die Plattform das Ziel, typische Aufgaben der Bildverarbeitung mit KI auf einem kompakten Edge-Gerät demonstrieren zu können.

Ein wesentlicher Vorteil der Plattform liegt in der klaren Aufgabenteilung zwischen dem Raspberry Pi und dem AI-HAT+. Der Raspberry Pi übernimmt die Steuerung der Anwendung, sowie die Anbindung der Kamera und die Auswertung der Modell-Ausgaben. Die rechenintensive Ausführung der neuronalen Netze übernimmt dabei der AI-HAT+. Durch die Auslagerung der Inferenz wird gezeigt, dass die CPU-Auslastung des Raspberry Pi moderat bleibt und somit weiterhin in der Lage ist, die restlichen Aufgaben wie Vorverarbeitung, Nachverarbeitung und Darstellung zuverlässig zu übernehmen. Die Tests zeigen auch, dass selbst bei längerem Betrieb, keine kritischen Einbrüche der Bildrate auftreten. Für Lehr- und Demonstrationszwecke, sowie für erste Prototypen im Bereich Edge-KI ist dies ein deutlicher Pluspunkt.

Ein weiterer positiver Aspekt ist, dass zwei der drei Anwendungen auf derselben technischen Grundlage aufgebaut werden konnten. Von der Kameraanbindung bis zur Inferenz und Ausgabe wurde die Basisstruktur der Anwendungen wiederverwendet. Dies vereinfacht die Entwicklung deutlich und erleichtert die Wiederverwendung der Struktur bei verschiedenen KI-Szenarien auf derselben Plattform. Die Gesichtserkennung bildet hier die Ausnahme, da sie aufgrund der Bildmanipulation von der einheitlichen Struktur abweicht. Jedoch zeigt die Pipeline-Latenz, dass die komplexere Struktur nur einen geringen Unterschied macht, und dass die Anwendung praxistauglich ist. Aus der Sicht einer Abschlussarbeit mit praxisorientiertem Schwerpunkt war die gemeinsame Basisstruktur trotzdem wertvoll, da keine Anwendung vollständig neu aufgebaut werden musste.

Gleichzeitig werden in den Tests aber Grenzen der Plattform sichtbar. Zum einen sind die verfügbaren Rechenressourcen trotz KI-Beschleuniger begrenzt. Komplexere Modelle, höhere Auflösungen oder zusätzliche Verarbeitungsschritte können die Bildrate schnell reduzieren. Die

Modellwahl und die Konfiguration der Videopipeline müssen daher aufeinander abgestimmt sein, um einen Kompromiss zwischen Erkennungsqualität und Echtzeitfähigkeit zu erreichen. Zudem hängen Ergebnisse von den Rahmenbedingungen ab. Eine schlechte Beleuchtung oder eine sehr komplexe Szene führen schneller zu instabilen Erkennungen.

Auch aus der Entwicklersicht gibt es bei der Plattform einige Einschränkungen. Die Nutzung des AI-HAT+ setzt die bereitgestellte Softwareumgebung von Hailo voraus, und viele Schritte können nur innerhalb dieses Ökosystems erfolgen. Für die vorliegende Arbeit wurde auf vortrainierte und bereits kompilierte Modelle zurückgegriffen, die den Aufwand deutlich erleichterten. Für andere Modelle ist die Konvertierung zum HEF-Modell nötig und wäre zusätzlicher Aufwand, der über den Rahmen dieser Arbeit hinausgeht.

Insgesamt lässt sich die Plattform aus Raspberry Pi 5 mit AI-HAT+ dennoch als passend für die in dieser Arbeit verfolgten Ziele bewerten. Sie bietet genügend Rechenleistung, um unterschiedliche KI-Anwendungen in Echtzeit zu demonstrieren.

7. Zusammenfassung und Ausblick

Zum Abschluss werden die Ergebnisse zusammengefasst und im Hinblick auf die Ziele dieser Arbeit bewertet. Zunächst werden die Schritte des Entwicklungsprozesses sowie die Erkenntnisse zusammengefasst. Dazu gehört die Eignung des Raspberry Pi 5 mit AI-HAT+ als Plattform für praxisnahe KI-Beispiele. Anschließend werden mögliche Weiterentwicklungen und offene Fragen aufgezeigt, die sich für zukünftige Arbeiten und Projekte ergeben.

7.1 Zusammenfassung der Arbeit

In dieser Arbeit wurde die Entwicklung und Implementierung praxisorientierter KI-Beispiele auf dem Raspberry Pi 5 mit dem AI-HAT+ untersucht. Der Ausgangspunkt dieser Arbeit war die Frage, inwieweit sich die Plattform für die Entwicklung von praxisorientierten KI-Anwendungen eignet. Um diese Frage zu beantworten, wurden drei unterschiedliche Anwendungen realisiert: eine Posenerkennung auf Basis der Körperpose, eine Gesichtserkennung zur Maskierung von Gesichtern sowie eine Fahrzeugerkennung zur Fahrzeugzählung.

Ein zentrales Ergebnis der Arbeit ist, dass diese Plattformarchitektur sich in der Praxis bewährt. Die drei umgesetzten Anwendungen konnten weitgehend alle auf derselben Grundlage realisiert werden. Auch wenn die Anwendung der Gesichtsmaskierung von der entworfenen Grundarchitektur abweicht, wurden mehrere Bausteine der Grundarchitektur wiederverwendet, um die Anwendung erfolgreich zu realisieren. Unterschiede ergaben sich sonst nur im eingesetzten Modell und in der Logik der Nachverarbeitung.

Die Evaluation hat gezeigt, dass die Anwendungen mit Bildraten arbeiten, die vom Benutzer als flüssig wahrgenommen werden. Die Transition von Pose zu Pose wird in typischen Situationen nahezu in Echtzeit vom System wahrgenommen. Die Gesichtsmaskierung ist in der Lage, Gesichter in einem Videobild zu erkennen und unmittelbar zu maskieren. Die Fahrzeugerkennung kann Fahrzeuge zählen, die eine virtuell definierte Linie im Bild überqueren. Die Umsetzung praxisorientierter KI-Beispiele auf dem Raspberry Pi 5 mit AI-HAT+ wurde somit erreicht.

Allerdings wurden im Rahmen der Evaluation auch Grenzen deutlich. Die Erkennungsqualität hängt stark von den Rahmenbedingungen ab. Eine schlechte Beleuchtung, verdeckte Objekte,

stark seitliche Ansichten oder sehr komplexe Szenen führen zu instabileren Ergebnissen. Dies betrifft sowohl die Robustheit der Posenerkennung als auch die Fahrzeugzählung und die Gesichtsmaskierung. Hier zeigt sich der typische Konflikt von Edge-KI: Einerseits sollen Modelle klein und effizient genug sein, um auf beschränkter Hardware zu laufen, andererseits wird dadurch die maximale Erkennungsleistung begrenzt.

Insgesamt kann festgehalten werden, dass die entwickelte Plattform für den Einsatz in Lehre, Demonstration und prototypischer Entwicklung geeignet ist. Sie macht es möglich, verschiedene KI-Anwendungen direkt auf einem kleinen, kostengünstigen System ausführbar zu machen, ohne auf eine Cloud-Infrastruktur angewiesen zu sein. Die entwickelten Anwendungen bilden somit nicht nur ein Ergebnis dieser Arbeit, sondern auch eine Grundlage, auf der zukünftige Projekte und Erweiterungen aufbauen können.

7.2 Weiterentwicklungsmöglichkeiten und offene Fragen

Aus den Ergebnissen der Arbeit ergeben sich mehrere Ansatzpunkte für zukünftige Erweiterungen. Eine Möglichkeit besteht darin, robustere oder speziell angepasste Modelle zu integrieren, die besser mit schwierigen Beleuchtungssituationen oder komplexeren Szenen zurechtkommen.

Ein weiterer Schritt wäre die Weiterentwicklung der bestehenden Anwendungen. Anstatt Posen zu erkennen, könnte die Anwendung um Gesten erweitert werden, wie zum Beispiel „Winken“. Ebenso wäre es sinnvoll, die Erkennung der Posen auf mehrere Personen auszuweiten und Strategien zu entwickeln, wie mit überlappenden Personen umgegangen wird. Für die Fahrzeugzählung wäre eine manuelle Definition der virtuellen Linie innerhalb der Anwendung interessant. Die Gesichtsmaskierung ist ein spezieller Fall, da die Verarbeitungskette von der ursprünglichen Struktur abweicht. Es wäre sinnvoll, die Maskierung weiter in die Verarbeitungskette einzubinden oder nach Alternativen zu suchen, die eine Verpixelung direkt in der Pipeline ermöglicht, ohne dass Leistungsverluste entstehen.

Auf der Plattformebene ergeben sich technische Fragen, da sich die vorliegende Arbeit auf eine einzelne Kamera und einen KI-Beschleuniger konzentriert. Eine interessante Erweiterung wäre die Verarbeitung von mehreren Videostreams oder der Einsatz von weiteren Sensoren. Somit könnte die Plattform extremer auf ihre Grenzen testen. Auch die Frage, ob sich mehrere KI-

Anwendungen gleichzeitig ausführen lassen, ohne die Echtzeitfähigkeit zu verlieren, bleibt offen.

Weiterhin wäre eine datengetriebene Evaluation denkbar, um die Modelle quantitativ zu bewerten, zum Beispiel in Form von Präzisions- und Recall-Messungen. Auch ein Vergleich mit alternativen Edge-KI-Plattformen oder KI-Beschleunigern könnte spannend sein, um die Leistungsfähigkeit des Raspberry Pi 5 mit AI-HAT besser einordnen zu können.

Abschließend kann festgehalten werden, dass die Plattform für die Entwicklung und Implementierung von praxisorientierten KI-Beispielen geeignet ist, jedoch auch Entwicklungsmöglichkeiten bietet, die im Rahmen dieser Arbeit nicht möglich waren. Die Abschlussarbeit bietet jedoch einen Ausgangspunkt, auf dem zukünftige Projekte aufbauen können, um die genannten Entwicklungsmöglichkeiten zu realisieren.

Literaturverzeichnis

[1] Cole Stryker und Eda Kavlakoglu. IBM. Was ist künstliche Intelligenz (KI)? (besucht am 03.12.2025). URL: https://www.ibm.com/de-de/think/topics/artificial-intelligence

[2] Dave Bergmann. IBM. Was ist maschinelles Lernen? (besucht am 03.12.2025). URL: https://www.ibm.com/de-de/think/topics/machine-learning

[3] dogado. Starke KI. (besucht am 05.12.2025). URL: https://www.dogado.de/ki-lexikon/starke-ki

[4] Fangfang Lee. IBM. Was sind neuronale Netzwerke? (besucht am 03.12.2025). URL: https://www.ibm.com/de-de/think/topics/neural-networks

[5] Mesh Flinders und Ian Smalley. IBM. Was ist KI-Inferenz? (besucht am 04.12.2025). URL: https://www.ibm.com/de-de/think/topics/ai-inference

[6] IBM. Was ist Edge KI? (besucht am 04.12.2025). URL: https://www.ibm.com/de-de/think/topics/edge-ai

[7] Imagination Technologies. What is Edge AI? (besucht am 04.12.2025). URL: https://www.imaginationtech.com/what-is-edge-ai/

[8] NVIDIA. Entwickler-Kits und Module für eingebettete Systeme von NVIDIA Jetson. (besucht am 07.12.2025). URL: https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/

[9] Connect Tech Inc. Jetson TX2 Datasheet. (besucht am 07.12.2025). URL: https://connecttech.com/ftp/pdf/jetson_tx2_datasheet.pdf

[10] NVIDIA. Jetson TX1: A New Low-Power CUDA Platform for Deep Learning and Computer Vision. (besucht am 08.12.2025). URL: https://www.nvidia.com/content/tegra/embedded-systems/pdf/jetson_tx1_whitepaper.pdf

[11] NVIDIA. JetPack SDK. (besucht am 07.12.2025). URL: https://developer.nvidia.com/embedded/jetpack

[12] Coral (Google). Coral Accelerator Module. (besucht am 08.12.2025). URL: https://www.coral.ai/products/accelerator-module#description

[13] Coral (Google). Models and transfer learning. (besucht am 08.12.2025). URL: https://www.coral.ai/docs/edgetpu/models-intro#transfer-learning

[14] Raspberry Pi Ltd. AI HAT+. (besucht am 27.11.2025). URL: https://www.raspberrypi.com/documentation/accessories/ai-hat-plus.html

[15] Leela S. Karumbunathan. NVIDIA. NVIDIA Jetson AGX Orin Technical Brief. (besucht am 07.12.2025). URL: https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf

[16] NVIDIA. Jetson Modules. (besucht am 07.12.2025). URL: https://developer.nvidia.com/embedded/jetson-modules

[17] Coral (Google). Edge TPU benchmarks. (besucht am 09.12 2025). URL: https://www.coral.ai/docs/edgetpu/benchmarks/

[18] Eben Upton. Raspberry Pi Ltd. Introducing Raspberry Pi 5. (besucht am 29.11.2025). URL: https://www.raspberrypi.com/news/introducing-raspberry-pi-5/

[19] Raspberry Pi Ltd. Raspberry Pi Documentation. (besucht am 04.12.2025). URL: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

[20] Raspberry Pi Ltd. Processors BCM2712. (besucht am 04.12.2025). URL: https://www.raspberrypi.com/documentation/computers/processors.html#bcm2712

[21] Hailo AI. hailo_model_zoo. (besucht am 26.11.2025). URL: https://github.com/hailo-ai/hailo_model_zoo

[22] Hailo AI. HAILO8L Pose Estimation. (besucht am 26.11.2025). URL: https://github.com/hailo-ai/hailo_model_zoo/blob/master/docs/public_models/HAILO8L/HAILO8L_pose_estimation.rst2

[23] Hailo AI. HAILO8L Object Detection. (besucht am 26.11.2025). URL: https://github.com/hailo-ai/hailo_model_zoo/blob/master/docs/public_models/HAILO8L/HAILO8L_object_detection.rst

[24] Hailo AI. HAILO8L Face Detection. (besucht am 26.11.2025). URL: https://github.com/hailo-ai/hailo_model_zoo/blob/master/docs/public_models/HAILO8L/HAILO8L_face_detection.rst

[25] GStreamer Project. GStreamer. (besucht am. 26.12.2025). URL: https://gstreamer.freedesktop.org/

[26] GStreamer Project. Elements. (besucht am 26.12.2025). URL: https://gstreamer.freedesktop.org/documentation/application-development/basics/elements.html?gi-language=c

[27] GStreamer Project. Dynamic pipelines. (besucht am 27. 12. 2025). URL: https://gstreamer.freedesktop.org/documentation/tutorials/basic/dynamic-pipelines.html?gi-language=c

[28] GStreamer Project. Pads. (besucht am 26.12.2025). URL: https://gstreamer.freedesktop.org/documentation/application-development/basics/pads.html?gi-language=c

[29] Hailo Technologies Ltd. TAPPAS User Guide. (besucht am 27.12.2025). URL: https://f.hubspotusercontent30.net/hubfs/3383687/TAPPAS%20User%20Guide.pdf

[30] GStreamer Project. Probes. (besucht am 26.12.2025). URL: https://gstreamer.freedesktop.org/documentation/additional/design/probes.html?gi-language=c

[31] OnLogic. Hailo-8 AI Accelerator Integration. (besucht am 25.11.2025). URL: https://support.onlogic.com/product-documentation/components-and-expansion/hailo-8-ai-accelerator-integration

[32] Hailo AI. hailort. (besucht am 25.11.2025). URL: https://github.com/hailo-ai/hailort

[33] Hailo AI. HailoRT v5.2.0 Documentation. (besucht am 27.11.2025). URL: https://hailo.ai/developer-zone/documentation/hailort-v5-2-0/

[34] Raspberry Pi Ltd. How to set up the Raspberry Pi AI Kit with Raspberry Pi 5. (besucht am 22.11.2025). URL: https://www.raspberrypi.com/news/how-to-set-up-the-raspberry-pi-ai-kit-with-raspberry-pi-5/

[35] Hailo AI. hailo-rpi5-examples. (besucht am 10.12.2026). URL: https://github.com/hailo-ai/hailo-rpi5-examples

[36] GeeksforGeeks. OpenCV Overview. (besucht am 06.01.2026). URL: https://www.geeksforgeeks.org/computer-vision/opencv-overview/

[37] Python Wiki. NumPy. (besucht am 06.01.2026). URL: https://wiki.python.org/moin/NumPy

[38] IONOS. FPS – Framerates im TV, Kino und FPS beim Gaming. (besucht am 07.01.2026). URL: https://www.ionos.de/digitalguide/server/knowhow/fps/

[39] Wikipedia. Bildfrequenz. (besucht am 08.01.2026). URL: https://de.wikipedia.org/wiki/Bildfrequenz

[40] Alasdair Allan. Raspberry Pi Ltd. Heating and cooling Raspberry Pi 5. (besucht am 08.01.2026). URL: https://www.raspberrypi.com/news/heating-and-cooling-raspberry-pi-5/

[41] Conrad. Nvidia Super Developer Kit Jetson Orin Nano 8 GB 6 x 1.5 GHz. (besucht am 15.01.2026). URL: https://www.conrad.de/de/p/nvidia-super-developer-kit-jetson-orin-nano-8-gb-6-x-1-5-ghz-2998506.html

[42] Amazon. Google Coral Dev Board. (besucht am 15.01.2026). URL: https://www.amazon.de/Google-G950-01455-01-Coral-Entwicklungsplatine/dp/B07QF582TG?th=1

[42] Anuj Khandelwal. Vehicle Dataset Sample 2. (besucht am 15.01.2026). URL: https://www.youtube.com/watch?v=JqhdBCCUVyQ

Anhang

GitHub Repository des Projekts:

https://github.com/22anh03/ai-prototypes-raspberrypi5-aihat