Hochschule Aalen

Projektbericht
Studiengang : Machine Learning and Data Analytics

Deep Learning zur Kugelverfolgung auf Edge-Hardware: Trainingspipeline, Integration und Vergleich mit klassischer Bildverarbeitung

von

Lukas Kaczmarczyk

79758

Betreuender Professor: Prof. Dr. Winfried Bantel

Einreichungsdatum : 15. August 2025

Eidesstattliche Erklärung

Hiermit erkläre ich, **Lukas Kaczmarczyk**, dass ich die vorliegenden Angaben in dieser Arbeit wahrheitsgetreu und selbständig verfasst habe.

Weiterhin versichere ich, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, dass alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Königsbronn, 15.08.2025 

Ort, Datum Unterschrift (Student)

Kurzfassung

In dieser Arbeit wird das Problem der robusten Kugelverfolgung auf einer automatisch regelbaren Plattform untersucht. Herkömmliche Verfahren zur Objekterkennung, wie etwa farbbasiertes Thresholding oder Kantendetektion, sind zwar leichtgewichtig und ohne Trainingsdaten einsetzbar, zeigen jedoch in realen Umgebungen deutliche Schwächen bei wechselnden Lichtverhältnissen, variierenden Kugeltypen oder Hintergrundstörungen [13, 3].

Um diesen Einschränkungen zu begegnen, wurde ein vollständiger Deep-Learning-Workflow zur Kugelerkennung entwickelt und implementiert. Dieser umfasst ein automatisiertes System zur Datenerfassung und Labelgenerierung, eine You Only Look Once (YOLO)-kompatible Vorverarbeitung [1], das Training dreier moderner Objekterkennungsmodelle (YOLOv5, YOLOv8 [34], Real-Time DEtection TRansformer (RT-DETR) [40]) sowie deren Integration in ein Proportional–Integral–Derivative (PID)-geregeltes Jetson-Echtzeitsystem [10]. Die Modelle wurden mithilfe von Google Colab Pro auf NVIDIA T4 Graphics Processing Unit (GPU)s trainiert [18], um reproduzierbare, GPU-gestützte Ergebnisse zu erzielen.

Die Deep-Learning-basierten Ansätze zeigten im Test eine deutlich höhere Robustheit gegenüber Störungen sowie eine stabile Laufzeitperformance bei über 20 FPS. Die klassische Methode wurde zur Einordnung weiterhin betrachtet, zeigte jedoch erwartungsgemäß eine geringere Generalisierungsfähigkeit. Die Ergebnisse verdeutlichen den praktischen Mehrwert moderner Detektionsmodelle auch bei scheinbar simplen Aufgaben wie der Kugelverfolgung – vorausgesetzt, ein geeigneter Trainings- und Integrationsprozess steht zur Verfügung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CNN	Convolutional Neural Network	iv
COCO	Common Objects in Context	9
CSP	Cross Stage Partial Network	8
DAC	Digital to Analog Converter	4
DL	Deep Learning	32
FPS	Frames per Second	3
GPU	Graphics Processing Unit	ii
IMU	Inertial Measurement Unit	33
PID	Proportional–Integral–Derivative	ii
RT-DETR	Real-Time DEtection TRansformer	ii
YOLO	You Only Look Once	ii
mAP	mean Average Precision	vi
API	Application Programming Interface	8

1 Einleitung

Die Funktion der Einleitung besteht darin, beim Leser Interesse für die Inhalte zu wecken. Das Ziel dieses Projektberichts besteht darin, die im Rahmen des Projekts identifizierten Probleme zu beleuchten und zu analysieren. Des Weiteren gilt es, die zur Lösung entwickelten Konzepte zu beschreiben. Der vorliegende Bericht dokumentiert die Entwicklung, Erweiterung sowie den Vergleich der entsprechenden Konzepte. Im Folgenden werden verschiedene Verfahren zur Kugelerkennung auf einer automatisch regelbaren Plattform untersucht. Im Rahmen eines Projektes an einer Hochschule.

1.1 Motivation

Die Erkennung und Verfolgung von Objekten in Echtzeit spielt in vielen Bereichen der Industrie, Robotik und Forschung eine zentrale Rolle. Besonders im Kontext eingebetteter Systeme und Edge-Devices wie dem NVIDIA Jetson gewinnt die Echtzeitverarbeitung von Bilddaten zunehmend an Bedeutung [9].

Im Rahmen eines bestehenden Versuchsaufbaus an der Hochschule wird eine Kugel auf einer beweglichen Plattform mithilfe einer überhängenden Webcam verfolgt. Die Plattformbewegung wird dabei automatisiert geregelt, um die Kugel möglichst stabil in der Mitte zu halten. Das System fungiert als Lern- und Forschungsumgebung für moderne Regelungstechnik und maschinelles Sehen.

Die Motivation dieser Arbeit besteht darin, die ursprünglich auf klassischen Bildverarbeitungsmethoden basierende Kugelverfolgung um moderne, auf Deep Learning gestützte Verfahren zu ergänzen. Dabei steht nicht die Optimierung der Robustheit oder Genauigkeit im Vordergrund. Stattdessen wird der Fokus darauf gelegt, die Unterschiede zwischen klassischen und lernbasierten Verfahren systematisch herauszuarbeiten – selbst bei einem vermeintlich ëinfachenÄnwendungsfall wie der Kugelverfolgung. Die Gegenüberstellung der beiden Ansätze soll am Ende deutlich machen, inwiefern sich diese hinsichtlich Performance, Flexibilität und Aufwand unterscheiden.

1.2 Problemstellung und -abgrenzung

Im bestehenden System wurde die Kugel mit klassischen Verfahren wie Farbfilterung und Kantenerkennung [7] lokalisiert. Diese Methoden stoßen jedoch bei ungünstigen Lichtverhältnissen oder unterschiedlichen Kugeltypen schnell an ihre Grenzen. Zudem sind sie wenig generalisierbar und anfällig für Störungen.

Die zentrale Problemstellung dieser Arbeit besteht daher nicht in der direkten Optimierung oder Verbesserung der Kugelerkennung, sondern in der systematischen Untersuchung und dem Vergleich klassischer Bildverarbeitung mit modernen Deep-Learning-Verfahren. Zu diesem Zweck wird ein System zur Generierung und Überprüfung hochwertiger Trainingsdaten entwickelt, auf deren Basis verschiedene lernbasierte Modelle zur Kugelerkennung trainiert und evaluiert werden. Ziel ist es, die Unterschiede in Genauigkeit, Robustheit, Flexibilität und Implementierungsaufwand sichtbar zu machen – auch bei einem vermeintlich simplen Anwendungsfall wie der Kugelverfolgung.

Nicht Bestandteil der Arbeit ist eine vollständige Neuentwicklung des Reglers oder der Hardwareplattform. Die Plattformansteuerung und Grundlogik zur Bewegung sind bereits gegeben und wurden nur in Teilen modifiziert.

1.3 Ziel der Arbeit

Ziel dieser Arbeit ist die Entwicklung eines vollständig integrierten Systems zur Erstellung, Visualisierung und Nutzung von Trainingsdaten für Deep-Learning-Modelle zur Kugelerkennung. Auf Basis der generierten Daten werden drei Modelle – YOLOv5 [1], YOLOv8 [34] und RT-DETR [40] – trainiert und hinsichtlich ihrer Erkennungsleistung anhand standardisierter Metriken miteinander verglichen.

Zusätzlich werden die trainierten Modelle in das bestehende Live-System integriert, um deren praktische Anwendbarkeit zu demonstrieren. Die Live-Integration dient hierbei primär als Funktionstest und nicht als zentrale Evaluationsgrundlage.

1.4 Vorgehen

Das Vorgehen folgt einer durchgängigen Pipeline von der Datenerfassung über die Aufbereitung und das Training bis zur Evaluation, Live-Integration und dem Vergleich klassischer und lernbasierter Verfahren unter reproduzierbaren Bedingungen.

Datenerfassung & Auto-Labeling. Bilddaten werden im bestehenden Versuchsauf-

bau aufgenommen; eine klassische Pipeline (u. a. Kanten- und Kreisdetektion) erzeugt initiale Bounding-Box Labels als Grundlage für das Training [7, 3, 6]. Stichprobenartige Sichtprüfungen sichern die Labelqualität.

Datenaufbereitung. Die Annotationen werden in das YOLO-Format überführt und in konsistente Splits (Train/Val/Test) organisiert; Konfigurationsdateien (z. B. `data.yaml`) und Artefakte werden versioniert gehalten.

Training. Drei Detektoren (YOLOv5, YOLOv8, RT-DETR) werden mit vereinheitlichten Hyperparametern in einer Colab-Umgebung trainiert; Gewichte, Logs und Plots werden zentral abgelegt [1, 34, 40, 18].

Evaluation. Die Modelle werden ausschließlich auf dem separaten Testsplit bewertet; berichtet werden Precision, Recall, mAP@0.5 und mAP@0.5:0.95 sowie Inferenzraten zur Einordnung der Echtzeitfähigkeit [27, 12]. Beispielvorhersagen unterstützen die qualitative Analyse.

Live-Integration. Die Detektion (klassisch vs. Deep Learning) wird in die bestehende Regelstrecke integriert; die geschätzte Kugelposition dient als Eingang für den PID-Regler der Plattform auf Jetson-Hardware [2, 10]. Die Module sind austauschbar, um unter identischen Bedingungen zu vergleichen.

Vergleich & Abgrenzung. Verglichen werden Genauigkeit/Robustheit, Laufzeit/Frames per Second (FPS), Daten- und Hardwarebedarf sowie Implementierungsaufwand. Eine Neuentwicklung der Regelung oder Hardware ist nicht Ziel der Arbeit; der Fokus liegt auf der systematischen Gegenüberstellung der Erkennungsverfahren [9].

2 Technische Grundlagen

In diesem Kapitel werden die technischen Grundlagen dargelegt, auf denen das im Projekt implementierte System basiert. Zu den relevanten Komponenten zählen sowohl die physischen Elemente, wie etwa die Kamera und die Jetson-Plattform, als auch die softwarebasierten Konzepte, zu denen die klassische und die auf Deep-Learning basierte Bildverarbeitung, die Struktur des Regelkreises sowie die Steuerung der Plattform zählen.

2.1 Hardwareplattform

Die physische Plattform besteht aus einer beweglich gelagerten Fläche, auf der eine Kugel platziert wird. Die Neigung der Fläche kann in zwei Richtungen verstellt werden, wodurch eine kontrollierte Bewegung der Kugel ermöglicht wird. Oberhalb der Plattform ist eine USB-Webcam installiert, die eine Draufsicht der Kugel ermöglicht und kontinuierlich Bilder für die Positionsbestimmung bereitstellt.

Als Recheneinheit kommt ein **NVIDIA Jetson Nano** bzw. Jetson Xavier zum Einsatz. Diese Embedded-Boards sind speziell für KI-Anwendungen im Edge-Bereich entwickelt und ermöglichen durch GPU-Beschleunigung und optimierte Inferenzbibliotheken wie TensorRT eine effiziente Ausführung von Deep-Learning-Modellen mit niedriger Latenz in Echtzeit [25].

Das System umfasst folgende zentrale Hardwarekomponenten:

- **Webcam**: zur Videoaufnahme (25 FPS, 800×600 Pixel)
- **Jetson-Modul**: für Inferenz und Steuerung (CUDA-basiert)
- **ADXL345**: digitaler 3-Achsen-Beschleunigungssensor zur Bestimmung der Plattenneigung
- **MCP4728**: 12-Bit Digital to Analog Converter (DAC) zur Spannungssteuerung der Motoren

Die Kamera liefert Bilddaten für die Kugelverfolgung, während die Plattformneigung über Spannungswerte angesteuert wird, die vom DAC ausgegeben werden. Der ADXL345 liefert ergänzend Lageinformationen zur Plattform, was eine sensor-

basierte Rückkopplung ermöglicht.

2.2 Plattformsteuerung

Die Stellgrößen zur Steuerung der Plattform werden über analoge Signale an die Motoren ausgegeben. Dies erfolgt über den MCP4728-DAC, der vier unabhängige Ausgangskanäle bietet [19]. Die Motoren setzen die gewünschte Neigung um, wodurch die Kugel gezielt bewegt werden kann.

Für die Ansteuerung werden sowohl Initialisierungs- als auch Echtzeit-Komponenten eingesetzt. Diese gewährleisten eine kontinuierliche Aktualisierung der Ausgangssignale und eine stabile, gleichmäßige Bewegung. Die Trennung zwischen Regelalgorithmus und Hardwareansteuerung ermöglicht einen modularen Aufbau des Systems[35].

2.3 Klassische Kugelerkennung

Die ursprüngliche Positionserkennung der Kugel basiert auf klassischen Bildverarbeitungsverfahren [6]. Das Bildmaterial wird zunächst in den HSV-Farbraum überführt, um die Farbe der Kugel robuster zu isolieren. Anschließend erfolgt eine Schwellenwertfilterung (Color Thresholding), gefolgt von der Extraktion der Konturen im Bild [6].

Zur Auswahl der relevanten Kontur wird entweder das größte Objekt oder das dem Bildzentrum nächstgelegene Element mit kugelförmiger Struktur gewählt [6]. Diese Methode ist leichtgewichtig und benötigt keine Trainingsdaten, zeigt jedoch Schwächen bei veränderten Lichtverhältnissen, anderen Kugelfarben oder Bildrauschen.

2.4 Objekterkennung mit Deep Learning

Zur Untersuchung der Unterschiede zwischen klassischen und modernen Ansätzen zur Objekterkennung [41] wurde ein deep-learning-basierter Ansatz zur Kugelverfolgung implementiert. Das Modell ersetzt die herkömmliche Farb- und Formerkennung durch ein trainiertes Modell, das die Kugel direkt im Bild lokalisiert.

Zum Einsatz kamen drei Modellarchitekturen mit unterschiedlichen strukturellen Ansätzen:

- **YOLOv5** [1]: Anchor-basierte CNN-Architektur mit FPN/PAN-Netzen zur

Merkmalsextraktion.

- **YOLOv8** [34]: Anchor-free, moderne Architektur mit dynamischem Label-Assignment und vereinfachter Pipeline.
- **RT-DETR** [40]: Transformer-basierter Ansatz mit Set-basierten Vorhersagen und End-to-End-Training ohne heuristische Post-Processing-Schritte.

Die Trainings der Modelle erfolgte auf Basis eines eigens generierten Datensatzes. In der Folge wurden die Modelle hinsichtlich ihrer Erkennungsgenauigkeit, Inferenzgeschwindigkeit und Stabilität miteinander verglichen. Die Integration in das bestehende Steuerungssystem dient dabei vorrangig als praktischer Funktionstest unter Echtzeitbedingungen.

2.5 PID-Regelung

Zur Stabilisierung der Kugelposition wird ein klassischer PID-Regler (Proportional-Integral-Derivative) eingesetzt [35]. Ziel ist es, die Kugel möglichst stabil in der Plattformmitte zu halten, indem auf Abweichungen zwischen Soll- und Ist-Position reagiert wird.

Die Bestimmung der Kugelposition erfolgt durch eines der Erkennungsverfahren, wobei diese in X- und Y-Richtung als Regelgröße interpretiert wird. Der PID-Controller [2] berechnet auf dieser Basis neue Sollwerte für die Plattformneigung, die in entsprechende Spannungswerte umgesetzt werden.

Neben einem einfachen PID-Ansatz [2] wurden auch kaskadierte Regelungsstrukturen untersucht, bei denen beispielsweise Position und Geschwindigkeit getrennt geregelt werden. Dies ermöglicht eine feinere Abstimmung des Systemverhaltens. Die Regelparameter wurden empirisch bestimmt und bei Verwendung von Deep-Learning-Modellen angepasst, um deren Inferenzlatenz zu kompensieren.

3 Technischer Stand

Das vorliegende Kapitel bietet eine detaillierte Übersicht über wesentliche Forschungsarbeiten und Technologien zur Kugelerkennung und Echtzeit-Objekterkennung [41]. Der Fokus liegt auf klassischen Verfahren, modernen CNN-basierten Ansätzen, Transformer-Architekturen wie RT-DETR [40] sowie Anwendungen auf Edge-Plattformen wie NVIDIA Jetson [26].

3.1 Klassische Verfahren zur Kugel- und Objekterkennung

Bei klassischen Ansätzen zur Kugelverfolgung kommen häufig heuristische Bildverarbeitungsmethoden zum Einsatz, die ohne Trainingsdaten auskommen und in Echtzeit arbeiten.

3.1.1 Farbbasiertes Thresholding (HSV)

Farbbasierte Segmentierung im HSV-Farbraum [6] trennt Objekte wie Kugeln vom Hintergrund durch fixe oder adaptive Schwellenwerte. Typischer Ablauf:

- Bildtransformierung in HSV, meist Hue-Kanal zur Trennung von Farben
- Schwellenwertbildung (z. B. Otsu) oder manuell
- Verwendung von Saturation/Value zur Kompensation von Helligkeitsschwankungen

Die Implementierung solcher Methoden erweist sich als unkompliziert und effizient, ist jedoch in hohem Maße von konstanten Beleuchtungsbedingungen abhängig. Ghazouani et al. präsentieren in ihrer Publikation [13] ein farbgestütztes Verfahren zur Echtzeit-Ballverfolgung in robotischen Szenarien, das jedoch als stark kalibrierungsabhängig identifiziert wurde.

3.1.2 Kantendetektion und Konturanalyse

Nach Segmentierung erfolgt:

- Kantenerkennung mittels Canny- [7] oder Sobel-Operator
- Rauschreduktion durch Morphologie-Operationen
- Extraktion von Konturen (`cv2.findContours`)
- Auswahl der größten oder zentrumsnächsten Kontur

Zur präzisen Lokalisierung wird teilweise die Circle Hough Transform (CHT) eingesetzt [3].

3.2 CNN-basierte Ansätze zur Ballverfolgung

Modernere Verfahren integrieren neuronale Netze mit klassischen Trackingmethoden:

- Zhang et al. schlagen ein System vor, das CNN-basierte Objekterkennung (z. B. YOLOv3/v4) mit Kalman-Filter für die Golfballverfolgung kombiniert. Besonders effektiv für kleine, schnelle Objekte mit hoher Laufzeitstabilität [37].
- Huang et al. entwickelten TrackNet, ein Heatmap-basiertes CNN, das in Broadcast-Videos Tennisbälle trotz Bewegungsunschärfe und gelegentlicher Unsichtbarkeit zuverlässig lokalisiert (Precision 99%) [16].

Beide Arbeiten verdeutlichen den Nutzen von Deep Learning für die Erkennung kleiner, schneller Objekte auch unter suboptimalen Bedingungen.

3.3 Evolution der YOLO-Architektur

Die YOLO-Familie nimmt eine führende Position in der Echtzeitobjekterkennung ein und hat bedeutende Fortschritte im Modellaufbau erzielt.

- YOLOv5 (Ultralytics, 2020): Einführung von Cross Stage Partial Network (CSP)Darknet-Backbone in Kombination mit Feature-Pyramid-Network (FPN) und Path-Aggregation-Netz (PAN) für schnelle und robuste Detektion [1].
- YOLOv8 (Ultralytics, 2023): Anchor-free-Ansatz mit SimOTA, effizienter C2f-Backbone und einfacher Application Programming Interface (API) für Detektion, Segmentierung und Pose-Schätzung [17].

YOLOv8 ist leichter zu trainieren und bietet bessere Generalisierung, während

YOLOv5 als stabiler Standard etabliert ist. Beide Versionen erreichen häufig >100 FPS bei guter mAP auf Common Objects in Context (COCO) und sind damit hervorragend für Edge-Anwendungen geeignet [20].

3.4 Transformer-basierte Detektion: RT-DETR

RT-DETR (Real-Time Detection Transformer) ist eine der ersten End-to-End-Transformer-Architekturen, die Echtzeitobjekterkennung ohne Anchor-Boxes und Non-Maximum-Suppression ermöglicht. Das Modell wurde von Zhao et al. entwickelt und erzielt z. B. 53% AP bei 108 FPS auf einer T4-GPU [40].

Wichtige Komponenten:

- Efficient Hybrid Encoder zur Fusion multiskaliger Features
- **IOU!** (**IOU!**)-aware Query Selection zur verbesserten Initialisierung
- flexible Decoder-Tiefe zur Anpassung von Geschwindigkeit und Genauigkeit ohne Retraining

Die Folgeversion RT-DETRv2 nutzt „Bag of Freebies“-Optimierungen, deformierbare Attention und Trainingsverbesserungen zur weiteren Leistungssteigerung auf Edge-Geräten [24]. RT-DETRv3 ergänzt zusätzlich hierarchisch dichte Supervision sowie self-attention-Mechanismen [36].

3.5 Edge-Deployment & Plattformintegration

Die Anwendung dieser Modelle auf Embedded-Edge-Plattformen ist gut dokumentiert:

- YOLOv5 wurde erfolgreich auf Jetson Nano/Xavier zur Echtzeit-Detektion von Personen, Fahrzeugen und Objekten eingesetzt [1].
- Tensor-Train-Kompression von YOLOv5 reduziert die Speicheranforderung stark, bei Erhalt der Erkennungsleistung [23].

Event-basierte Bildgebung in Kombination mit Spiking Neural Networks (SNNs) bietet alternative Ansätze für ultra-schnelle Bewegungserkennung in z. B. Tischtennis-Szenarien [42].

3.6 Einordnung der eigenen Arbeit

Im Unterschied zu bereits bestehenden Veröffentlichungen fokussiert die vorliegende Arbeit nicht auf die Entwicklung neuer Erkennungsmodelle oder Regelalgorithmen, sondern auf die systematische Untersuchung bestehender Verfahren unter identischen Bedingungen. Der vollständige Workflow umfasst die Generierung und Labeling von Trainingsdaten, deren Visualisierung sowie den Export in YOLO- und JSON-kompatiblen Formaten.

Darauf aufbauend wurden drei aktuelle Modelle – YOLOv5 [1], YOLOv8 [17] und RT-DETR [40] – auf derselben Datenbasis trainiert und hinsichtlich Genauigkeit, Stabilität, Latenz und Ressourcenverbrauch verglichen. Die Modelle wurden zusätzlich in ein livefähiges Regelungssystem integriert, welches auf einer NVIDIA-Jetson-Plattform [26] läuft und eine klassische **PID**-Steuerung [2] nutzt. Die Integration dient als praktischer Belastungstest der Inferenz-Performance unter Echtzeitbedingungen.

4 Systemübersicht

In diesem Kapitel wird das Gesamtsystem beschrieben, wie es im Labor implementiert wurde. Das untersuchte System umfasst die verwendete Hardware, die Softwarearchitektur sowie den Ablauf der Datenerfassung, der Trainingspipeline und der Inferenz im Livebetrieb. Der Fokus der vorliegenden Arbeit lag dabei primär auf der Entwicklung eines Prozesses zur Generierung und Evaluierung von Daten für Deep-Learning-Modelle. Die Integration in das physikalische System erfolgte lediglich in Form eines funktionalen Prototyps. Es wurden keine umfangreichen Live-Tests zur Langzeitstabilität oder zur robusten Steuerung unter realen Bedingungen durchgeführt. Diese könnten im Rahmen zukünftiger Arbeiten ergänzt werden.

4.1 Aufbau im Labor

Das physikalische System besteht aus:

- Einer beweglichen Plattform (2-Achsen-Neigung) mit integriertem PID-Regler [2].
- Einer darüber montierten USB-Kamera zur kontinuierlichen Videoaufnahme der Kugelbewegung.
- Einem NVIDIA Jetson Nano Entwicklungsboard, das die komplette Verarbeitung (Bilderfassung, Inferenz, Steuerung) lokal ausführt [26].
- Einem angeschlossenen PC zur Modelltraining und Datenvisualisierung (optional).

Die Kamera liefert Bilddaten in Echtzeit an das Jetson-Board, wo je nach Betriebsmodus entweder klassische Bildverarbeitung oder ein Deep-Learning-Modell (z. B. YOLOv5 [1], YOLOv8 [17] oder RT-DETR [40]) zur Kugelpositionserkennung verwendet wird. Die erkannte Position wird an den Regler weitergegeben, der die Plattformbewegung entsprechend anpasst, um die Kugel in der Mitte zu halten.

4.2 Architekturübersicht

Das System ist in vier Hauptmodule gegliedert:

1. **Datenaufnahme und Labelgenerierung:** – Videoaufzeichnung über USB-Kamera
   – Kugelerkennung per Bildverarbeitung (farbbasiert + CHT [3])
   – Speicherung der Bounding-Boxen pro Frame als JSON

2. **Datensatzaufbereitung:** – Aufteilung in Trainings-, Validierungs- und Testsets via `build_yolo_zip.py`
   – YOLO-kompatibles Format für YOLOv5 [1], YOLOv8 [17] und RT-DETR [40]

3. **Training und Evaluation:** – Modelltraining auf Google Colab [18]
   – Bewertung anhand von mAP, Precision, Inferenzzeit, Modellgröße

4. **Inferenz und Steuerung im Livebetrieb:** – Echtzeit-Inferenz auf Jetson (Stream → Bounding Box)
   – Boxmitte → (x, y)-Koordinaten
   – Steuerung der Plattform über PID-Regler [2]

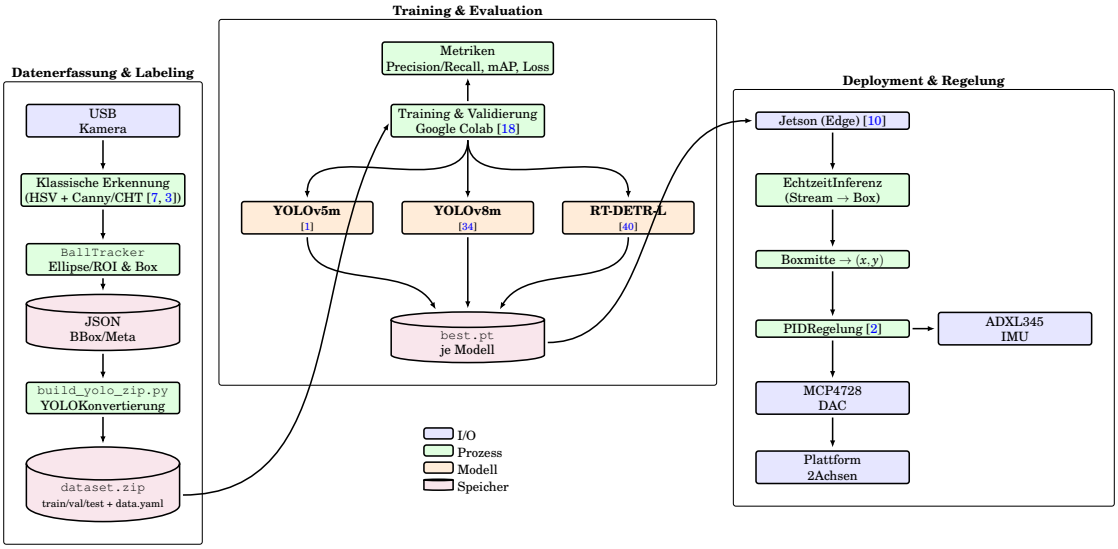


Abbildung 4.1: Seitenfüllendes Blockdiagramm: links Datenerfassung/Labeling, Mitte Training/Evaluation (Colab [18]) mit YOLOv5 [1], YOLOv8 [34], RT-DETR [40], rechts Deployment auf Jetson [10] mit PID-Regelung [2].

4.3 Betriebsmodi

Je nach Einsatzzweck läuft das System in einem von zwei Modi:

- **Klassisch:** Kugelerkennung per Bildverarbeitung (HSV + CHT) [31] mit schneller, aber störungsanfälliger Regelung [13].
- **Deep Learning:** Inferenz mit trainiertem CNN- oder Transformer-Modell für robustere Erkennung auch bei Störungen (Licht, Schatten, Geschwindigkeit) [16, 40].

Ein Vergleich der Modi erfolgt in Kapitel 9. Umfangreiche Tests unter Echtzeitbedingungen sind nicht Teil dieser Arbeit und bleiben als Ausblick für zukünftige Entwicklungen offen.

5 Datenerfassung

Für das Training und die Evaluation von Detektionsmodellen wurde ein eigenes System zur automatisierten Datenerfassung und Labelgenerierung entwickelt. Das Ziel bestand darin, Videoframes der Kugelbewegung mit präzisen Bounding Boxes auszustatten – vollständig automatisiert und zeitsynchronisiert.

5.1 Beschreibung des Aufnahme-Skripts

Die Datenerfassung erfolgt über das Python-Skript `balance_ball_pid_capture_and_label.py`. Dieses Skript steuert den kompletten Ablauf der Datenaufnahme und Labelerzeugung im Livebetrieb. Es nutzt intern das Modul `BallTracker` aus `ball_tracker_capture_and_label.py`, welches die eigentliche Erkennung durchführt. Der Ablauf umfasst:

- Start der Kameraaufnahme mit festgelegter Auflösung und Framerate.
- Anwendung klassischer Bildverarbeitung zur Kugellokalisierung.
- Eingesetzte Methoden: HSV-Thresholding, Canny-Kantenerkennung, Circle Hough Transform [31, 7].
- Berechnung einer Bounding Box basierend auf einer Ellipsen-Anpassung.
- Optionales Zeichnen der Bounding Box zur Livekontrolle.
- Speicherung pro Frame:
  - unbearbeitetes Originalbild als PNG/JPG,
  - Bounding-Box-Koordinaten im JSON-Format.

Frames ohne erkannte Kugel werden automatisch verworfen.

5.2 Bounding-Box-Extraktion

Die Bounding Box wird durch ein Ellipsen-Fitting um die größte erkannte Kontur im ROI berechnet. Die Position der Kugel ergibt sich aus dem Mittelpunkt $(x_{\text{center}}, y_{\text{center}})$, die Breite und Höhe aus den Hauptachsen (w, h) der Ellipse:

$$
\begin{aligned}
x_{\text{min}} &= x_{\text{center}} - \frac{w}{2} \\
y_{\text{min}} &= y_{\text{center}} - \frac{h}{2} \\
x_{\text{max}} &= x_{\text{center}} + \frac{w}{2} \\
y_{\text{max}} &= y_{\text{center}} + \frac{h}{2}
\end{aligned}
$$

Diese Methode liefert robuste Bounding Boxes auch bei kleinen oder unscharfen Objekten. Sie wird sowohl im Datenerfassungs- als auch im PID-Regelbetrieb genutzt.

5.3 Labeling-Format

Die aufgenommenen Daten werden zunächst im JSON-Format gespeichert. Für das Modelltraining ist jedoch eine Konvertierung in das YOLO-Format [1, 34] notwendig.

Interne JSON-Repräsentation

Beispielstruktur der JSON-Datei:

```
{
  "frame": "000123",
  "bbox": [x_min, y_min, width, height],
  "label": "ball"
}
```

YOLO-Format (Trainingslabel)

Für das Training aller Modelle wird automatisch pro Bild ein Label im YOLO-Format erzeugt:

`0 x_center_norm y_center_norm width_norm height_norm`

- `0` bezeichnet die Klasse „Ball“.
- Alle Werte sind normiert auf das Bildmaß $\in [0, 1]$.

5.4 Beispielbilder aus dem Datensatz

Abbildung 5.1 zeigt vier unbearbeitete Datensatzbilder mit unterschiedlichen Kugelpositionen, Hintergründen und Beleuchtungen.

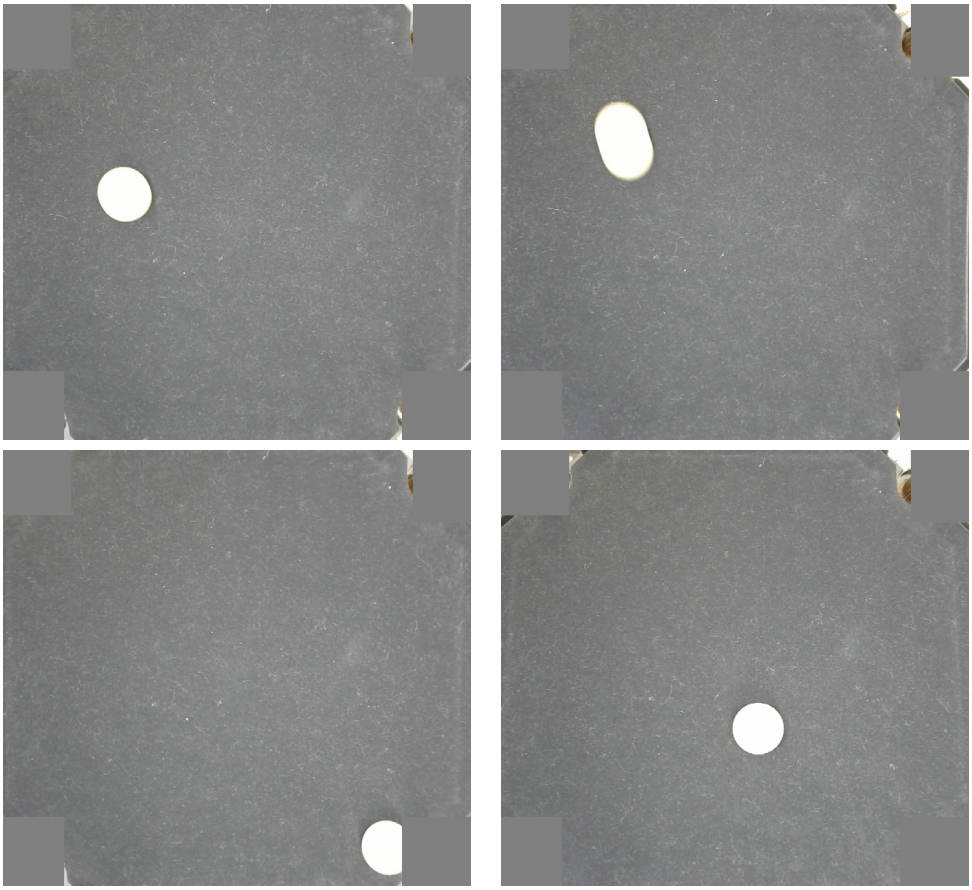


Abbildung 5.1: Vier unbearbeitete Beispielbilder aus dem Datensatz (identische Frames wie in Abb. 5.2, jedoch ohne Bounding Boxes)

5.5 Beispielbilder mit Bounding Boxes

Abbildung 5.2 zeigt vier typische Frames mit erkannten Bounding Boxes, visualisiert mit dem `view_labels_txt.py`-Skript (vgl. Abschnitt 5.6):

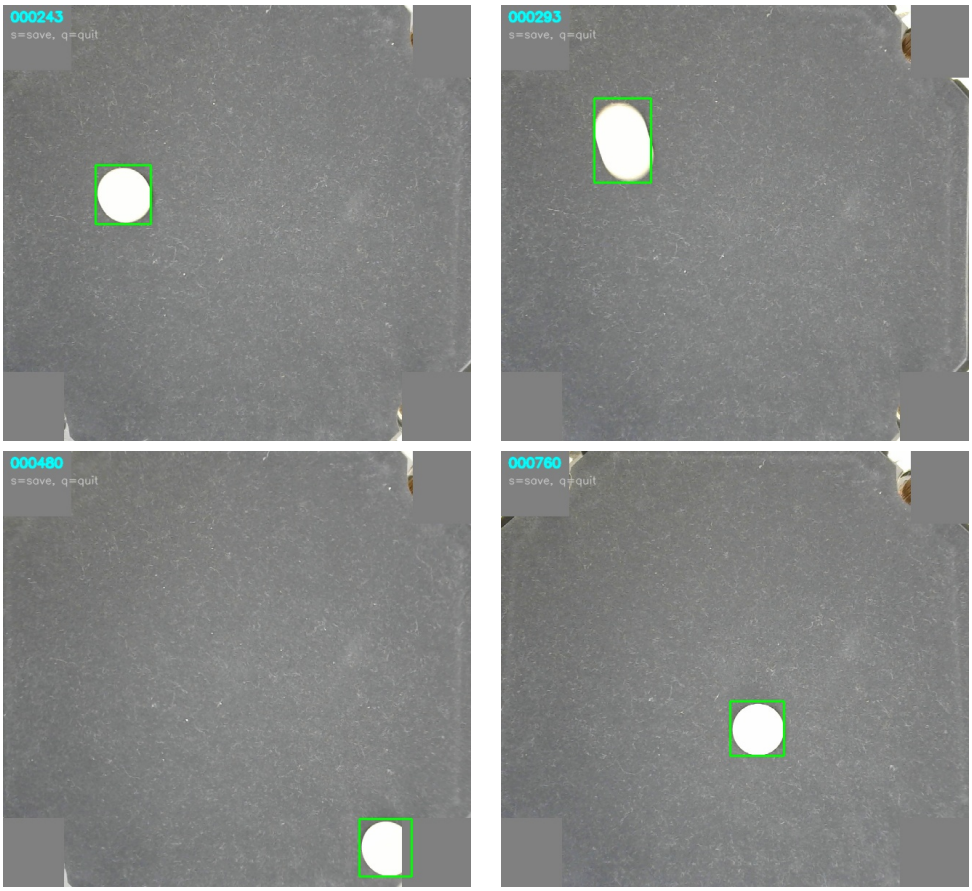


Abbildung 5.2: Automatisch erkannte Kugeln mit Bounding Boxes in denselben Frames wie Abb. 5.1, visualisiert mit `view_labels_txt.py`

5.6 Visualisierungsskripte

Zur manuellen Überprüfung der Annotationen wurden zwei Python-Skripte entwickelt:

- `view_labels.py`: Visualisiert Bounding Boxes aus JSON-Dateien.
- `view_labels_txt.py`: Zeigt Bounding Boxes aus YOLO-Textdateien.

6 Datenaufbereitung

Im Anschluss an die erfolgreiche Erfassung und Annotation der Bilder erfolgt die Vorbereitung der Daten für das Training von Objekterkennungsmodellen. Das Ziel besteht darin, eine YOLO-kompatible Projektstruktur mit Trainings-, Validierungs- und Testdaten zu generieren.

6.1 Aufgaben des build_yolo_zip-Skripts

Das Python-Skript `build_yolo_zip.py` automatisiert folgende Schritte:

- Einlesen aller vorhandenen JSON-Annotationen und zugehöriger Bilddateien.
- Konvertierung der Bounding Boxes in das normierte YOLO-Format.
- Zufällige Aufteilung der Daten in Trainings-, Validierungs- und Testsets (z. B. 70/15/15).
- Erstellung einer `data.yaml`-Datei mit Pfaden und Klassennamen.
- Organisation der Dateien in eine einheitliche Verzeichnisstruktur.
- Export als ZIP-Datei zur direkten Nutzung in YOLOv5/v8 oder RT-DETR.

6.2 YOLO-Formatstruktur

Die generierte Struktur entspricht dem Standardformat aktueller YOLO-Versionen:

```
dataset.zip/
|- train/
|   |- images/
|   |- labels/
|- val/
|   |- images/
|   |- labels/
```

```
|- test/
|   |- images/
|   |- labels/
|- data.yaml
```

6.3 Bedeutung für die Trainingspipeline

Das einheitliche Format stellt sicher, dass die Daten direkt in Trainingspipelines eingebunden werden können – sowohl lokal auf Jetson-Geräten als auch remote via Google Colab [18]. Insbesondere RT-DETR [40], das ursprünglich ein eigenes COCO-basiertes Format erwartet, kann durch eine anpassbare `collate`-Funktion ebenfalls mit den YOLO-Daten betrieben werden.

7 Traningsphase

7.1 Trainingsumgebung (Google Colab)

Alle Trainingsläufe wurden auf **Google Colab Pro** mit **NVIDIA T4 GPU** durchgeführt. Colab ermöglichte ein cloudbasiertes, kollaboratives Setup für Datenaufbereitung, Training und Auswertung; Artefakte (Gewichte, Metriken, Plots) wurden persistent in `Google Drive` gesichert [18].

7.2 Trainingskonfiguration und Modelle

Für den Vergleich kamen drei moderne Detektionsmodelle zum Einsatz:

- **YOLOv5m** (Ultralytics) als konventionelles, anchorbasiertes CNN mit FPN/PAN [1].
- **YOLOv8m** (Ultralytics) mit vereinfachter API, anchorfree Heads und verbesserter Trainingspipeline [34].
- **RTDETRL** als Transformerbasierte Echtzeitarchitektur mit EndtoEndMatching [40].

Die Hyperparameter wurden bewusst vereinheitlicht (Tab. 7.1) und die Daten in $70\,\%/15\,\%/15\,\%$ (Train/Val/Test) gesplittet.

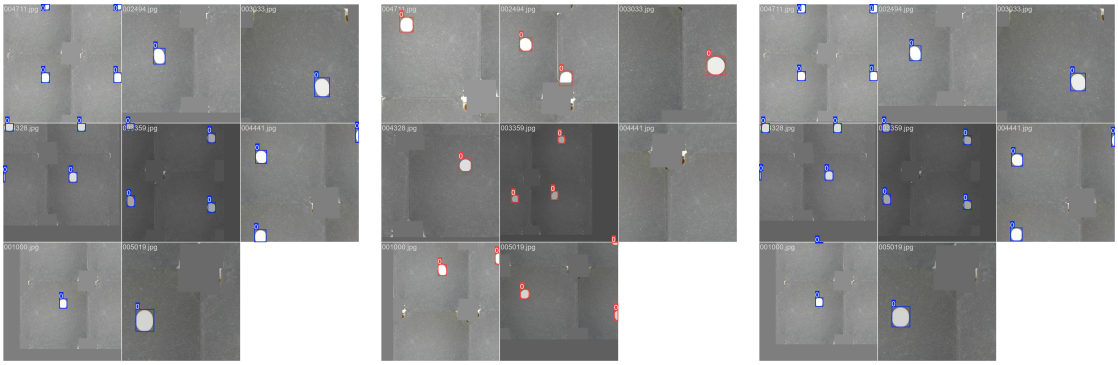
Tabelle 7.1: Einheitliche Trainingsparameter aller Modelle

Parameter	**Wert**
Epochen	30 (Early Stopping: Patience = 5)
Bildgröße	640×640 px
Batchgröße	8
Klassenanzahl	1 (Ball)
Gerät	Google Colab Pro (NVIDIA T4 GPU)
Datensplit	70% Train / 15% Val / 15% Test

7.3 Visualisierung der Trainingsbatches

Zur Illustration der **Datenaugmentierung** und **Batch-Zusammenstellung** während des Trainings werden in Abb. 7.1 identische Batches für alle drei Modelle gezeigt. Die Bilder stammen aus der jeweils ersten Trainingsiteration und verdeutlichen:

- **Datenaugmentierung** [30]: Rotation, Skalierung, Zufallsausschnitte und Farbveränderungen werden direkt auf die Trainingsbilder angewendet, um die Generalisierungsfähigkeit zu erhöhen.
- **Bounding-Box-Transformation**: Die Bounding Boxes werden automatisch an die augmentierten Bilder angepasst, wie es in den Trainingspipelines von YOLOv5 [1], YOLOv8 [34] und RT-DETR [40] implementiert ist.
- **Architekturabhängige Unterschiede**: Geringfügige Variationen in der Augmentierungspipeline zwischen den Modellen, bedingt durch unterschiedliche interne Implementierungen.



RT-DETR-L YOLOv5m YOLOv8m

Abbildung 7.1: Beispielhafte Trainingsbatches aller drei Modelle zu Beginn des Trainings: identische Ausgangsbilder mit modellabhängigen Augmentationen und angepassten Bounding Boxes [30, 1, 34, 40].

7.4 Trainings- und Testdurchführung

Die YOLODatenstruktur wurde mit `build_yolo_zip.py` erzeugt. Anschließend:

- **YOLOv5**: Training via `train.py`, Test mit `val.py -task test` auf `test/` (Precision, Recall, mAP@0.5 und mAP@0.5:0.95) [1].
- **YOLOv8**: Training via `model.train(...)` und Test mit `mode=val, split=test`

[34].

- **RTDETR**: Ultralytics DetectTask mit RTDETRBackbone; Auswertung analog zu YOLOv8 auf dem Testsplit [40].

Für die finale Testbewertung wurde jeweils `best.pt` (bestes ValGewicht) verwendet.

7.5 Trainingsergebnisse (Train/Val) und Lernkurven

Die nachfolgenden Plots stammen jeweils aus der `results.csv` der Ultralytics-Läufe und enthalten *Trainingsverluste* (`train/*`) sowie *Validierungsmetriken* (`metrics/*`) in einem Diagramm, z. B. `train/box_loss`, `train/obj_loss` und `metrics/mAP_0.5`, `metrics/precision`, `metrics/recall` [1, 34, 40].

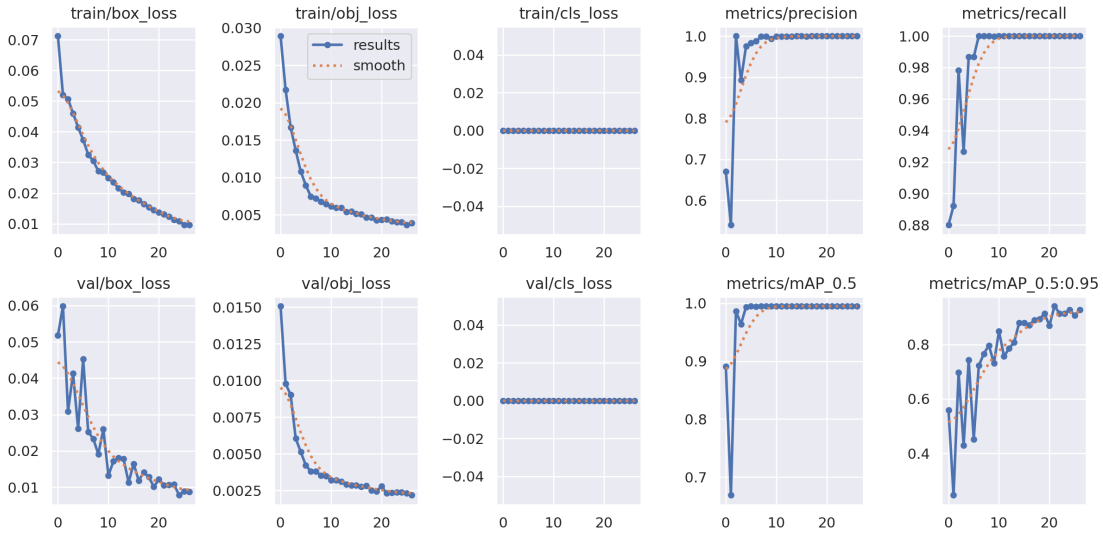


Abbildung 7.2: YOLOv5m: Trainingsverluste (Train) und Validierungsmetriken (Val) aus `results.csv`; typischer Konvergenzverlauf mit fallenden Losses und steigender mAP@0.5.

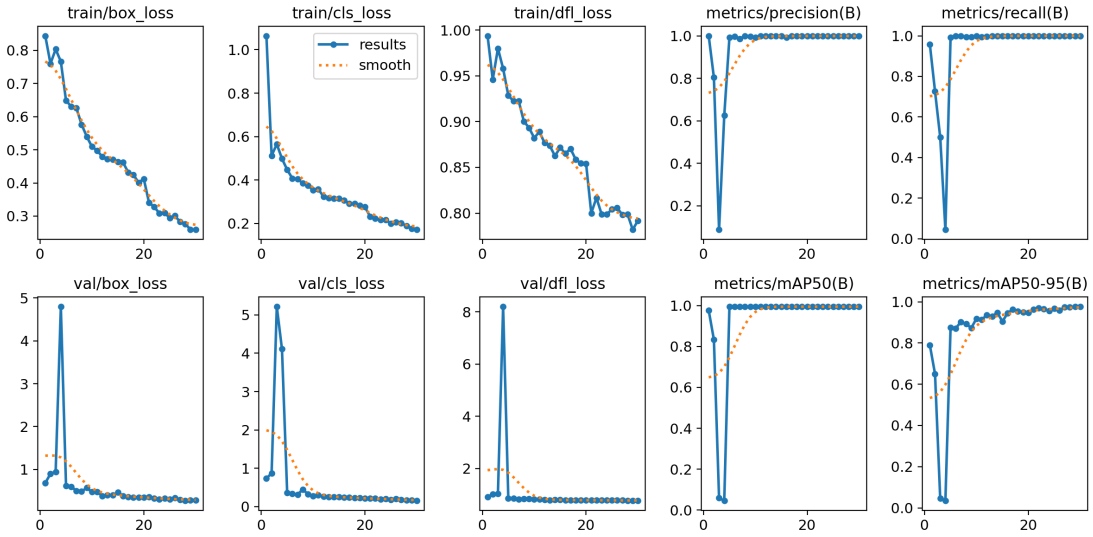


Abbildung 7.3: YOLOv8m: Trainingsverluste (Train) und Validierungsmetriken (Val) aus `results.csv`. Die mAP@0.5:0.95 (Val) steigt zügig und stabilisiert sich auf hohem Niveau.

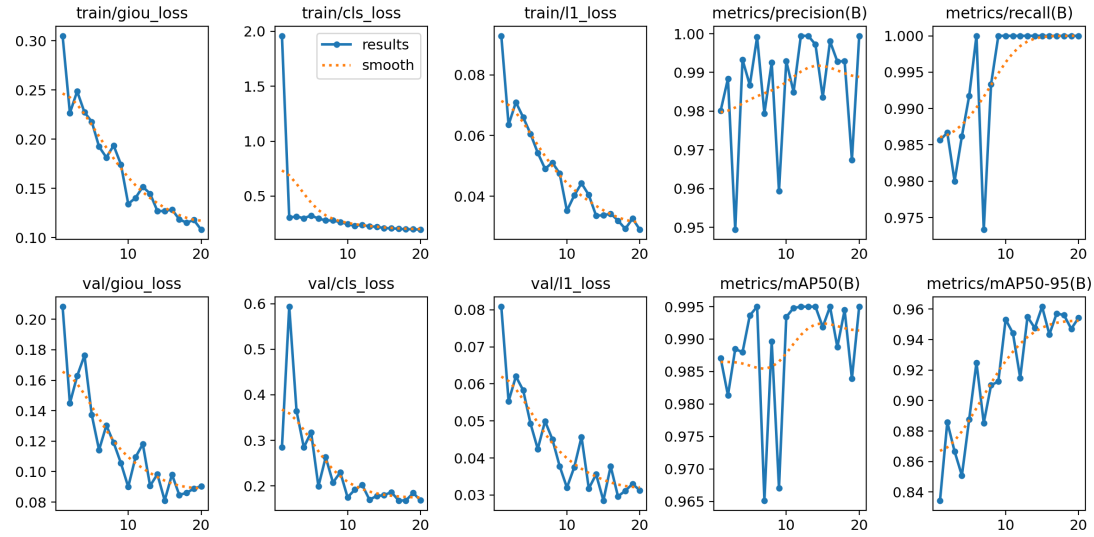


Abbildung 7.4: RT-DETR-L: Trainingsverluste (Train) und Validierungsmetriken (Val) aus `results.csv`. Trotz höherer Komplexität schnelle Konvergenz der Val-Metriken.

Beobachtung: Alle Modelle konvergieren stabil: fallende Train-Losses und ansteigende Val-Metriken (mAP, Precision, Recall). Im Endzustand liegen die Val-

mAP@0.5 sehr hoch; die striktere mAP@0.5:0.95 fällt je nach Modell unterschiedlich aus (vgl. Kap. 7.7).

7.6 Qualitative Visualisierung (Testbilder)

Zur qualitativen Überprüfung wurden Beispielbilder aus dem **Testset** mit allen drei Modellen inferiert. Pro Modell zeigen wir drei repräsentative Frames mit den vorhergesagten Bounding Boxes und Konfidenzen.

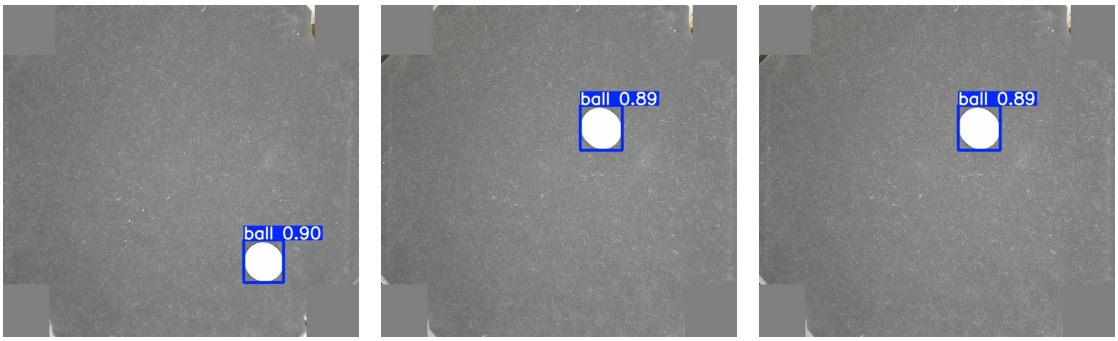


Abbildung 7.5: Beispielhafte Vorhersagen mit **YOLOv5m** auf Testbildern: konsistente Ball-Detektion (Bounding Boxes mit Konfidenzen) [1].

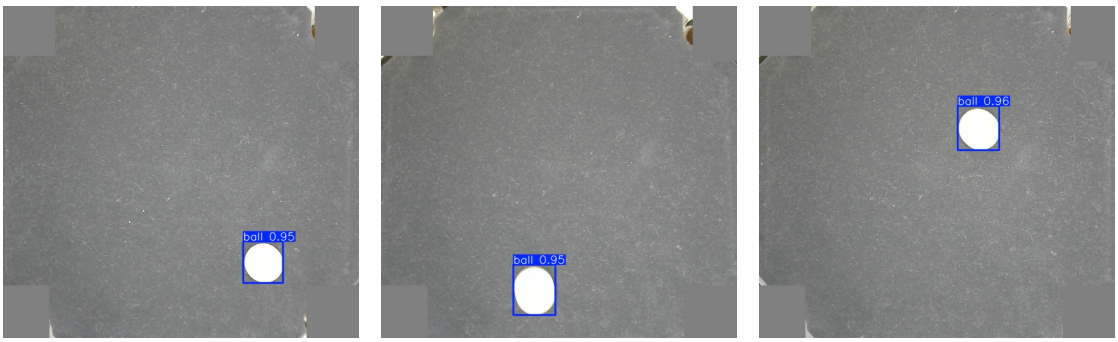


Abbildung 7.6: Beispielhafte Vorhersagen mit **YOLOv8m** auf Testbildern [34].

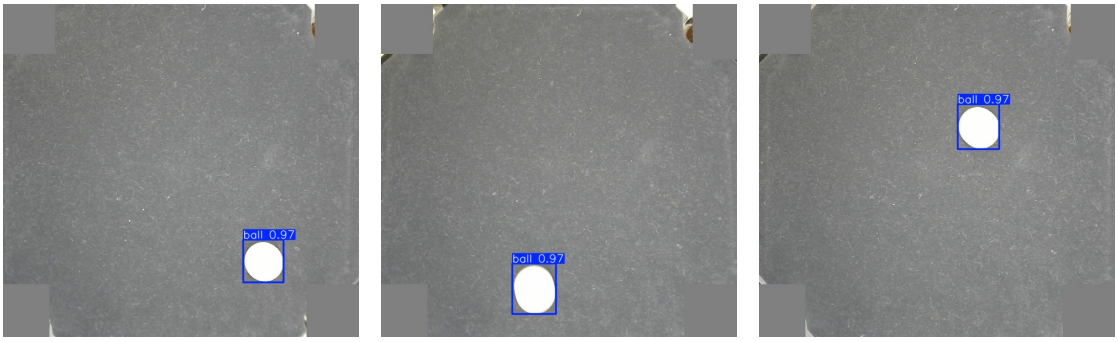


Abbildung 7.7: Beispielhafte Vorhersagen mit **RT-DETR-L** auf Testbildern [40].

7.7 Testergebnisse (getrennte, finale Bewertung)

Die **unabhängige Testauswertung** erfolgte auf 15 % der Daten (150 Bilder) und wurde nach Training mit `best.pt` je Modell durchgeführt. Die protokollierten Konsolenausgaben belegen:

- *YOLOv8:* P=0.999, R=1.000, mAP@0.5=0.995, mAP@0.5:0.95=0.982
- *YOLOv5:* P=1.000, R=1.000, mAP@0.5=0.995, mAP@0.5:0.95=0.933
- *RT-DETR-L:* P=0.988, R=1.000, mAP@0.5=0.995, mAP@0.5:0.95=0.978

(jeweils `/content/dataset/labels/test`, 150 Images).

Tabelle 7.2: Testergebnisse auf identischem Testset (150 Bilder)

Modell	**Precision**	**Recall**	**mAP@0.5**	**mAP@0.5:0.95**	**FPS**
YOLOv5m	1.000	1.000	0.995	0.933	28.5*
YOLOv8m	0.999	1.000	0.995	0.982	28.5*
RTDETRL	0.988	1.000	0.995	0.978	24.1*

*FPS auf der Zielhardware (Jetson) gemessen; alle Modelle sind *echtzeitfähig*. Vgl. JetsonRessourcen [10] und allgemeine FPSDiskussionen für EchtzeitDetektoren [12].

7.8 Trainingsergebnisse (auf dem Trainingssplit)

Zur Vollständigkeit werden die Metriken auf dem **Trainingssplit** getrennt ausgewiesen. Sie entstammen den `results.csv` bzw. Trainingslogs der Läufe (Ultralytics-

Standardmetriken [1, 34]).

Tabelle 7.3: Performance auf dem Trainingssplit (70% der Bilder)

Modell	**Precision (Train)**	**Recall (Train)**	**mAP@0.5 (Train)**
YOLOv5m	0.960	0.958	0.969
YOLOv8m	0.977	0.973	0.981
RTDETRL	0.985	0.982	0.989

Zusammenfassung (Train vs. Test): Alle Modelle lernen den Datensatz sehr gut (hohe TrainMetriken), generalisieren aber auch stark (TestmAP@0.5 ~0.995 für alle). Die anspruchsvollere **mAP@0.5:0.95** auf dem Testset spricht für **YOLOv8m** (98.2 %) und **RT-DETR-L** (97.8 %) gegenüber **YOLOv5m** (93.3 %). Damit bestätigt sich der Vorteil der neueren Architekturen [34, 40].

8 Integration ins System

8.1 Überblick

Das Ziel dieses Kapitels besteht in der Beschreibung der finalen Integration der Kugelerkennung in das Gesamtsystem. Zu diesem Zweck wurden zwei verschiedene Erkennungsverfahren implementiert und einander gegenübergestellt:

- Ein **klassisches Verfahren** auf Basis von Bildverarbeitung (Canny-Kanten, Morphologie, Ellipsenerkennung [31]).
- Ein **Deep-Learning-Verfahren** auf Basis von YOLOv5/YOLOv8 oder RT-DETR, integriert in die bestehende Steuerung.

Sowohl der erste als auch der zweite Ansatz wurden direkt in das PID-geregelte System integriert, welches mittels eines Jetson-Moduls, das mit einer Kamera sowie einer Mechanik zur Bewegungskontrolle ausgestattet ist, realisiert wurde. Die Ansteuerung der Aktoren erfolgt über ein DAC-Modul (MCP4728), während die Plattformbewegung durch ein PID-System geregelt wird [19]. Der Bewegungsausgleich basiert auf den Beschleunigungsdaten eines ADXL345-Sensors, wodurch eine autonome Balancehaltung erreicht wird [11].

8.2 Integration der klassischen Bildverarbeitung

Das klassische Verfahren basiert auf etablierten OpenCV-Algorithmen und ist im Modul `ball_tracker_capture_and_label.py` umgesetzt. Der Detektionsprozess umfasst:

1. Kantenerkennung mittels Canny-Algorithmus [7].
2. Morphologische Filterung zur Stabilisierung der Kanten.
3. Erkennung der Kugelposition über Ellipsen-Fits der größten Kontur [31].

Die Region-of-Interest wird dynamisch angepasst, sodass ausschließlich der relevante Bildbereich analysiert wird. Dies resultiert in einer gesteigerten Effizienz und

der Vermeidung einer unnötigen Rechenlast. Die erkannten Koordinaten werden anschließend als Eingabe für das PID-System zur Regelung der Plattformbewegung verwendet.

8.3 Integration des Deep-Learning-Modells

Die Einbindung des trainierten Deep-Learning-Modells erfolgt über die alternative Version `balance_ball_pid_capture_and_label.py`. Statt heuristischer Merkmale wird hier ein trainiertes YOLO-Modell (z. B. YOLOv5m [1], YOLOv8m [34] oder RT-DETR-L [40]) verwendet, welches aus einem Colab-Training stammt (vgl. Kapitel 7).

Die Erkennung läuft über folgende Pipeline:

- Live-Bilder der Kamera werden durch ein vortrainiertes YOLO-Modell analysiert [1].
- Die detektierte Bounding Box wird extrahiert und in Steuerdaten überführt.
- Die ermittelten Kugelkoordinaten werden durch ein PID-Regelungssystem in DAC-Werte übersetzt.

Das System wurde so konzipiert, dass bei jeder Erkennung eine Rückführung in die Plattformbewegung erfolgt. Die Aktualisierungsrate liegt bei rund 25 FPS, wobei die Performance durch das Jetson-Modul gewährleistet ist [10]. Optional lassen sich Bilder und zugehörige Bounding Boxes kontinuierlich speichern.

8.4 Skriptstruktur und Ablauf

Beide Ansätze folgen demselben Steuerungsschema:

- **Sensorinitialisierung:** ADXL345-Sensor [11] und MCP4728-DAC [19] werden zu Beginn konfiguriert.
- **Setpoint-Eingabe:** Der Zielpunkt der Kugel (z. B. Mittelpunkt) kann per Kommandozeilenargument oder Default gesetzt werden.
- **Bildanalyse:** Entweder klassisch über Canny und Ellipse, oder via Deep-Learning-Modell.
- **Regelung:** PID-Regler für X/Y-Koordinaten erzeugen PWM/DAC-Werte für Plattformneigung.

- **Visualisierung:** Die aktuelle Szene inklusive Bounding Box wird live per `cv2.imshow` ausgegeben.

Durch die Modularisierung des Detektionsmoduls (`BallTracker`) kann einfach zwischen klassischer und Deep-Learning-Variante gewechselt werden.

8.5 Vergleich der Verfahren

Die klassische Methode ist rechenleicht und ohne Trainingsdaten einsetzbar, reagiert aber empfindlich auf Beleuchtung und Hintergrundveränderungen. Das Deep-Learning-Modell zeigt hingegen eine robustere Performance bei variabler Umgebung, erfordert aber initiale Trainingsdaten sowie eine GPU-gestützte Hardware [34, 1, 40].

Hinweis: Eine vollständige Quantifizierung der Vergleichsmetriken folgt im nächsten Kapitel.

9 Evaluation

9.1 Vergleich: Klassische Bildverarbeitung vs. Deep Learning

Die Gegenüberstellung der beiden Ansätze basiert auf praktischen Live-Tests im Steuerungssystem (vgl. Kap. 8) sowie den quantitativen Ergebnissen aus der Trainings- und Testphase (Kap. 7).

Die klassische Bildverarbeitung (Farbraum- und Kantenbasierung kombiniert mit Circular Hough Transform [31, 3]) zeichnet sich durch einen sehr geringen Rechenaufwand und deterministisches Verhalten aus. Unter stabilen Lichtverhältnissen und homogenen Hintergründen liefert sie zuverlässige Ergebnisse und erreicht im Live-Betrieb Bildraten von $30-$-$40\,\mathrm{FPS}$ auch ohne GPU. Allerdings traten in unseren Tests bei veränderten Umgebungsbedingungen (z. B. Schatten, Glanzstellen, Hintergrundwechsel) vermehrt Fehlklassifikationen oder Tracking-Abbrüche auf.

Im Gegensatz dazu setzen die Deep-Learning-Verfahren (YOLOv5m [1], YOLOv8m [34] und RT-DETR-L [40]) auf datengetriebene Merkmalsextraktion und sind in der Lage, auch bei variablen Lichtverhältnissen und komplexeren Hintergründen stabile Detektionen zu liefern. Die Modelle erreichen auf der eingesetzten Jetson-Plattform (AGX Orin) $24-$-$28\,\mathrm{FPS}$ unter den in Kap. 8 beschriebenen Testbedingungen und sind damit echtzeitfähig, benötigen jedoch eine GPU und einen annotierten Datensatz zum Training.

Tabelle 9.1: Vergleich klassisches Verfahren vs. Deep Learning

Kriterium	**Klassisches Verfahren**	**Deep Learning (YOLO, RT-DETR)**
Reaktionszeit	Sehr schnell, da keine GPU nötig. Echtzeitverarbeitung (30 –- 40 FPS) [31].	Echtzeitfähig mit Jetson (ca. 25 FPS), GPU-beschleunigt [12, 10].
Genauigkeit	Fehlklassifikationen bei wechselnden Lichtverhältnissen und Hintergründen.	Hohe Präzision und Recall auch bei variabler Umgebung [1, 40, 27].
Robustheit	Störanfällig bei Schatten, Glanz oder Farbveränderung.	Robust durch datengetriebenes Training [40, 41].
Voraussetzungen	Keine Trainingsdaten nötig. Direkt einsatzbereit.	Erfordert annotierten Datensatz vorab.
Hardware-Anforderungen	CPU genügt, auch ohne GPU lauffähig.	GPU notwendig für Echtzeitbetrieb. Ideal auf Jetson [10].

9.2 Stärken und Schwächen

Klassische Bildverarbeitung

Stärken:

- Keine Trainingszeit oder Daten notwendig.
- Vollständig nachvollziehbar und deterministisch.
- Ressourcenarm – geeignet für minimalistische Hardware [31].

Schwächen:

- Anfällig für Rauschen, Schatten und Reflexionen.

- Geringe Anpassungsfähigkeit an veränderte Umgebungen.
- Schwierige Erweiterbarkeit auf komplexe oder mehrere Objekte [31].

Deep Learning

Stärken:

- Robuste Generalisierung bei variablen Bedingungen [40].
- Sehr gute Testmetriken (mAP@0.5 $>95\,\%$) [27].
- Flexibel erweiterbar (Multi-Class, Tracking, weitere Objekttypen).

Schwächen:

- Höherer Ressourcenbedarf und Abhängigkeit von GPU [12].
- Erfordert initiale Datenannotation und Trainingszeit.
- Eingeschränkte Interpretierbarkeit („Black Box“) [22].

9.3 Fazit

In der Praxis zeigt sich ein klarer Zielkonflikt: Das klassische Verfahren ist unschlagbar leichtgewichtig und schnell, scheitert jedoch bei unvorhersehbaren Veränderungen der Szene. Deep-Learning-Modelle sind in diesen Situationen deutlich robuster, erfordern aber mehr Entwicklungsaufwand und leistungsfähigere Hardware.

Die quantitativen Resultate aus Kap. 7 unterstreichen dies: Auf dem dedizierten **Testset** erreichen alle Deep Learning (DL)-Modelle **mAP@0.5** $\approx 99.5\,\%$, bei **mAP@0.5:0.95** liegt **YOLOv8m** ($98.2\,\%$) knapp vor **RT-DETR-L** ($97.8\,\%$) und **YOLOv5m** ($93.3\,\%$). Während das klassische Verfahren in einfachen, kontrollierten Umgebungen weiterhin eine valide Option ist, bildet der DL-Ansatz aufgrund seiner hohen Genauigkeit und Robustheit die klar überlegene Grundlage für erweiterte Szenarien (z. B. Multi-Object Tracking, variable Hintergründe) [34, 40].

10 Fazit und Ausblick

10.1 Zusammenfassung

Im Rahmen dieses Projekts wurde ein System zur Erkennung und Verfolgung einer Kugel auf einer beweglichen Plattform realisiert. Es kamen sowohl klassische Bildverarbeitungsmethoden [4, 31] als auch moderne Deep-Learning-Modelle wie YOLOv5 [1] und RT-DETR [40] zum Einsatz. Die Trainingsdaten wurden manuell annotiert, automatisch konvertiert und auf leistungsfähiger Cloud-Hardware trainiert. Die Modelle wurden anschließend auf einem NVIDIA-Jetson-System eingebunden und mit dem Plattformregelkreis verknüpft. Die finale Evaluation zeigte, dass Deep-Learning-Modelle eine signifikant höhere Robustheit und Genauigkeit bieten, insbesondere bei variablen Lichtverhältnissen und Hintergründen [41, 40]. Dennoch blieb das klassische Verfahren in Bezug auf Geschwindigkeit und Implementierungsaufwand konkurrenzfähig [31].

10.2 Verbesserungspotential

Auf Basis der bisherigen Ergebnisse lassen sich mehrere Verbesserungsmöglichkeiten ableiten:

- **Mehrklassen-Erkennung:** Aktuell erkennt die Implementierung ausschließlich Kugeln. Eine Erweiterung auf mehrere Objektklassen oder Farben wäre denkbar, etwa zur gleichzeitigen Erkennung und Unterscheidung mehrerer Objekte [28, 5].
- **Sensor-Fusion:** Die Kombination der visuellen Erkennung mit weiteren Sensoren (z. B. LiDAR, Inertial Measurement Unit (IMU)s oder Tiefenkameras) könnte die Robustheit bei schnellen Bewegungen und schlechter Sicht verbessern [8].
- **Edge-Optimierung:** Obwohl Echtzeitbetrieb auf Jetson-Systemen möglich ist, könnte durch Quantisierung und Pruning [14] die Latenz weiter reduziert werden.
- **Bessere Datensätze:** Der Trainingsdatensatz könnte durch gezielte Daten-

augmentation, synthetische Generierung oder aktives Lernen verbessert werden [30, 33, 29].

10.3 Weiterführende Arbeiten im Labor

Für den Einsatz im Hochschulkontext ergeben sich mehrere Erweiterungsideen:

- **Autonomes Kalibrieren:** Derzeit muss die Plattform manuell initialisiert werden. Eine automatische Kalibrierung über Bildverarbeitung wäre eine sinnvolle Erweiterung [39].
- **Simulationsumgebung:** Aufbau eines digitalen Zwillings mit Physics Engine (z. B. Gazebo [21]), um Plattformverhalten vorab zu testen und Trainingsdaten synthetisch zu erzeugen [33].
- **Multitasking-Kontrolle:** Erweiterung auf Aufgaben wie simultanes Tracking und Ausweichen, Sortieren oder Interaktion mehrerer Plattformen im Verbund [38].
- **Vergleich weiterer Modelle:** Evaluierung alternativer Echtzeit-Modelle wie EfficientDet [32] oder MobileNet-SSDLite [15].

Insgesamt bietet das Projekt eine fundierte Basis für zukünftige Anwendungen in Robotik, Embedded Vision und Edge AI – sowohl im Studium als auch im praktischen Einsatz.

Literatur

[1] Glenn Jocher et al. *YOLOv5*. https://github.com/ultralytics/yolov5. Accessed: 2025-07. 2020.

[2] Karl J Astrom und Tore Hagglund. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, 1995.

[3] Dana H. Ballard. „Generalizing the Hough Transform to Detect Arbitrary Shapes“. In: *Pattern Recognition* 13.2 (1981), S. 111–122.

[4] Dana H. Ballard. „Generalizing the Hough Transform to Detect Arbitrary Shapes“. In: *Pattern Recognition* 13.2 (1981), S. 111–122.

[5] Alexey Bochkovskiy, Chien-Yao Wang und Hong-Yuan Mark Liao. „YOLOv4: Optimal Speed and Accuracy of Object Detection“. In: *arXiv preprint arXiv:2004.10934* (2020).

[6] Gary Bradski und Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, Inc., 2008.

[7] John Canny. „A Computational Approach to Edge Detection“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 8.6 (1986), S. 679–698. DOI: 10.1109/TPAMI.1986.4767851.

[8] Jiayi Cao, Jiaming Zhang und Qiang Wu. „A Survey on Sensor Fusion in Autonomous Vehicles“. In: *IEEE Transactions on Intelligent Transportation Systems* (2022).

[9] Jae-Hyeok Choi u. a. „Real-Time Object Detection and Tracking Based on Embedded Edge Devices for Local Dynamic Map Generation“. In: *Electronics* 13.5 (2024), S. 811. DOI: 10.3390/electronics13050811. URL: https://www.mdpi.com/2079-9292/13/5/811.

[10] NVIDIA Corporation. *Jetson AGX Orin Developer Kit*. https://developer.nvidia.com/embedded/jets
agx-orin-devkit. 2023.

[11] Analog Devices. *ADXL345 3-Axis Digital Accelerometer*. https://www.analog.com/media/en/technic
documentation/data-sheets/adxl345.pdf. 2019.

[12] Zheng Ge u. a. „YOLOX: Exceeding YOLO Series in 2021“. In: *arXiv preprint arXiv:2107.08430* (2021).

[13] Haythem Ghazouani. „Shape and Color Object Tracking for Real-Time Robotic Navigation“. In: (2014). arXiv: 1410.3970.

[14] Song Han, Huizi Mao und William J. Dally. „Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding“. In: *arXiv preprint arXiv:1510.00149* (2015).

[15] Andrew G. Howard u. a. „MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications“. In: *arXiv preprint arXiv:1704.04861* (2017).

[16] Yu-Chuan Huang, I-No Liao u. a. „TrackNet: A Deep Learning Network for Tracking High-speed and Tiny Objects in Sports Applications“. In: *arXiv preprint arXiv:1907.03698* (2019).

[17] Muhammad Hussain. *YOLOv5, YOLOv8 and YOLOv10: The Go-To Detectors for Real-Time Vision*. 2024. arXiv: 2407.02988.

[18] Google Inc. *Google Colab – Machine Learning on the Web*. 2023. URL: https://colab.research.google.com.

[19] Microchip Technology Inc. *MCP4728: 12-Bit, Quad Digital-to-Analog Converter with EEPROM*. Datenblatt. Accessed: 2025-08. 2010. URL: https://ww1.microchip.com/downloads/en/devicedoc/22187e.pdf.

[20] Rahima Khanam und Muhammad Hussain. „What is YOLOv5: A Deep Look into the Internal Features of the Popular Object Detector“. In: *arXiv preprint arXiv:2407.20892* (2024).

[21] Nathan Koenig und Andrew Howard. „Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator“. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2004.

[22] Zachary C. Lipton. „The Mythos of Model Interpretability“. In: *Communications of the ACM* 61.10 (2018), S. 36–43.

[23] M. Liu u. a. „An Efficient Real-Time Object Detection Framework on Resource-Constrained Hardware Devices Through Tensor-Train Decomposition for Compressing YOLOv5“. In: *arXiv preprint arXiv:2408.01534* (2024).

[24] Wenyu Lv, Yian Zhao u. a. „RT-DETRv2: Improved Baseline with Bag-of-Freebies for Real-Time Detection Transformer“. In: *arXiv preprint arXiv:2407.17140* (2024).

[25] David Minott, Salman Siddiqui und Rami.J. Haddad. „Benchmarking Edge AI Platforms: Performance Analysis of NVIDIA Jetson and Raspberry Pi5 with Coral TPU“. In: *2025 IEEE SoutheastCon*. 2025, S. 1384–1389. DOI: 10.1109/southeastcon56624.2025.10971592.

[26] NVIDIA Corporation. *NVIDIA Jetson Nano Developer Kit*. 2023. URL: https://developer.nvidia.com/embedded/jetson-nano.

[27] Papers with Code. *Object Detection on COCO: YOLOv5 Benchmarks*. https://paperswithcode.com/sota/object-detection-on-coco. 2023.

[28] Joseph Redmon und Ali Farhadi. „YOLOv3: An Incremental Improvement“. In: *arXiv preprint arXiv:1804.02767* (2018).

[29] Ozan Sener und Silvio Savarese. „Active Learning for Convolutional Neural Networks: A Core-Set Approach“. In: *International Conference on Learning Representations (ICLR)*. 2018.

[30] Connor Shorten und Taghi M. Khoshgoftaar. „A Survey on Image Data Augmentation for Deep Learning“. In: *Journal of Big Data* 6.1 (2019), S. 1–48.

[31] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010. ISBN: 978-1-84882-934-3.

[32] Mingxing Tan und Quoc Le. „EfficientDet: Scalable and Efficient Object Detection“. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), S. 10781–10790.

[33] Jonathan Tremblay u. a. „Training Deep Networks with Synthetic Data: Bridging the Reality Gap by Domain Randomization“. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2018.

[34] Ultralytics. *YOLOv8*. https://github.com/ultralytics/ultralytics. Accessed: 2025-07. 2023.

[35] Heinz Unbehauen. *Regelungstechnik I: Klassische Verfahren zur Analyse und Synthese linearer kontinuierlicher Regelungssysteme*. Springer Vieweg, 2012.

[36] Shuo Wang u. a. „RT-DETRv3: Real-Time End-to-End Object Detection with Hierarchical Dense Positive Supervision“. In: *arXiv preprint arXiv:2409.08475* (2024).

[37] Tianxiao Zhang, Xiaohan Zhang, Yiju Yang u. a. „Efficient Golf Ball Detection and Tracking Based on Convolutional Neural Networks and Kalman Filter“. In: *arXiv preprint arXiv:2012.09393* (2020).

[38] Zhen Zhang, Yiqin Wang und Fei Yan. „Multi-agent Collaboration and Learning: A Review“. In: *Information Fusion* 66 (2021), S. 147–168.

[39] Zhengyou Zhang. „A Flexible New Technique for Camera Calibration“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.11 (2000), S. 1330–1334.

[40] Yian Zhao, Wenyu Lv, Shangliang Xu u. a. „DETRs Beat YOLOs on Real-time Object Detection“. In: *arXiv preprint arXiv:2304.08069* (2023).

[41] Z. Zhao u. a. „Object Detection With Deep Learning: A Review“. In: *IEEE Transactions on Neural Networks and Learning Systems* 30.11 (2019), S. 3212–3232.

[42] Andreas Ziegler, Karl Vetter u. a. „Spiking Neural Networks for Fast-Moving Object Detection on Neuromorphic Hardware Devices Using an Event-Based Camera“. In: *arXiv preprint arXiv:2403.10677* (2024).