Fakultät Informatik
INF6

TOPR

Abschlussbericht des teamorientierten Projekts

Bearbeitende:

Chantal Deusch, Jeremy Diem, Jan Gaschler, Serhat Gürel, Valentin Talmon-l'Armée, Paulina Pyczot

Fakultät für Informatik

Sommersemester 2025

Betreuung:

Prof. Dr. A. Franz, Prof. Dr. M. Schäffter, S. Schäbel, M. Balser

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Gesichts- und Schrifterkennungstechnologien gehören heute zu den zentralen Komponenten moderner automatisierter Systeme. Anstatt nur visuelle Daten aufzunehmen, können Maschinen durch diese Technologien spezifische Muster, Objekte oder Zeichen erkennen und klassifizieren. Dieser technologische Fortschritt hat das Potenzial, die zukünftige Entwicklung in Schlüsselbereichen wie der Medizin und der Verteidigungstechnologie maßgeblich zu beeinflussen - etwa durch präzisere Diagnoseverfahren, automatisierte Auswertung medizinischer Bilddaten oder den Einsatz intelligenter Erkennungssysteme in sicherheitsrelevanten Anwendungen und militärischen Szenarien. Trotz dieser Erfolge stoßen aktuelle Systeme immer noch an ihre Grenzen. Faktoren wie schlechte Bildqualität, komplexe Lichtverhältnisse, verdeckte Gesichter oder unterschiedliche Schriftarten können die Erkennungsgenauigkeit erheblich beeinträchtigen. Zusätzlich werfen Fragen des Datenschutzes und der ethischen Nutzung neue Herausforderungen auf, die nicht ignoriert werden dürfen. Statt vorgefertigte Lösungen zu liefern, möchten wir Studierenden einen einfachen und praxisnahen Zugang zu komplexen Technologien wie der Gesichts- und Schrifterkennung ermöglichen. Ziel ist es, Neugier zu wecken und ein tieferes Verständnis für intelligente Systeme zu schaffen.

1.2 Zielsetzung des Projekts

1.2.1 Primäres Ziel

Ziel dieses Projekts ist die Entwicklung eines praxisorientierten Hands-On Tutorials für internationale Studierende im Rahmen der THU Summer School. Im Fokus steht die Umsetzung grundlegender Methoden zur Personen- und Schrifterkennung unter kostengünstigen Laborbedingungen. Dabei kommt ein Raspberry Pi als preiswerte und kompakte Hardwareplattform zum Einsatz, die eine realistische Umsetzung auch unter begrenzten Ressourcen ermöglicht. Das Tutorial vermittelt nicht nur technische Grundlagen, sondern demonstriert anhand zweier konkreter Anwendungsbeispiele die Leistungsfähigkeit aktueller Computer-Vision-Technologien auf Basis der YOLOv5 und YOLOv8-Architektur:

1. **Schrifterkennung (OCR):** Die automatische Erkennung und das Auslesen von Kfz-Kennzeichen aus Bildmaterial mithilfe der Zeichenerkennung. Ziel ist es, die alphanumerischen Informationen aus Bildern zu extrahieren.

2. **Personenerkennung:** Die Erkennung und Identifikation von Personen über Gesichter in Bild- oder Videodaten. Neben der reinen Gesichtserkennung ermöglicht das System auch das Einlernen neuer Gesichter, sodass unbekannte Personen erfasst, gespeichert und bei späterem Auftreten wiedererkannt werden können.

Neben der Implementierung liegt ein besonderer Schwerpunkt auf der praktischen Anwendbarkeit in ressourcenbeschränkten Umgebungen sowie auf der kritischen Auseinandersetzung

mit den Grenzen und ethischen Fragestellungen dieser Technologien.

1.2.2 Sekundäres Ziel

Neben der Umsetzung im Laborumfeld dienen beide Erkennungsaufgaben als praxisnahe Demonstrationen potenzieller Anwendungsszenarien. Dabei steht nicht die Entwicklung eines marktreifen Produkts im Vordergrund, sondern die schrittweise Hinführung zu einem Verständnis für den praktischen Einsatz solcher Technologien.

Im Fall der Schrifterkennung wird ein automatisierter Mautprozess simuliert, bei dem Kfz-Kennzeichen erfasst und ausgewertet werden. Dieses Szenario vermittelt Einblicke in die Funktionsweise moderner Verkehrssysteme. Die Gesichtserkennung wird im Kontext einer Zutrittskontrolle umgesetzt, bei der autorisierte Personen anhand ihrer Gesichter identifiziert werden. Ziel ist es, exemplarisch aufzuzeigen, wie solche Systeme technisch realisiert werden können und welche Herausforderungen, in Bezug auf Genauigkeit oder Datenschutz, damit einhergehen.

Unser Projekt versteht sich nicht als Endpunkt, sondern als Einladung, weiterzudenken, zu experimentieren und sowohl die Potenziale als auch Grenzen intelligenter Systeme kritisch und kreativ zu erforschen.

2 Theoretische Grundlagen und Stand der Technik

2.1 Stand der Technik in der Objekterkennung

Die moderne Objekterkennung wird heute maßgeblich von Deep-Learning-Ansätzen, insbesondere *Convolutional Neural Networks* (CNNs), dominiert. Diese haben traditionelle Methoden, die auf handgefertigten Merkmalen wie HOG (*Histogram of Oriented Gradients*) basierten, hinsichtlich Genauigkeit und Flexibilität weitgehend abgelöst. CNNs lernen Merkmale direkt aus den Daten und ermöglichen dadurch robustere und besser generalisierbare Modelle [1].

Innerhalb der Deep-Learning-basierten Objekterkennung existieren zwei Hauptstrategien:

1. **Two-Stage Detectors:** Zunächst werden potenzielle Regionen (*Region Proposals*) generiert, in denen sich Objekte befinden könnten. Anschließend werden diese Regionen klassifiziert (z.B. Faster R-CNN). Diese Ansätze erreichen oft eine hohe Genauigkeit, sind jedoch tendenziell langsamer.

2. **One-Stage Detectors:** Hierbei wird die Objekterkennung als direktes Regressionsproblem formuliert. Bounding-Box-Koordinaten und Klassenwahrscheinlichkeiten werden in einem einzigen Durchlauf vorhergesagt. Bekannte Vertreter sind SSD (*Single Shot MultiBox Detector*) und insbesondere YOLO (*You Only Look Once*), die sich durch hohe Verarbeitungsgeschwindigkeit für Echtzeitanwendungen auszeichnen.

2.1.1 Bildverarbeitung und Computer Vision mit OpenCV

OpenCV (Open Source Computer Vision Library) ist eine weit verbreitete Open-Source-Bibliothek, die eine umfangreiche Sammlung von Algorithmen für die Bildverarbeitung und Computer Vision bereitstellt [2]. Sie wurde ursprünglich von Intel entwickelt und wird heute von einer großen Community gepflegt und weiterentwickelt. OpenCV bietet Funktionen für Aufgaben wie Objekterkennung, Gesichtserkennung, Merkmalsextraktion, Kamerakalibrierung und vieles mehr. Die Bibliothek ist in C++ implementiert, bietet aber auch Schnittstellen für Python, Java und andere Sprachen, was sie für eine breite Palette von Anwendungen zugänglich macht.

OpenCV ist eine frei verfügbare Softwarebibliothek, die viele Verfahren zur Bildverarbeitung und für Aufgaben der Computer Vision enthält. Sie ist Open Source und wird unter der Apache-2.0-Lizenz bereitgestellt. OpenCV kann mit verschiedenen Programmiersprachen wie C, C++, Python und Java verwendet werden. Die Bibliothek unterstützt eine Vielzahl plattformübergreifender Betriebssysteme und nutzt sowohl die CPU als auch die GPU, um eine besonders schnelle Bildverarbeitung zu ermöglichen. Entwickelt wurde OpenCV ursprünglich von Intel, das auch heute noch an der Weiterentwicklung beteiligt ist.

OpenCV spielt in vielen Bereichen eine zentrale Rolle – von der Steuerung autonomer Fahrzeuge bis hin zur Gesichtserkennung in mobilen Anwendungen. Besonders durch seine Fähigkeit, komplexe visuelle Daten in Echtzeit zu verarbeiten, ist die Bibliothek ein unverzichtbares Werkzeug in der Forschung und Entwicklung künstlicher Intelligenz, insbesondere im Bereich des maschinellen Lernens und Deep Learnings. Die von OpenCV bereitgestellten

Werkzeuge ermöglichen die Entwicklung fortschrittlicher Vision-Anwendungen, die früher als schwer umsetzbar oder nur mit großem Aufwand realisierbar galten [1].

2.1.1.1 Grundlagen

Die Bildverarbeitung bildet einen zentralen Bestandteil der Computer Vision. Sie befasst sich mit der gezielten Veränderung und Analyse von Pixelwerten in digitalen Bildern, um diese zu verbessern oder bestimmte Informationen daraus zu gewinnen. Zu den grundlegenden Konzepten der Bildverarbeitung zählen:

- **Pixel:** Die kleinste Einheit eines digitalen Bildes, die einzelne Farbinformationen enthält. Die Farbwerte eines Pixels hängen vom verwendeten Farbraum ab.
- **Bildauflösung:** Gibt die Anzahl der Pixel in einem Bild an und wird durch die Breite und Höhe in Pixeln beschrieben. Eine höhere Auflösung ermöglicht in der Regel eine detailliertere Darstellung von Bildinhalten.
- **Farbräume:** Sie definieren, wie Farben in einem Bild mathematisch dargestellt werden. Gängige Farbräume sind unter anderem RGB (Rot, Grün, Blau), HSV (Farbton, Sättigung, Helligkeit) und YCbCr, die je nach Anwendungsfall unterschiedliche Vorteile bieten.

2.1.1.2 Bildverarbeitungsfunktionen

Die Bildverarbeitung umfasst eine Vielzahl an Techniken zur Analyse und gezielten Veränderung digitaler Bilder. Zu den grundlegenden Funktionen zählen unter anderem:

- **Geometrische Transformationen:** Hierzu gehören Operationen wie Rotation, Skalierung und Verschiebung (Translation), mit denen die Form oder Ausrichtung eines Bildes verändert wird.
- **Farbraumkonversionen:** Dabei wird ein Bild von einem Farbraum in einen anderen überführt – zum Beispiel von RGB in HSV –, was häufig die Bildanalyse erleichtert oder die Bearbeitung effizienter macht.
- **Filteroperationen:** Durch den Einsatz spezieller Filter, wie dem Gaußschen Weichzeichner, Kantenfiltern oder Medianfiltern, lassen sich bestimmte Bildmerkmale hervorheben, Kanten verstärken oder Bildrauschen reduzieren.

2.1.1.3 Installation und Einrichtung der Umgebung

Die Installation von OpenCV hängt vom verwendeten Betriebssystem sowie von der bevorzugten Programmiersprache ab. Für Python-Anwender bietet sich der einfachste Weg über den Paketmanager `pip` an:

```
pip install opencv-python
```

Wird hingegen mit C++ gearbeitet, empfiehlt es sich, die Bibliothek direkt von der offiziellen OpenCV-Website herunterzuladen. Anschließend muss sie entsprechend den Anweisungen für

das jeweilige Betriebssystem kompiliert werden. Dabei ist darauf zu achten, dass alle erforderlichen Abhängigkeiten – insbesondere CMake und ein geeigneter Compiler – im Vorfeld korrekt installiert sind, um eine reibungslose Einrichtung sicherzustellen.

2.1.1.4 Grundlegende OpenCV-Funktionen und Methoden

OpenCV stellt eine umfangreiche Sammlung an Funktionen bereit, mit denen sich sowohl einfache als auch komplexe Aufgaben der Bildverarbeitung umsetzen lassen. Zu den grundlegenden Möglichkeiten zählen unter anderem:

- **Bild einlesen und speichern:** Mithilfe der Funktionen `cv2.imread()` und `cv2.imwrite()` können Bilder unkompliziert geladen und gesichert werden. Dabei werden zahlreiche Bildformate wie JPEG, PNG oder TIFF unterstützt.
- **Kamerazugriff:** OpenCV ermöglicht auch die Verarbeitung von Live-Bildern über angeschlossene Kameras. Durch die Nutzung der Klasse `cv2.VideoCapture` lassen sich Videoströme von Kameras oder auch von gespeicherten Videodateien erfassen und weiterverarbeiten.

2.1.1.5 Kernfunktionen

- **Bildskalierung:** Mit OpenCV lässt sich die Größe von Bildern anpassen, ohne das ursprüngliche Seitenverhältnis zu verändern. Die Funktion `cv2.resize()` ermöglicht es, ein Bild effizient auf eine neue Dimension zu skalieren.
- **Schwellenwertsetzung (Thresholding):** Dieser Prozess wandelt ein Bild in ein binäres Format um. OpenCV bietet eine Reihe von Schwellenwerttechniken, wie die einfache, adaptive und Otsu-Schwellenwertsetzung. Mit der Funktion `cv2.threshold()` lässt sich ein Bild optimal für die nächste Analysephase vorbereiten.
- **Kantenerkennung:** Die Erkennung von Kanten stellt einen wichtigen Schritt in vielen Computer-Vision-Anwendungen dar. OpenCV bietet verschiedene Algorithmen, um Kanten im Bild zu identifizieren, darunter den Sobel-Operator sowie den bekannten Canny-Kantendetektor.

2.1.1.6 Zusammenfassung

OpenCV ist eine leistungsstarke Bibliothek für die Bildverarbeitung und Computer Vision, die es Entwicklern ermöglicht, komplexe visuelle Aufgaben effizient zu lösen und eine Vielzahl von Anwendungsfällen in unterschiedlichen Branchen zu unterstützen.[3], [4]

2.1.2 YOLO als Basis für schnelle Detektion

YOLO unterteilt das Eingabebild in ein Gitter. Jede Zelle ist für die Erkennung von Objekten zuständig, deren Mittelpunkt in sie fällt. Für jede Zelle werden Begrenzungsrahmen, ein Konfidenzwert und Klassenwahrscheinlichkeiten vorhergesagt.

Seit der ersten Veröffentlichung wurden zahlreiche Weiterentwicklungen (YOLOv3, v4, v5, ..., v8 etc.) vorgestellt, die:

- die Architektur verfeinerten,
- die Genauigkeit, insbesondere bei kleinen Objekten, verbesserten und
- Techniken wie Feature Pyramid Networks zur besseren Merkmalsextraktion integrierten.

Die Stärke von YOLO liegt in seiner Geschwindigkeit. Daher eignet es sich besonders für die initiale Detektion von Objekten wie Gesichtern oder Kennzeichen in Videoströmen, wie auch in diesem Projekt genutzt.

2.1.3 MediaPipe Framework für Wahrnehmungsaufgaben

MediaPipe ist ein Open-Source-Framework von Google, das die Entwicklung plattformübergreifender Computer-Vision-Anwendungen erleichtert. Es ermöglicht effiziente Machine-Learning-Pipelines für Video-, Bild- und Audiodaten in Echtzeit. Dank modularer Architektur und vorgefertigter Komponenten (*Calculators*) können Entwickler schnell Prototypen erstellen und zu ausgereiften Anwendungen weiterentwickeln [5].

2.1.3.1 Graphbasierte Architektur Ein zentrales Merkmal von MediaPipe ist seine graphbasierte Architektur. Datenflüsse werden in *Graphs* definiert, wobei jeder Knotenpunkt spezifische Aufgaben übernimmt (z.B. Bilddekodierung, ML-Inferenz, Landmarken-Rendering). Diese Struktur ermöglicht eine flexible und effiziente Verarbeitung von Datenströmen, besonders für Echtzeitanwendungen.

2.1.3.2 MediaPipe Face Mesh: Technische Funktionsweise Die Gesichtserkennung und -analyse in MediaPipe erfolgt in mehreren, aufeinander aufbauenden Schritten:

1. **Gesichtsdetektion**: Zunächst kommt ein schneller Gesichtsdetektor (basierend auf BlazeFace) zum Einsatz, der ein einzelnes CNN verwendet, um Gesichter im Bild zu lokalisieren. Dieser Detektor ist für Mobilgeräte optimiert und identifiziert die grundlegende Position und Größe des Gesichts durch Bounding Boxes und sechs Schlüsselpunkte (Augen, Nasenspitze, Mundmitte).

2. **Ausrichtung und Normalisierung**: Das detektierte Gesicht wird anhand der Schlüsselpunkte normalisiert und ausgerichtet, um eine konsistente Eingabe für den nachfolgenden Landmarken-Detektor zu schaffen.

3. **Landmarken-Prediktion**: Ein spezialisiertes neuronales Netzwerk, basierend auf einer angepassten CNN-Architektur, verarbeitet das ausgerichtete Gesichtsbild und sagt die genauen Positionen von 468 3D-Landmarken voraus. Jede Landmarke wird als 3D-Punkt (x, y, z) kodiert, wobei x und y die Bildkoordinaten sind und z die relative Tiefe darstellt. Diese hohe Dichte an Landmarken ermöglicht eine äußerst detaillierte Abbildung der Gesichtsgeometrie.

4. **Refinement und Stabilisierung**: Um zeitliche Konsistenz bei Videoanalysen zu gewährleisten, verwendet MediaPipe Tracking-Algorithmen, die die Landmarken über aufeinanderfolgende Frames stabilisieren. Dies reduziert Flackern und erhöht die Genauigkeit der Bewegungserfassung.

Diese Pipeline, die Detektion, Ausrichtung, Landmarken-Prädiktion und Stabilisierung kombiniert, ermöglicht eine robuste und präzise Gesichtsanalyse in Echtzeit [6].

Die 468 Landmarken des Face Mesh erfassen präzise die Geometrie des gesamten Gesichts, einschließlich feiner Details wie:

- Augenkontur und Iris-Position für Eye-Tracking
- Lippen und Mundform für Lippenbewegungsanalyse
- Augenbrauen für Ausdruckserkennung
- Wangenknochen und Kinnlinie für Gesichtsstrukturanalyse
- Nasenkontur für die zentrale Ausrichtung

Diese detaillierte Erfassung ist eine Kernfunktion des MediaPipe Face Mesh Moduls [6].

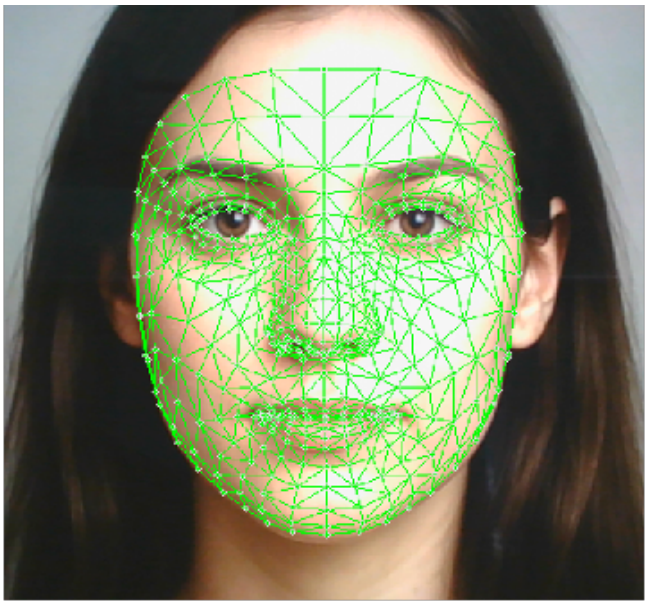


Figure 1: MediaPipe Face Mesh mit 468 Landmarken, die verschiedene Gesichtsbereiche präzise erfassen. Quelle: [6]

2.1.4 MediaPipe-Architektur und Face Mesh

MediaPipe ist eine Open-Source-Plattform von Google zur Erstellung plattformübergreifender Pipelines für multimodale Daten. Die Architektur basiert auf modularen Graphen, in denen einzelne Calculatoren Datenvorverarbeitung, ML-Modelle und Nachverarbeitungsschritte verknüpfen [5].

Das Face Mesh-Modul kombiniert zunächst einen schnellen BlazeFace-Detektor mit einem darauf aufsetzenden Landmarken-Vorhersagemodell und liefert in Echtzeit bis zu 468 3D-Landmarken pro Gesicht [6]. Die Topologie der Landmarken (siehe Abbildung 2) beschreibt die Kantenverbindungen zwischen den Punkten und ermöglicht eine präzise meshing-Repräsentation.

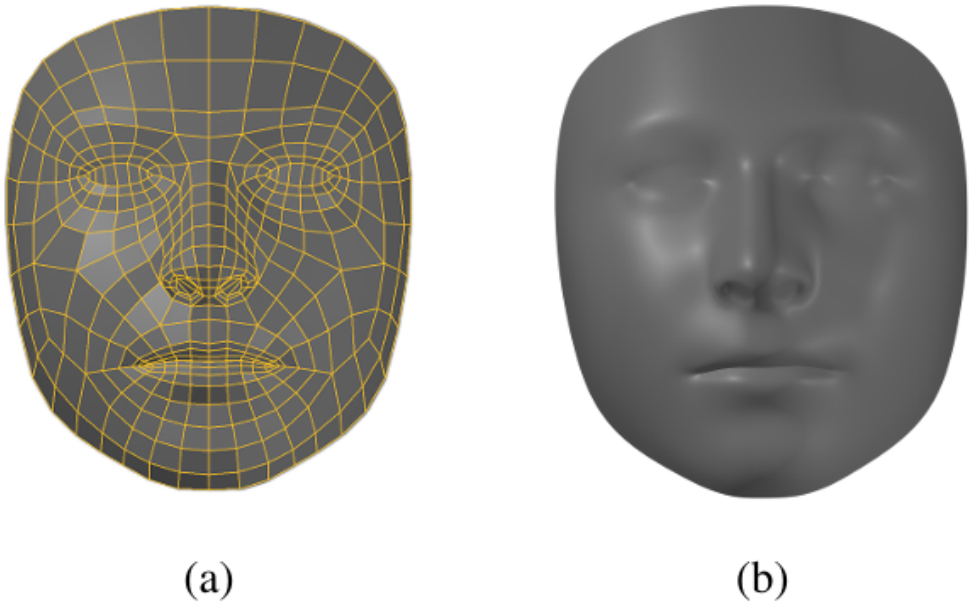


Figure 2: MediaPipe Face Mesh Topologie mit 468 Landmarken und Kantenverbindungen. Quelle: [7]

2.2 Funktionsweise von YOLO

YOLO (*You Only Look Once*) verfolgt einen effizienten Ansatz für die Objekterkennung, der sich durch seine hohe Geschwindigkeit auszeichnet. Anstatt das Bild in mehreren Schritten zu analysieren, betrachtet YOLO das gesamte Bild nur einmal (daher der Name) und sagt alle Objekte gleichzeitig vorher. Man kann sich das wie ein schnelles „Überfliegen“ des Bildes vorstellen.

Das Gitter (Grid):

Der Kern von YOLO ist die Aufteilung des Eingangsbildes in ein gedachtes Gitter, ähnlich einem Schachbrett (z.B. 13×13 oder 19×19 Zellen). Jede dieser Zellen bekommt eine spezielle Aufgabe: Sie ist dafür verantwortlich, Objekte zu erkennen, deren Mittelpunkt genau in diese Zelle fällt.

Vorhersagen pro Zelle:

Jede Zelle im Gitter macht mithilfe eines neuronalen Netzes Vorhersagen über mögliche Objekte:

- **Bounding Boxes:** Die Zelle schlägt eine oder mehrere „Boxen“ (Rechtecke) vor, die ein Objekt umschließen könnten. Für jede Box werden Position (x-, y-Koordinaten des Mittelpunkts), Größe (Breite w, Höhe h) und ein Konfidenzwert vorhergesagt. Dieser Konfidenzwert gibt an, wie sicher sich das Modell ist, dass sich überhaupt ein Objekt in der Box befindet und wie gut die Box passt.

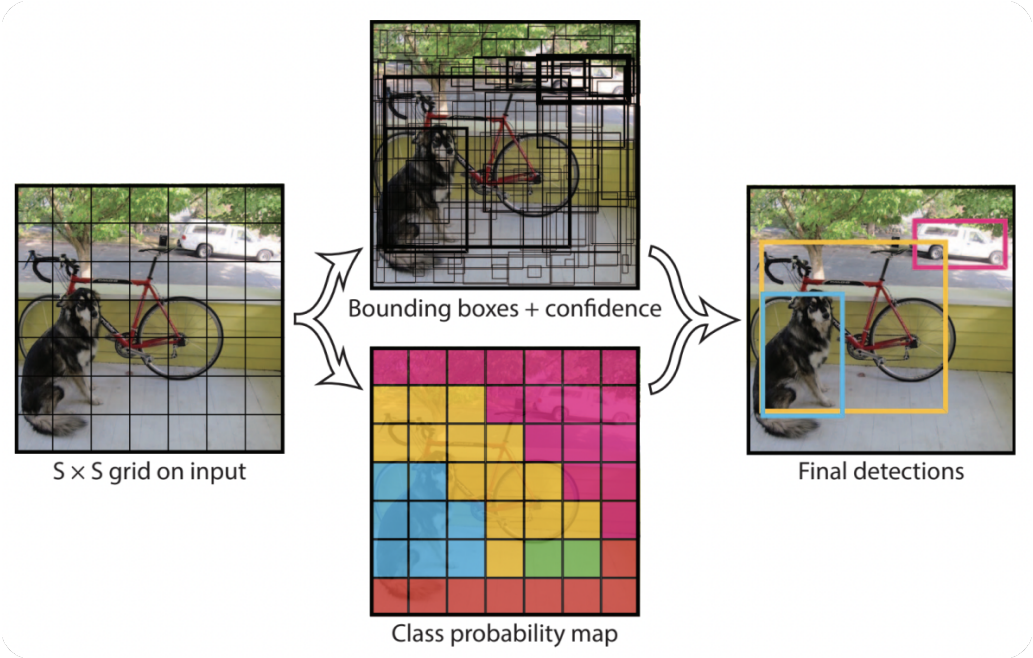


Figure 3: YOLO-Architektur: Das Bild wird in ein Gitter unterteilt, jede Zelle sagt Bounding Boxes und Klassenwahrscheinlichkeiten voraus. Quelle: [8]

- **Klassenwahrscheinlichkeiten:** Zusätzlich sagt die Zelle voraus, zu welcher Klasse (z.B. „Auto“, „Person“, „Kennzeichen“) ein erkanntes Objekt am wahrscheinlichsten gehört.

Architektur und Entwicklung:
Das Herzstück ist ein einzelnes Convolutional Neural Network (CNN), das diese Vorhersagen für alle Zellen gleichzeitig generiert. Frühe Versionen nutzten Architekturen wie Darknet. Spätere Versionen (YOLOv3 bis YOLOv8) wurden deutlich komplexer und leistungsfähiger, indem sie z.B. Merkmale aus verschiedenen Ebenen des Netzwerks kombinieren (Feature Pyramid Networks, FPN), um sowohl kleine als auch große Objekte besser zu erkennen. Auch wurden *Anchor Boxes* eingeführt – vordefinierte Box-Formen, die dem Netzwerk helfen, Objekte mit typischen Seitenverhältnissen schneller und genauer zu finden.

Aufräumen der Ergebnisse (Non-Max Suppression, NMS):
Da oft mehrere Zellen oder Boxen dasselbe Objekt erkennen, liefert YOLO zunächst viele überlappende Boxen. Um nur die relevanteste Box pro Objekt zu behalten, wird ein „Aufräumschritt“ namens Non-Max Suppression (NMS) durchgeführt. Dabei werden Boxen mit geringer Konfidenz entfernt und von den verbleibenden, überlappenden Boxen wird nur die mit der höchsten Konfidenz behalten, während die anderen verworfen werden.

Versionen:
YOLO wird ständig weiterentwickelt. In diesem Projekt kam hauptsächlich YOLOv8 zum Einsatz. Diese Version stellt einen aktuellen Stand der Technik dar und bietet eine gute Balance aus Erkennungsgenauigkeit und hoher Geschwindigkeit, was für die Analyse von Videoströmen vorteilhaft ist. Sie profitiert von den Architekturoptimierungen und Trainingstechniken früherer Versionen und ist oft einfacher zu verwenden. Die Entwicklung schreitet kontinuierlich voran, wie neuere, von Ultralytics dokumentierte Varianten wie YOLOv11 zeigen, was die Dynamik

in diesem Bereich unterstreicht [1].

2.3 OCR für Texterkennung

Optical Character Recognition (OCR) bezeichnet den Prozess der Umwandlung von Bildern, die getippten, gedruckten oder handgeschriebenen Text enthalten, in maschinenlesbaren Text. Im Kontext dieses Projekts ist OCR die Schlüsseltechnologie zur Extraktion der Zeichenfolgen aus detektierten Fahrzeugkennzeichen.

2.3.1 Typische OCR-Pipeline

Ein OCR-System durchläuft typischerweise mehrere Stufen: [9]

1. **Vorverarbeitung (Preprocessing):** Verbesserung der Bildqualität zur Optimierung der Erkennung. Dies kann Schritte wie Binarisierung (Umwandlung in Schwarz-Weiß), Rauschunterdrückung, Schräglagenkorrektur (Deskewing) und Kontrastanpassung umfassen.

2. **Layout-Analyse (oder Segmentierung):** Identifizierung von Textbereichen und deren Struktur. Bei strukturierten Dokumenten wie Kennzeichen kann dies die Segmentierung einzelner Zeichen oder Zeichengruppen umfassen. Bei allgemeineren Texten geht es darum, Absätze, Spalten und Zeilen zu erkennen.

3. **Zeichenerkennung (Character Recognition):** Dies ist der Kernschritt, bei dem die segmentierten Zeichen oder Wörter klassifiziert werden. Traditionelle Methoden nutzten Merkmalsextraktion (z.B. topologische Merkmale, Momente) und Klassifikatoren (z.B. SVMs, k-NN). Moderne Ansätze verwenden überwiegend tiefe neuronale Netze, insbesondere CNNs und Recurrent Neural Networks (RNNs), oft in Kombination (CRNN - Convolutional Recurrent Neural Network). RNNs, speziell LSTMs (Long Short-Term Memory), sind gut geeignet, um sequentielle Informationen in Wörtern oder Zeichenketten zu verarbeiten. Attention-Mechanismen können zusätzlich helfen, relevante Bildbereiche für jedes Zeichen zu fokussieren.

4. **Nachverarbeitung (Postprocessing):** Korrektur von Erkennungsfehlern unter Verwendung von Sprachmodellen, Wörterbüchern oder kontextuellen Informationen. Bei Kennzeichen kann dies die Überprüfung auf gültige Formate oder die Nutzung von Prüfsummen (falls vorhanden) beinhalten.

2.3.2 Methoden und Werkzeuge

Es existieren verschiedene OCR-Engines und Bibliotheken. Eine der bekanntesten Open-Source-Lösungen ist Tesseract OCR, ursprünglich von HP entwickelt und nun von Google

weitergeführt. Tesseract verwendet seit Version 4 LSTM-basierte neuronale Netze und bietet eine gute Erkennungsleistung für viele Sprachen und Schriftarten. Daneben gibt es kommerzielle Lösungen und spezialisierte Bibliotheken, die oft für spezifische Anwendungsfälle wie die Kennzeichenerkennung optimiert sind.

2.3.3 Anwendung bei der Kennzeichenerkennung (ANPR/LPR)

Automatic Number Plate Recognition (ANPR) oder License Plate Recognition (LPR) ist ein spezialisierter Anwendungsfall von OCR. Die typische Pipeline sieht oft wie folgt aus:

1. **Kennzeichen-Detektion:** Zunächst wird der Bereich des Kennzeichens im Bild lokalisiert. Hierfür eignen sich Objekterkennungsmodelle wie YOLO (siehe Abschnitt 2.2) hervorragend, die darauf trainiert werden, Kennzeichen als Objektklasse zu erkennen.
2. **Zeichensegmentierung:** Innerhalb des detektierten Kennzeichenbereichs werden die einzelnen Zeichen isoliert. Dies kann durch traditionelle Bildverarbeitungsalgorithmen (z.B. Konturanalyse, Projektionsprofile) oder durch spezialisierte neuronale Netze erfolgen.
3. **Zeichenerkennung:** Jedes segmentierte Zeichen wird durch ein OCR-Modul klassifiziert.

Alternativ können End-to-End-Ansätze verwendet werden, die Detektion und Erkennung in einem einzigen Netzwerk kombinieren oder direkt die gesamte Zeichenkette ohne explizite Segmentierung erkennen.

2.3.4 Herausforderungen

Die OCR von Kennzeichen unterliegt spezifischen Herausforderungen wie:

- variierenden Lichtverhältnissen (Tag/Nacht, Schatten, Blendung),
- unterschiedlichen Kameraperspektiven und -abständen,
- Verschmutzung oder Beschädigung der Kennzeichen,
- verschiedenen Kennzeichenlayouts und Schriftarten (je nach Land/Region),
- Bewegungsunschärfe

Die Robustheit des OCR-Systems gegenüber diesen Faktoren ist entscheidend für die praktische Anwendbarkeit.

2.4 Gesichtserkennung mit neuronalen Netzen

Die Gesichtserkennung ist ein Spezialfall der Objekterkennung und der biometrischen Identifikation, die in den letzten Jahren durch den Einsatz neuronaler Netze erhebliche Fortschritte erzielt hat. Sie umfasst typischerweise mehrere Schritte: Gesichtsdetektion, Vorverarbeitung, Merkmalsextraktion und Vergleich oder Klassifikation.

2.4.1 Technologien im Detail

2.4.1.1 Gesichtsdetektion Wie in Abschnitt 2.1 und 2.2 beschrieben, können Modelle wie YOLO hierfür eingesetzt werden. Alternativ nutzen viele Frameworks dedizierte, oft schnellere Detektoren (z.B. basierend auf SSD oder spezialisierten Architekturen).

2.4.1.2 Landmarken-Detektion Frameworks wie Google's MediaPipe bieten spezialisierte Modelle zur Detektion von hunderten Gesichtslandmarken in Echtzeit (z.B. das Face Mesh mit 468 3D-Landmarken). Diese Landmarken sind nützlich für die Gesichtsausrichtung, Mimik- und Augenbewegungsanalyse (z.B. Eye Tracking) sowie die Extraktion biometrischer Merkmale. Sie können als Merkmalsvektor dienen, wobei für die Identifikation meist Embeddings aus tieferen CNNs verwendet werden.

2.4.1.3 Biometrische Identifikation und Verifikation In der biometrischen Gesichtserkennung werden individuelle physiologische Gesichtsmerkmale wie Gesichtsform, Abstand der Augen, Ausprägung der Nasen- und Mundpartie sowie weitere geometrische Beziehungen genutzt, um Personen zu identifizieren oder zu verifizieren. Diese Merkmale werden in mathematischen Repräsentationen (Gesichts-Embeddings) kodiert, die für jede Person einzigartig sind. Der Vorteil biometrischer Systeme ist, dass sie auf inhärente Eigenschaften zurückgreifen, die nicht wie Passwörter vergessen oder wie Ausweise verloren werden können. Neurale Netzwerke haben dabei entscheidende Verbesserungen in der Zuverlässigkeit und Robustheit gebracht, insbesondere gegenüber Variationen in Beleuchtung, Gesichtsausdruck und Alterung.

2.4.1.4 Embedding-Generierung Bibliotheken wie `face_recognition` (basierend auf dlib) vereinfachen die Implementierung von Gesichtserkennungspipelines. Sie kapseln Detektion, Landmarken-Findung und Embedding-Generierung mithilfe vortrainierter Modelle (z.B. ResNet-basiert) und liefern 128-dimensionale Embeddings für den Vergleich. Diese hochdimensionalen Vektoren kodieren die unverwechselbaren Merkmale eines Gesichts in einem mathematischen Raum, in dem die euklidische Distanz zwischen Embeddings als Ähnlichkeitsmaß verwendet wird. Für die Verifikation wird typischerweise ein Schwellenwert festgelegt – liegt die Distanz unter diesem Wert, gilt das Gesicht als verifiziert.

2.4.1.5 Gesichtsliveness-Erkennung Um Sicherheitsrisiken durch Präsentationsangriffe (z.B. Verwendung von Fotos oder Videos) zu minimieren, implementieren moderne biometrische Systeme Liveness-Erkennungstechniken. Diese können aktive Methoden (Aufforderung zu Bewegungen wie Blinzeln oder Kopfdrehen) oder passive Methoden (Analyse von Mikrotexturen der Haut, Erfassung unwillkürlicher Bewegungen oder Erkennung von Reflexionseigenschaften) umfassen. Moderne KI-Systeme kombinieren diese Ansätze mit tiefen neuronalen Netzen, die Echtzeitanalysen der Bilddaten ermöglichen und die Sicherheit biometrischer Anwendungen erhöhen.

3 Anforderungen

3.1 Funktionale Anforderungen

- **Kennzeichenerkennung:**Im Rahmen eines Hands-On-Tutorials wird eine KI-gestützte automatisierte Kennzeichenerkennung umgesetzt. Ausgedruckte Kennzeichen werden von dem System in Form von Bildmaterial aufgenommen und ausgelesen. Anhand einfacher Codebeispiele und Echtzeit-Tests können die Teilnehmenden nachvollziehen, wie zuverlässig oder fehleranfällig solche Systeme in der Praxis sind. Dabei liegt trotz Laborumgebung auch ein Fokus auf realistische Bedingungen wie unscharfe Aufnahmen oder schlechte Lichtverhältnisse. Die Studierenden sollen selbst ausprobieren, wie die Technik funktioniert, und kritisch reflektieren, wann und warum Fehler passieren.

- **Gesichtererkennung:**Als weiterer Bestandteil des Hands-On-Tutorials wird eine KI-gestützte Personenerkennung mit Gesichtsunterscheidung implementiert, um Studierenden einen praktischen Einstieg in die Welt der künstlichen Intelligenz zu ermöglichen. Das System lernt Gesichter von verschiedenen Personen, unterscheidet diese voneinander und erkennt auch Unbekannte Gesichter als nicht bekannte Personen. Den Studenten wird der Umgang sowohl mit dem YOLO- als auch dem Mediapipe Modell gelehrt.

- **Benutzerinteraktion und Visualisierung:**Die visuelle Rückmeldung der KI-Modelle spielt eine zentrale Rolle. Die Ergebnisse werden in den Bildaufnahmen und dem Videostream durch Bounding Boxes, Beschriftungen und farblichen Markierungen verdeutlicht. Dadurch ist es den Studenten ermöglicht die Vorgehensweise der KI-Modelle visuell zu erkennen und ein tieferes Verständnis hinter den Technischen Grundlagen zu erlangen.

3.2 Nicht-funktionale Anforderungen

- **Benutzerfreundlichkeit und Umsetzbarkeit:**Das Hands-On-Tutorial soll für Internationale Studenten gut verständlich und nachvollziehbar sein. Die eingesetzten KI-Modelle sollen so integriertsein, dass sie für Studierende mit unterschiedlichen Vorkenntnissen leicht zugänglich sind.

- **Zuverlässigkeit:**Das System soll auch bei ungeeigneten oder fehlerhaften Eingaben stabil bleiben und dem Nutzer Rückmeldung geben, ohne abzustürzen. Fehler- oder Warnmeldungen sollen verständlich formuliert sein und zur Problemlösung beitragen.

- **Skalierbarkeit:**Das Tutorial soll, solange die Hardwarezugänglichkeit es zulässt, skalierbar sein und für eine unbegrenzte Anzahl an Teilnehmern verfügbar sein.

- **Hardwarelimitierung:**Das System soll auf leistungsschwacher Hardware wie einem Raspberry Pi lauffähig ist. Ziel ist es nicht nur eine kostengünstige Option für Laborbedingen zu schaffen, sondern auch ein auch ein Gefühl für Tiny Machine Learning schaffen.

3.3 User Stories

- Als Studierender möchte ich in einem verständlich aufgebauten Tutorial praxisnah in das Thema KI einsteigen
- Als Studierender möchte ich eine klar strukturierte Einführung in Modelle zur Kennzeichen- und Gesichtserkennung erhalten, um deren Funktionsweise und Einsatzmöglichkeiten nachvollziehen zu können.
- Als Studierender möchte ich auch außerhalb der Lehrveranstaltung Zugriff auf das Projekt haben, damit ich selbstständig weiterarbeiten und mein Wissen vertiefen kann.
- Als Student möchte ich die Grenzen intelligenter Systeme erleben, um ein realistisches Verständnis für aktuelle KI-Technologien zu entwickeln.
- Als Lehrender möchte ich, dass das Tutorial auf einfacher Hardware wie dem Raspberry Pi funktioniert, damit ich reale Einsatzgrenzen sowie praxisnahe Beispiele aus dem Bereich Tiny Machine Learning vermitteln kann.
- Als Lehrender möchte ich, dass das Tutorial die Lernenden zur kritischen Reflexion anregt. Sie sollen nicht nur die Funktionen nutzen, sondern auch die dahinterliegenden Konzepte und Grenzen verstehen.

4 Methodik

4.1 Kennzeichenerkennung

Die Kennzeichenerkennung ist ein Verfahren der Bildverarbeitung, das zur Erkennung und Interpretation von Fahrzeugkennzeichen auf Bildern oder Videoaufnahmen eingesetzt wird. Sie findet Anwendung in zahlreichen Bereichen wie der Verkehrsüberwachung, Mauterfassung, Parkraumbewirtschaftung oder Zufahrtskontrolle.

Das System basiert im Wesentlichen auf drei zentralen Prozessschritten: der Bilderfassung, der Lokalisation des Kennzeichens im Bild sowie der Texterkennung. Zunächst wird ein Bild eines Fahrzeugs aufgenommen, durch eine Kamera. Im nächsten Schritt wird das Kennzeichen im Bild identifiziert dies geschieht mithilfe moderner Objekterkennungsalgorithmen wie YOLO oder klassischer Verfahren der Konturenerkennung. Nachdem der relevante Bildausschnitt extrahiert wurde, folgt die optische Zeichenerkennung, bei der die alphanumerischen Zeichen des Kennzeichens ausgelesen und digital erfasst werden.

Die Genauigkeit der Kennzeichenerkennung hängt stark von der Qualität der Vorverarbeitung ab, bei der Methoden wie Graustufenumwandlung, Rauschfilterung, Binarisierung und Kantenanalyse eingesetzt werden, um ein möglichst klares Bild für die Texterkennung zu erzeugen. Moderne Systeme kombinieren dabei klassische Bildverarbeitung mit maschinellem Lernen, um auch bei schwierigen Lichtverhältnissen, verzerrten Perspektiven oder verunreinigten Kennzeichen verlässliche Ergebnisse zu erzielen.

4.1.1 Systemarchitektur und Funktionsweise

Das implementierte System zur Kennzeichenerkennung basiert auf einer Kombination aus Hardware- und Softwarekomponenten. Für die grundlegende Funktionsfähigkeit werden folgende Hardware-Voraussetzungen benötigt: ein Raspberry Pi oder ein funktionsfähiger Computer, eine handelsübliche oder integrierte Webcam, eine SD-Karte mit dem entsprechenden Betriebssystem-Image für den Raspberry Pi sowie eine Tastatur, Maus und ein HDMI-Kabel zur Verbindung mit einem Monitor. Alternativ können Tastatur, Maus und HDMI-Kabel entfallen, wenn der Raspberry Pi über den RealVNC Viewer ferngesteuert und eingerichtet wird.

Auf der Softwareseite kommen verschiedene Technologien zum Einsatz. Zur Objekterkennung wird das Modell YOLOv5 verwendet, während OpenCV zur Bildverarbeitung und Tesseract OCR für die Texterkennung eingesetzt werden.

Der Ablauf der Kennzeichenerkennung erfolgt in drei Schritten:
1. Bilderfassung: Die Kamera nimmt acht Bilder auf, die lokal auf dem Gerät gespeichert werden.
2. Erkennung: Die gespeicherten Bilder werden mithilfe von YOLOv5 analysiert. Die erkannten Kennzeichenbereiche (Bounding Boxes) werden markiert und die Bilder erneut gespeichert.
3. Texterkennung: Im letzten Schritt erfolgt die optische Zeichenerkennung (OCR) mit Tesseract. Dabei werden die Buchstaben extrahiert, separat gespeichert und das erkannte Kennzeichen im Konsolenfenster ausgegeben.

Das System wurde speziell am Beispiel für den Einsatz an europäischen Mautstationen konzipiert und optimiert.

4.1.2 Training und Optimierung des Modells

Für die Erkennung der Kennzeichenbereiche wurde bewusst das Modell YOLOv5 eingesetzt. Dieses Modell bietet eine ausgewogene Kombination aus Genauigkeit und Effizienz und eignet sich besonders gut für ressourcenbeschränkte Umgebungen wie den Raspberry Pi. Zwar existieren mittlerweile neuere und theoretisch leistungsfähigere Modellversionen, doch diese bringen meist deutlich höhere Anforderungen an Rechenleistung und Speicher mit sich, was eine stabile Ausführung auf einem Mikrocomputer wie dem Raspberry Pi erschwert oder sogar unmöglich macht. Gerade im Hinblick auf die beschränkten Hardwarekapazitäten des Raspberry Pi ist YOLOv5 insbesondere in den Varianten „nano“ oder „small“ eine sinnvolle Wahl. Diese Varianten sind speziell darauf ausgelegt, auch auf schwächerer Hardware zuverlässig zu funktionieren und dabei dennoch eine ausreichend hohe Erkennungsgenauigkeit zu gewährleisten. In der Praxis zeigt sich, dass mit YOLOv5 eine robuste Objekterkennung in nahezu Echtzeit möglich ist, ohne den Systembetrieb zu beeinträchtigen.

Zur Detektion von KFZ-Kennzeichen im Bildmaterial kommt in unserem Projekt ein speziell trainiertes YOLOv5-Modell zum Einsatz. YOLO ist eine auf Echtzeit ausgelegte Objekterkennungsarchitektur, die es ermöglicht, Objekte wie Nummernschilder direkt in Bildern zu lokalisieren und zu klassifizieren. Wir haben uns dabei für die besonders kompakte Variante YOLOv5n (nano) entschieden, da diese für ressourcenschwache Systeme wie den Raspberry Pi 4 optimiert ist und eine hohe Inferenzgeschwindigkeit bei geringer Modellgröße bietet.

4.1.2.1 Datengrundlage Als Trainingsgrundlage diente ein öffentlich zugänglicher Datensatz aus Roboflow Universe mit rund 3.000 annotierten Bilder von KFZ-Kennzeichen. Der Datensatz enthält bereits diverse Perspektiven, Lichtverhältnisse und Rotationen, was zur Robustheit des Modells beiträgt.

Zusätzlich wurden während des Trainings verschiedene Datenaugmentierungen vorgenommen, um die Generalisierungsfähigkeit weiter zu verbessern. [10] Hierzu zählten unter anderem:

- Auto-Orientierung zur Korrektur der Bildausrichtung
- Resize auf ein einheitliches Format von 640x640 Pixel
- Rauschüberlagerung (bis zu 1,6% der Pixel)
- Unschärfeeffekte (bis zu 1,6% Pixel Blur)

Diese Maßnahmen wurden mit Roboflow automatisiert vorgenommen. Das Modell wurde anschließend in Google Colab auf einer GPU-Instanz trainiert. Die Trainingsdauer betrug etwa 1,5 bis 2 Stunden. Als Hyperparameter wurden 50 Epochen mit einer Batch-Größe von 10 verwendet. Die restlichen Parameter (zum Beispiel Lernrate) blieben auf den Standardwerten des YOLOv5-Frameworks.

4.1.2.2 Trainingsergebnisse und Validierung Zur Bewertung des Trainingsverlaufs wurden typische Metriken von YOLOv5 analysiert. Abbildung 4 zeigt die Entwicklung der Kennzahlen über den Verlauf von 50 Trainings-Epochen.

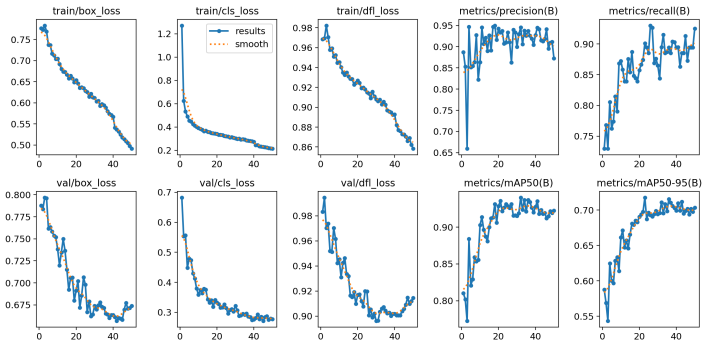


Figure 4: Trainingsergebnisse des YOLOv5-Modells.

Zu erkennen ist, dass sowohl die Trainings- als auch Validierungsverluste (box_loss, cls_loss, dfl_loss) im Verlauf kontinuierlich abnehmen, was auf ein stabiles und erfolgreiches Training hindeutet. Besonders der Klassifikationsverlust (cls_loss) zeigt eine schnelle Konvergenz in den ersten zehn Epochen. Die Präzision und der Recall pendeln sich im oberen Bereich ein – teils mit leichten Schwankungen, die jedoch durch das kleine Modell und die geringe Batch-Größe erklärbar sind.

Besonders relevant ist der mean Average Precision (mAP) – ein Maß für die Genauigkeit bei verschiedenen IoU-Schwellenwerten. Der Wert mAP@0.5 steigt im Verlauf des Trainings auf über 0,9, während mAP@0.5:0.95 einen stabilen Endwert um 0,70 erreicht. Dies spricht für eine hohe Detektionsqualität, auch bei komplexeren Szenarien.

4.1.2.3 Konfidenz und Modellanwendung Die Ausgabe eines YOLO-Modells enthält neben den erkannten Objektpositionen auch sogenannte Konfidenzwerte, also Wahrscheinlichkeiten, mit denen das Modell ein Objekt als erkannt einstuft. In unserem Projekt wurde ein Konfidenzschwellenwert definiert, um falsch-positive Erkennungen zu reduzieren. Nur Detektionen mit einer Konfidenz über diesem Schwellenwert wurden weiterverarbeitet.

Das trainierte YOLOv5-Modell im .pt-Format wurde zur Performanzoptimierung in das NCNN-Format konvertiert. NCNN ist ein hochoptimiertes Inferenz-Framework speziell für mobile und eingebettete Systeme. Dadurch konnte das Modell auf dem Raspberry Pi 4 mit hoher Effizienz ausgeführt werden. Die Anwendung läuft in einem Skript, das serienweise acht Bilder aufnimmt und analysiert – was nahezu Echtzeitverarbeitung ermöglicht.

4.1.2.4 Einfluss der Trainingsdaten und Verbesserungsmöglichkeiten Die Qualität und Diversität der Trainingsdaten hat direkten Einfluss auf die Erkennungsleistung. In unserem Fall war der Datensatz breit gefächert, jedoch auf Kennzeichen fokussiert. Mögliche Optimierungen für die Zukunft wären:

- Erweiterung des Datensatzes um reale eigene Aufnahmen
- Feinjustierung der Augmentierungstechniken
- Transfer Learning mit größeren Basismodellen
- Active Learning durch gezieltes Nachtraining mit fehlerhaften Beispielen

Dadurch könnte die Robustheit des Modells gezielt für Spezialfälle wie Nachtaufnahmen, verschmutzte Kennzeichen oder ungewöhnliche Kamerawinkel verbessert werden.

4.1.3 Image Preprocessing

Die Bildvorverarbeitung dient der Verbesserung der visuellen Eigenschaften der Bildausschnitte, in denen die Nummernschilder enthalten sind. Ziel ist es, die Zeichenstrukturen klar hervorzuheben, Bildrauschen zu minimieren und die Voraussetzungen für eine spätere Weiterverarbeitung, beispielsweise durch Texterkennung, zu schaffen.

Der erste Schritt der Pipeline besteht in der Umwandlung des Originalbildes in ein Graustufenbild. Dieses wird anschließend um den Faktor 3 vergrößert, um relevante Details, insbesondere feine Strukturen von Zeichen, deutlicher hervorzuheben. Im Anschluss erfolgt eine Glättung mittels eines Gaußschen Weichzeichners mit einer Kernelgröße von 3×3, um Bildrauschen zu unterdrücken.

Darauf folgt ein binäres Thresholding mit inversierter Darstellung (`cv2.THRESH_BINARY_INV`). Durch diese Umkehrung erscheinen potenzielle Zeichenbereiche weiß auf schwarzem Hintergrund, was die spätere Konturerkennung erleichtert.

Um benachbarte Strukturen zu verbinden und kleinere Lücken in Zeichenformen zu schließen, wird im nächsten Schritt eine morphologische Dilation durchgeführt. Hierbei wird ein rechteckiges Strukturierungselement der Größe 3×3 verwendet.

Die vorbereiteten Bilder werden im Anschluss zur Konturerkennung verwendet, wobei die Konturen in der Regel einzelnen Zeichen oder Zeichenkomponenten entsprechen. In späteren Verarbeitungsschritten können diese Bereiche gezielt extrahiert und analysiert werden.

Durch diese Kette an Vorverarbeitungsschritten wird sichergestellt, dass relevante Bildinformationen betont und störende Einflüsse reduziert werden, um die Qualität der anschließenden Bildanalyse zu maximieren.

4.1.4 Optical Character Recognition

Nach Abschluss der Bildvorverarbeitung, bei der die Kennzeichenbilder in eine geeignete Form gebracht wurden, erfolgt die Erkennung der enthaltenen alphanumerischen Zeichen mittels Optical Character Recognition (OCR). Zu Beginn des Projekts wurden dazu verschiedene OCR-Engines untersucht, darunter auch EasyOCR. Diese Deep-Learning-basierte Bibliothek liefert in vielen Anwendungsfällen gute Ergebnisse, zeigte jedoch im praktischen Einsatz auf ressourcenschwacher Hardware wie dem Raspberry Pi 4 deutliche Einschränkungen. Der hohe Speicherbedarf und die langen Inferenzzeiten verhinderten eine Echtzeitverarbeitung.

Im Vergleich dazu bot Tesseract eine deutlich geringere Systemlast bei gleichzeitig stabiler Erkennungsleistung. Besonders in unserem Anwendungsfall – kontrastreiche, segmentierte Kennzeichenbilder – erwies sich Tesseract als ausreichend genau. Durch gezielte Konfigurationsmöglichkeiten, wie die Beschränkung auf bestimmte Zeichensätze und Layout-Modi, ließ sich die Leistung zusätzlich optimieren. Aus diesen Gründen fiel die Entscheidung auf die Kombination aus Tesseract und der Python-Bibliothek `pytesseract`.

Im nächsten Schritt der Pipeline erfolgt eine Konturanalyse auf dem vorbereiteten binären Bild. Die erkannten Konturen werden nach verschiedenen geometrischen Kriterien gefiltert, um nur relevante Regionen weiterzuverarbeiten – etwa anhand von Größe, Seitenverhältnis und Position. Für jede verbleibende Kontur wird ein leicht vergrößerter Bildausschnitt (Region of Interest) definiert. Dieser wird an die OCR-Engine übergeben.

Die Texterkennung wird über die Funktion `image_to_data` durchgeführt. Dabei wird der zu analysierende Zeichensatz auf Großbuchstaben und Ziffern eingeschränkt (`tessedit_char_whitelist`) und mit dem Seitenlayout-Modus `-psm 8` gearbeitet, der für einzelne Zeichen geeignet ist. Das Ergebnis umfasst neben dem erkannten Text auch Positionsdaten und Konfidenzwerte.

Abschließend werden die erkannten Zeichenbereiche im Graustufenbild visuell hervorgehoben und als Bilddateien gespeichert, um die Ergebnisse nachvollziehbar und prüfbar zu machen. [11]

4.1.5 Herausforderungen bei realen Bedingungen

Im Rahmen der Entwicklung unseres Kennzeichenerkennungssystems, das auf eine Mautstationssituation ausgelegt ist, traten verschiedene Herausforderungen auf, die unter realen Bedingungen von besonderer Bedeutung sind.

Eine der ersten Schwierigkeiten bestand in der Vielfalt von Kfz-Kennzeichen weltweit. Während die Objekterkennung des Kennzeichens selbst (die Lokalisierung auf dem Bild) zuverlässig funktionierte, erwies sich das anschließende Auslesen der Schriftzeichen als problematisch. Dies lag an den unterschiedlichen Formaten, Schriftarten und Normen, die international variieren. Nach eingehender Analyse kamen wir zu dem Entschluss, dass die Entwicklung eines universellen Kennzeichenerkenners, der alle internationalen Standards abdeckt, den Rahmen unseres Projektes deutlich überschreiten würde. Stattdessen entschieden wir uns, eine optimierte Lösung speziell für deutsche Kfz-Kennzeichen zu entwickeln, um innerhalb der gegebenen Projektgrenzen eine funktional stabile Erkennung zu gewährleisten.

Eine weitere Herausforderung stellte die Anpassung des Systems an verschiedene Licht- und Schattenbedingungen dar. Schon bei kleinen aber vor allem bei stark wechselnden Beleuchtungsverhältnissen war die Festlegung geeigneter Thresholds bei der Bildverarbeitung für eine universale Lösung herausfordernd. Auch hier zeigte sich, dass eine umfassende allgemeine Lösung, die alle möglichen Umgebungsbedingungen robust abdeckt, einen erheblichen zusätzlichen Entwicklungsaufwand erfordern würde.

Zudem stießen wir auf technische Grenzen hinsichtlich der Leistungsfähigkeit des eingesetzten Raspberry Pi. Insbesondere die Schriftzeichenerkennung stellte hohe Anforderungen an die Prozessorleistung, was die Echtzeitfähigkeit des Systems deutlich beeinträchtigte. Es wurde schnell klar, dass der Raspberry Pi vor allem für den Einsatz unter kontrollierten Laborbedingungen geeignet ist, während für Anwendungen unter realen Bedingungen leistungsstärkere Hardware empfohlen wird. Die Herausforderung lag hier insbesondere darin, eine akzeptable Balance zwischen Hardwareanforderungen und Systemleistung herzustellen.

Vor diesem Hintergrund entschieden wir uns bewusst für die Verwendung der YOLOv5-Architektur, da diese im Vergleich zu neueren Modellen wie YOLOv8 eine deutlich geringere Rechenlast aufweist und damit besser auf ressourcenbeschränkter Hardware wie dem Raspberry Pi lauffähig ist. Die Wahl eines leichteren Modells ermöglichte es, die grundlegenden Funktionen der Gesichts- und Schrifterkennung trotz der beschränkten Systemressourcen zuverlässig zu demonstrieren.

4.1.6 Optimierung für wechselnde Lichtverhältnisse und Kameraperspektiven

Zur Verbesserung der Erkennungsstabilität unter realen Bedingungen wird besonderes Augenmerk auf die Bildverarbeitung gelegt. Ein vielversprechender Ansatz besteht in der Nutzung mehrstufiger Preprocessing-Pipelines. Hierbei wird dasselbe Eingangsbild mehreren parallel

ausgeführten Vorverarbeitungsschritten unterzogen, die jeweils auf typische Problemquellen wie Überbelichtung, Schatten oder Kontrastarmut optimiert sind. Die resultierenden Varianten werden anschließend bewertet, um die am besten geeignete für die Texterkennung auszuwählen.

Darüber hinaus kann perspektivische Verzerrung durch geometrische Transformationen, etwa mithilfe von Homographie-Matrizen, reduziert werden. Diese Maßnahme ist besonders bei schräg aufgenommenen Kennzeichenbildern sinnvoll, wie sie bei ungünstiger Kameraposition oder ungleichmäßiger Fahrzeugausrichtung entstehen.

Als mögliche Erweiterung wäre ein Postprocessing-Konzept denkbar, bei dem erkannte Zeichenfolgen hinsichtlich ihrer strukturellen Korrektheit überprüft werden. In professionellen Systemen wird dies häufig durch Plausibilitätsprüfungen oder den Vergleich mehrerer Verarbeitungsergebnisse realisiert. Dabei kommen Algorithmen zum Einsatz, die verschiedene OCR-Ergebnisse anhand formaler Merkmale, Zeichensatzvalidität und Konfidenzwerten bewerten, um das wahrscheinlich korrekteste Resultat zu ermitteln. Denkbar wäre zudem der Einsatz eines speziell trainierten KI-Modells, das ausschließlich auf die Erkennung von Schriftarten in KFZ-Kennzeichen ausgelegt ist. Ein solches Modell könnte Unsicherheiten der allgemeinen OCR durch domänenspezifisches Wissen kompensieren und die Gesamterkennungsrate unter realen Bedingungen deutlich verbessern. [12]

4.1.7 Evaluation und Genauigkeitsanalyse

4.2 Gesichtserkennung

4.2.1 Testaufbau

Das implementierte System realisiert eine Echtzeit-Gesichtserkennung mittels einer handelsüblichen Webcam (640×480 Pixel) und verwendet das Modell `YOLOv8n-face` zur Detektion von Gesichtern. Die Identifikation erfolgt durch Kombination der YOLO-basierten Gesichtserkennung mit dem `face_recognition`-Framework, das HOG-basierte Merkmalsextraktion nutzt. Bekannte Gesichter werden im lokalen Verzeichnis gespeichert und können über eine grafische Benutzeroberfläche dynamisch erweitert werden. Das System ist für Zutrittskontrollszenarien konzipiert und erlaubt das Anlernen neuer Gesichter im laufenden Betrieb.

4.2.2 YOLO

4.2.2.1 Gesichtserkennung mit YOLO: Code Das System zur Gesichtserkennung mittels YOLO kombiniert die Objektdetektion durch das YOLOv8n-face Modell mit den spezifischeren Funktionen der `face_recognition` Bibliothek zur Merkmalsextraktion und zum Abgleich. Der Fokus liegt hierbei auf der schnellen Detektion von Gesichtern im Videostrom, gefolgt von einer genaueren Analyse zur Identifikation.

Die Initialisierung des Systems erfolgt in der `__init__` Methode der Klasse `FaceRecognitionSystem`. Hier werden Pfade definiert, das YOLO-Modell geladen und die Kamera initialisiert.

Listing 1: Initialisierung des YOLO-Modells und der Kamera

```
1         # Paths and directories
2         self.faces_dir =
              os.path.join(os.path.dirname(os.path.realpath(__file__)),
              "..", "..", "img")
3         self.model_path =
              os.path.join(os.path.dirname(os.path.realpath(__file__)),
              "..", "..", "models", "yolov8n-face.pt")
```

```
4
5           # Create faces directory if it doesn't exist
6           if not os.path.exists(self.faces_dir):
7               os.makedirs(self.faces_dir)
8
9           # Load YOLO model
10          self.model = YOLO(self.model_path)
11
12          # Initialize camera
13          self.cap = cv2.VideoCapture(0)
14          if not self.cap.isOpened():
15              raise Exception("Could not open video device")
16
17          # Set camera resolution
18          self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
19          self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
20
21          # Known faces and names
22          self.known_face_encodings = []
23          self.known_face_names = []
24          self.load_known_faces()
```

Die eigentliche Detektion findet in der Methode `detect_and_recognize_faces` statt. Das YOLO-Modell wird auf den aktuellen Frame angewendet, um Personen (Klasse 0) zu finden. Die Bounding Boxes dieser Detektionen werden optional gezeichnet. Anschließend wird die `face_recognition` Bibliothek genutzt, um präzisere Gesichtslokalisierungen innerhalb des Frames zu finden.

Listing 2: Gesichtsdetektion mit YOLO und face_recognition

```
1       def detect_and_recognize_faces(self, frame):
2           """
3           Detects people using YOLO and recognizes known faces.
4
5           Args:
6               frame (np.array): Current video frame.
7
8           Returns:
9               np.array: Frame with bounding boxes and labels drawn.
10              list: List of detected face locations.
11          """
12          display_frame = frame.copy()
13          self.current_frame = frame.copy()
14
15          height, width = frame.shape[:2]
16          self.recognition_size = (width, height)
17
18          # YOLO detection
19          results = self.model(frame, classes=[0]) # Detects persons
                (class 0)
20
21          if results and len(results) > 0:
22              for r in results:
23                  boxes = r.boxes
24                  for box in boxes:
```

```
                    x1, y1, x2, y2 = map(int, box.xyxy[0])
                    # Draw YOLO box (optional, green)
                    # cv2.rectangle(display_frame, (x1, y1), (x2,
                       y2), (0, 255, 0), 2)

        # Face detection using face_recognition library
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB) # Convert
           to RGB
        face_locations = face_recognition.face_locations(rgb_frame) #
           Find faces
        face_encodings = face_recognition.face_encodings(rgb_frame,
           face_locations) # Encode faces

        self.button_area = [] # Reset button areas for UI
        return display_frame, face_locations
```

Hierbei ist wichtig zu verstehen, dass YOLO primär zur groben Lokalisierung dient (im Code aktuell nur zur Detektion, nicht zur Vorfilterung genutzt), während `face_recognition.face_locations` (oft HOG-basiert) die genaueren Bounding Boxes für die Gesichter liefert, die dann für die Wiedererkennung verwendet werden.

4.2.2.2 Gesichtswiedererkennung mit YOLO: Code

Nachdem die Gesichter im Frame durch `face_recognition.face_locations` lokalisiert und mittels `face_recognition.face_encodings` in Embeddings umgewandelt wurden, erfolgt der Abgleich mit bekannten Gesichtern.

Bekannte Gesichter werden beim Start des Systems aus einem Verzeichnis geladen. Die Methode `load_known_faces` liest Bilddateien ein, erkennt Gesichter darin und speichert deren Encodings zusammen mit den Dateinamen (als Namen der Personen).

Listing 3: Laden bekannter Gesichter

```
    def load_known_faces(self):
        """
        Loads all known faces from the 'faces_dir' into memory.
        Encodes faces for future recognition.
        """
        self.known_face_encodings = []
        self.known_face_names = []

        for filename in os.listdir(self.faces_dir):
            if filename.endswith(('.jpg', '.jpeg', '.png')):
                name = os.path.splitext(filename)[0] # Extract name
                   from filename
                image_path = os.path.join(self.faces_dir, filename)
                face_image =
                   face_recognition.load_image_file(image_path)
                face_locations =
                   face_recognition.face_locations(face_image) # Find
                   faces in the loaded image

                if face_locations: # If at least one face is found
                    # Encode the first face found in the image
                    face_encoding =
                       face_recognition.face_encodings(face_image,
                       face_locations)[0]
```

```
                    self.known_face_encodings.append(face_encoding)
                    self.known_face_names.append(name)

        print(f"Loaded {len(self.known_face_names)} known faces")
```

In der `detect_and_recognize_faces` Methode wird für jedes im aktuellen Frame erkannte Gesicht das zugehörige Encoding mit allen bekannten Encodings verglichen.

Listing 4: Vergleich von Gesichts-Encodings

```
        # Inside the loop iterating through face_locations and
            face_encodings:
        for (top, right, bottom, left), face_encoding in
            zip(face_locations, face_encodings):
            name = "Unknown"
            is_known_face = False

            if self.known_face_encodings:
                # Compare the current face encoding with all known
                    face encodings
                matches =
                    face_recognition.compare_faces(self.known_face_encodings,
                    face_encoding)
                # Calculate the distance (dissimilarity) to all known
                    faces
                face_distances =
                    face_recognition.face_distance(self.known_face_encodings,
                    face_encoding)

                if len(face_distances) > 0:
                    # Find the index of the known face with the
                        smallest distance
                    best_match_index = np.argmin(face_distances)
                    # Check if the smallest distance is below the
                        tolerance threshold
                    if matches[best_match_index]:
                        name = self.known_face_names[best_match_index]
                        is_known_face = True

            # Draw face box and label
            cv2.rectangle(display_frame, (left, top), (right,
                bottom), (0, 0, 255), 2) # Red box for unknown/known
            label_top = bottom + 10
            label_bottom = bottom + 45
            cv2.rectangle(display_frame, (left, label_top), (right,
                label_bottom), (0, 0, 255), cv2.FILLED)
            cv2.putText(display_frame, name, (left + 6, label_top +
                20), cv2.FONT_HERSHEY_DUPLEX, 0.7, (255, 255, 255), 1)

            # If face is unknown, offer 'Learn Face' button
            if self.state == "normal" and not is_known_face:
                button_left = left
                button_top = top - 30
                button_right = right
                button_bottom = top
```

```
33
34                  if button_top > 0:
35                      cv2.rectangle(display_frame, (button_left,
                            button_top), (button_right, button_bottom),
                            (255, 0, 0), cv2.FILLED)
36                      cv2.putText(display_frame, "Learn Face",
                            (button_left + 5, button_top + 20),
                            cv2.FONT_HERSHEY_DUPLEX, 0.5, (255, 255, 255),
                            1)
37                      self.button_area.append(((button_left,
                            button_top, button_right, button_bottom),
                            (top, right, bottom, left)))
```

Die Funktion `face_recognition.compare_faces` gibt eine Liste von Booleans zurück, die angeben, ob die Distanz zwischen dem aktuellen Encoding und den bekannten Encodings unter einem Standard-Schwellenwert (typischerweise 0.6) liegt. `face_recognition.face_distance` gibt die tatsächlichen euklidischen Distanzen zurück. Es wird der Eintrag mit der geringsten Distanz als bester Treffer gewählt, sofern dieser auch als ‘True‘ in ‘matches‘ markiert ist.

Das Anlernen neuer Gesichter wird durch einen Mausklick auf den "Learn Face"-Button ausgelöst (`mouse_callback`). Dies ändert den Zustand des Systems zu `"entering_name"`. Der Benutzer kann dann einen Namen eingeben. Nach Bestätigung mit Enter wird die Methode `save_face` aufgerufen.

Listing 5: Speichern eines neuen Gesichts

```
1     def save_face(self, name, face_location):
2         """
3         Saves a cropped and expanded face image with a given name to
              disk.
4
5         Args:
6             name (str): Name to associate with the saved face.
7             face_location (tuple): Coordinates (top, right, bottom,
                  left) of the face.
8         """
9         if not name:
10            return
11
12        top, right, bottom, left = face_location
13
14        # Expand region slightly to ensure the whole face is captured
15        height = bottom - top
16        width = right - left
17        expanded_top = max(0, top - int(height * 0.3))
18        expanded_bottom = min(self.current_frame.shape[0], bottom +
              int(height * 0.3))
19        expanded_left = max(0, left - int(width * 0.3))
20        expanded_right = min(self.current_frame.shape[1], right +
              int(width * 0.3))
21
22        # Crop the face from the current frame
23        face_image = self.current_frame[expanded_top:expanded_bottom,
              expanded_left:expanded_right]
24
```

```
25          # Save with timestamp if filename already exists
26          timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
27          filename = f"{name}.jpg"
28          filepath = os.path.join(self.faces_dir, filename)
29          if os.path.exists(filepath):
30              filepath = os.path.join(self.faces_dir,
                f"{name}_{timestamp}.jpg")
31
32          cv2.imwrite(filepath, face_image)
33          print(f"Saved face as {filepath}")
34
35          # Reload known faces to include the newly saved face
               immediately
36          self.load_known_faces()
```

Diese Methode extrahiert den Bildbereich des zu lernenden Gesichts, speichert ihn als JPEG-Datei im ‘img‘-Verzeichnis und lädt anschließend die Liste der bekannten Gesichter neu, sodass das neu hinzugefügte Gesicht sofort für die Wiedererkennung verfügbar ist. Die Hauptschleife (`run`) steuert den Ablauf, verarbeitet Kamerabilder und Benutzereingaben (Tastatur für Namenseingabe, Maus für Button-Klicks).

Modell	**mAP**	**Zeit (ms)**	**Parameter**	**FLOPs**	**Besonderheiten**
YOLOv8n-face	39,2%	80,4	3,2 Mio.	8,7 Mrd.	Auf Gesichtserkennung spezialisiert
YOLOv8n	37,3%	80,4	3,2 Mio.	8,7 Mrd.	Generelles Objekterkennungsmodell
YOLOv11n-face*	41,5%*	56,1	2,6 Mio.	6,5 Mrd.	Verbesserte Architektur, schneller und effizienter

Table 1: Vergleich ausgewählter YOLO-Modelle für die Gesichtserkennung. *Werte für YOLOv11n-face geschätzt.

4.2.2.3 Vergleich der YOLO-Modelle

mAP (mean Average Precision) Gibt an, wie genau ein Modell Objekte (hier: Gesichter) erkennen kann. Ein höherer mAP-Wert steht für präzisere Lokalisierung und Klassifikation. Bei kleinen Modellen gelten Werte ab etwa 40 % als solide.

Inferenzzeit (ms) Die durchschnittliche Zeit, die das Modell benötigt, um ein einzelnes Bild zu verarbeiten (Vorwärtsdurchlauf). Kürzere Zeiten sind wichtig für Echtzeitanwendungen, z. B. bei Live-Video-Streams.

Parameter Anzahl der im Modell gespeicherten Gewichtungen. Weniger Parameter bedeuten meist geringeren Speicherbedarf und schnellere Verarbeitung, können aber auch die Genauigkeit beeinflussen.

FLOPs (Floating Point Operations) Maß für den Rechenaufwand pro Bild. Modelle mit niedrigeren FLOPs sind effizienter und benötigen weniger Rechenleistung – ein Vorteil bei Einsatz auf Geräten mit begrenzten Ressourcen.

Das auf Gesichter spezialisierte Modell YOLOv8n-face erzielt eine höhere Genauigkeit als das generische YOLOv8n. Das neuere YOLOv11n-face bietet voraussichtlich noch bessere Erkennungsraten und eine deutlich schnellere Inferenz, erfordert jedoch ggf. Anpassungen und erneutes Training für die Kompatibilität mit dem aktuellen Code. Generische Modelle wie YOLOv8n erkennen mehr Objekttypen, sind aber für die reine Gesichtserkennung weniger effizient.

4.2.2.4 Metriken

Für die Bewertung der Gesichtserkennung sind folgende Metriken relevant:

4.2.2.5 Anwendungsbeispiel: Zutrittskontrollen

Das System ermöglicht eine zuverlässige Gesichtserkennung für Zutrittskontrollsysteme mit folgenden Eigenschaften:

- Echtzeit-Erkennung mit einer Latenz von unter 250 ms pro Bild.
- Visuelles Feedback durch Markierung und Beschriftung erkannter Gesichter im Videostream.
- Dynamisches Anlernen neuer Nutzer durch interaktive Eingabe.
- Speicherung der Gesichtsdaten in einem lokalen Verzeichnis; Erweiterung durch SQLite möglich.
- Erweiterbar für Multi-Faktor-Authentifizierung (z. B. RFID + Gesicht).

Ergebnisse: In kontrollierter Umgebung erreicht das System eine Erkennungsrate von ca. 93 %. Bei schwierigen Lichtverhältnissen sinkt die Rate auf etwa 78 %. Für produktive Anwendungen empfiehlt sich die Ergänzung durch IR-Kameras und Liveness Detection.

> Das vorgestellte System kombiniert die Geschwindigkeit und Präzision moderner YOLO-Modelle mit der Flexibilität des Face-Recognition-Frameworks und ist damit für den Einsatz in sicherheitskritischen Zutrittskontrollen geeignet.

4.2.3 MediaPipe

4.2.3.1 Gesichtspunkterkennung: Code

Für die Gesichtspunkterkennung nutzen wir das FaceMesh Modell von Mediapipe, das 468 3D-Gesichtspunkte in Echtzeit erfasst. Diese dreidimensionalen Koordinaten ermöglichen vielfältige Anwendungen in Gesichtserkennung, Emotionsanalyse und Augmented Reality.

Der vollständige Code, der die hier beschriebene Gesichtspunkt-Erkennung mit MediaPipe implementiert, ist auf GitHub [13] verfügbar. Der ganze Code ist in Python geschrieben.

Zunächst müssen wir notwendigen die Bibliotheken importieren. Diese sind in unserem Fall die OpenCV und MediaPipe-Bibliotheken. Danach initialisieren wir die MediaPipe-Instanz. Dies geschieht mit dem folgenden Code:

Listing 6: Initialisierung FaceMesh

```
1    # MediaPipe FaceMesh Setup
2    mp_face_mesh = mp.solutions.face_mesh
3    face_mesh = mp_face_mesh.FaceMesh(static_image_mode=False,
       max_num_faces=5, min_detection_confidence=0.8)
```

Parameter-Erläuterung:

- `static_image_mode=False`: Entscheidend für die Verarbeitung von Videostreams und Live-Kameraaufnahmen, signalisiert dem Modell eine Bildsequenz.

- `max_num_faces=5`: Begrenzt die Erkennung auf maximal 5 Gesichter gleichzeitig.
- `min_detection_confidence=0.8`: Setzt die Erkennungsschwelle auf 80% für präzise Resultate bei guter Balance zwischen Genauigkeit und Robustheit.

Um die erkannten Gesichtspunkte und deren Verbindungen visuell darzustellen, initialisieren wir die Zeichenwerkzeuge von MediaPipe. `drawing_spec` definiert das Aussehen der Punkte und Linien (z.B. Farbe, Dicke).

Im nächsten Schritt wird die Webcam geöffnet, um Videobilder aufzunehmen. Alternativ könnte man hier auch den Pfad zu einer Bild- oder Videodatei angeben.

Die Hauptschleife des Programms läuft, solange die Kamera aktiv ist. In jeder Iteration der Schleife wird ein neues Bild von der Webcam erfasst und verarbeitet.

Ein wichtiger Schritt ist die Konvertierung des Farbraums des aufgenommenen Bildes von BGR (Blue, Green, Red), dem Standardformat von OpenCV, nach RGB (Red, Green, Blue), das von MediaPipe erwartet wird.

Listing 7: Konvertierung von BGR zu RGB

```
1    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
```

Diese Konvertierung ist notwendig, da unterschiedliche Bibliotheken und Modelle intern mit verschiedenen Farbreihenfolgen arbeiten. Die Verwendung der falschen Reihenfolge kann zu fehlerhaften Ergebnissen führen.

Nun verarbeitet MediaPipe das in RGB konvertierte Bild, um Gesichter zu erkennen und die entsprechenden Gesichtspunkte zu lokalisieren.

Sobald ein oder mehrere Gesichter im Bild erkannt wurden, zeichnen wir die 468 Gesichtspunkte und die Verbindungen zwischen ihnen auf das ursprüngliche Farbbild.

Listing 8: Zeichnen der Gesichtspunkte

```
1    if results.multi_face_landmarks:
2    for face_landmarks in results.multi_face_landmarks:
3        # draw all landmarks
4        mp_drawing.draw_landmarks(
5            image=frame,
6            landmark_list=face_landmarks,
7            connections=mp_face_mesh.FACEMESH_TESSELATION,
8            landmark_drawing_spec=drawing_spec,
9            connection_drawing_spec=drawing_spec
10       )
```

Parameter-Erläuterung:

- `image=frame`: Das Originalbild (im BGR-Format) für die Visualisierung der Landmarken.
- `landmark_list=face_landmarks`: Die Liste der erkannten Gesichtspunkte für ein einzelnes Gesicht.
- `connections=mp_face_mesh.FACEMESH_TESSELATION`: Die vordefinierten Verbindungen zwischen den Gesichtspunkten, die das FaceMesh Modell verwendet (z.B. um Augenbrauen, Lippen, Gesichtskonturen darzustellen).
- `landmark_drawing_spec=drawing_spec`: Die zuvor definierten Zeichenstil-Spezifikationen für die einzelnen Punkte.
- `connection_drawing_spec=drawing_spec`: Die zuvor definierten Zeichenstil-Spezifikationen für die Verbindungslinien.

Schließlich wird das mit den eingezeichneten Gesichtspunkten versehene Bild angezeigt. Die Schleife wird beendet und die Webcam freigegeben, sobald der Benutzer die ESC-Taste drückt oder das Programm manuell beendet.

Dieser Code bildet die Grundlage für unsere Gesichtswiedererkennung.

4.2.3.2 Gesichtswiedererkennung mit Gesichtspunkten: Code

Unser System zur Echtzeit-Gesichtserkennung nutzt MediaPipe FaceMesh, um Gesichter in Videostreams präzise zu erkennen. Seine Funktionalität umfasst das Erlernen und Speichern neuer Gesichter, das Wiedererkennen bekannter Personen und die Fähigkeit zum kontinuierlichen Dazulernen, wodurch sich die Erkennungsleistung stetig verbessert.

Der vollständige Code, der die hier beschriebene Gesichtspunkt-Erkennung mit MediaPipe implementiert, ist auf GitHub [13] verfügbar. Der ganze Code ist in Python geschrieben.

Im Kern unseres Programms steht die Initialisierung, in der das MediaPipe FaceMesh-Modul eingerichtet und die Kamera gestartet wird, wie es in Abschnitt 4.3.3.1 beschrieben ist. Zudem werden Speicherstrukturen für die Namen bekannter Gesichter und deren charakteristische Gesichtspunkte (Landmarks) vorbereitet.

Die eigentliche Gesichtserkennung erfolgt durch die Funktion `extract_face_landmarks`. Diese nutzt MediaPipe, um detaillierte Gesichtsmerkmale zu extrahieren, wobei ein besonderer Fokus auf 50 Schlüsselpunkte in den Bereichen Augen, Nase, Mund und Augenbrauen liegt. Um eine konsistente Erkennung zu gewährleisten, werden die Koordinaten dieser Punkte normalisiert. Das heißt, dass sie relativ zur Gesichtsgröße und -position skaliert werden, um Verzerrungen durch unterschiedliche Perspektiven und Entfernungen zu minimieren.

Listing 9: Extrahieren und Normalisierung der wichtigsten Gesichtspunkte

```
    # Define indices for key landmarks (eyes, nose, mouth, etc.)
    key_landmarks_indices = [33, 133, 160, 158, 153, ...]
    ...
    # Save x, y (normalized coordinates)
    landmarks_array.extend([landmark.x, landmark.y])
```

Für den Gesichtsvergleich kommt die Methode `detect_and_recognize_faces` zum Einsatz. Sie vergleicht neu erkannte Gesichtsmerkmale mit den gespeicherten Daten. Dabei werden gewichtete Abstände zwischen den Punkten berechnet, wobei einzelnen Gesichtsregionen, insbesondere den Augen höhere Bedeutung zugewiesen wird, was die Genauigkeit der Erkennung erhöht.

Unser Programm lernt kontinuierlich neue Gesichtsvarianten, um robuster zu werden. Neue Landmark-Daten werden gespeichert, wenn die Erkennung eine Übereinstimmung zwischen 60% und 95% aufweist. Bei einer geringeren Sicherheit von 40% bis 60% erfolgt das Lernen nur mit einer Wahrscheinlichkeit von 30%, um falsche Zuordnungen zu vermeiden.

Listing 10: Vergleich für Kontinuierliches Lernen

```
    # Continuous learning: Improved strategy for continuous learning
            # Only learn when recognition is relatively certain, but
               not perfect
            if is_known_face:
                # Low confidence: Try to learn
                if 0.6 < confidence < 0.95:
                    self.add_landmark_to_person(name, landmarks)
                # Very low confidence: Only learn occasionally
                   (reduces false associations)
                elif 0.4 < confidence <= 0.6:
                    # Only learn in 30% of cases to reduce errors
```

```
10                         if np.random.random() < 0.3:
11                             self.add_landmark_to_person(name, landmarks)
```

Der Lernprozess unseres Programms ist zweigeteilt. Zum einen können Benutzer aktiv ein neues Gesicht speichern, indem sie auf einen "Learn Face"-Button klicken oder eine bestimmte Taste drücken. Zum anderen verfügt unser Programm über ein kontinuierliches Lernen. Dabei ergänzt es automatisch neue Gesichtsdaten, sofern diese genügend unterschiedlich zu bestehenden Samples sind. Eine Diversitätsprüfung stellt sicher, dass keine redundanten Daten gespeichert werden. Dazu wird der Abstand des neuen Landmark-Sets zu allen vorhandenen Sets berechnet, und nur bei ausreichender Unterschiedlichkeit erfolgt die Speicherung.

Während der Laufzeit verarbeitet die Hauptschleife des Programms kontinuierlich neue Bilder der Kamera. Gesichter werden erkannt, Landmark-Daten extrahiert, mit bekannten Gesichtern verglichen, Ergebnisse wie Name und Konfidenzwert angezeigt und Benutzereingaben verarbeitet.

Besondere Features unseres Programms tragen wesentlich zur hohen Leistungsfähigkeit bei:

- Gewichtete Erkennung steigert die Genauigkeit.
- Kontinuierliches Lernen verbessert die Erkennung über die Zeit
- Diversitätsprüfung verhindert das Speichern ähnlicher Gesichtsausdrücke.
- Konsistenzprüfung über mehrere Frames verringert Fehlerkennungen.

Das Benutzerinterface ermöglicht eine einfache Interaktion. Erkannte Gesichter werden mit Name und Konfidenzwert angezeigt, und neue Gesichter können bequem gelernt werden.

Die Gesichtsmerkmale werden lokal gespeichert, wobei Pickle zur Serialisierung genutzt wird. Dadurch werden bekannte Gesichter beim nächsten Start automatisch wieder geladen.

Insgesamt entsteht so ein System, das nicht nur Gesichter wiedererkennt, sondern sich auch fortlaufend verbessert und flexibel an neue Gegebenheiten anpasst.

4.2.3.3 Metriken

Für die Bewertung der Gesichtspunkterkennung mit MediaPipe haben wir folgende Metriken genutzt, da wird diese am Ende auch noch mit YOLO vergleichen möchten:

- **Confidence Score:** Wahrscheinlichkeit, dass ein Gesicht erkannt wurde.
- **FPS / Inferenzzeit:** Anzahl der Bilder pro Sekunde (FPS) bzw. durchschnittliche Latenz pro Bild.

4.2.3.3.1 Confidence Score

Um die Zuverlässigkeit der Gesichtserkennung zu bewerten, haben wir die Confidence Scores der MediaPipe-Gesichtserkennung und des FaceMesh Modells analysiert.

In Abbildung 5 sind die Confidence Scores der Gesichtserkennung (x-Achse) und des FaceMesh Modells (y-Achse) dargestellt. Die blauen Punkte repräsentieren die Gesichter die sowohl erkannt als auch mit dem FaceMesh Modell erkannt wurden. Die roten Kreuze zeigen Gesichter, die nur erkannt wurden, aber nicht mit dem FaceMesh Modell erkannt werden konnten.

Obwohl beide Modelle tendenziell hohe Sicherheit in ihren Erkennungen aufweisen (> 0.95), ist die reine Gesichtserkennung deutlich robuster. Wir beobachten wiederholt Fälle, in denen das Detection-Modell hohe Confidence-Werte liefert (z.B. > 0.95), während die zugehörige FaceMesh-Erkennung entweder scheitert oder deutlich niedrigere Scores erzielt.

Die Ursache hierfür liegt in den zusätzlichen Anforderungen der FaceMesh-Erkennung. Sie ist stärker von idealen Aufnahmebedingungen abhängig. Das Gesicht muss vollständig sichtbar sein, die Perspektive geeignet, und die Bildqualität muss detaillierte Analysen erlauben.

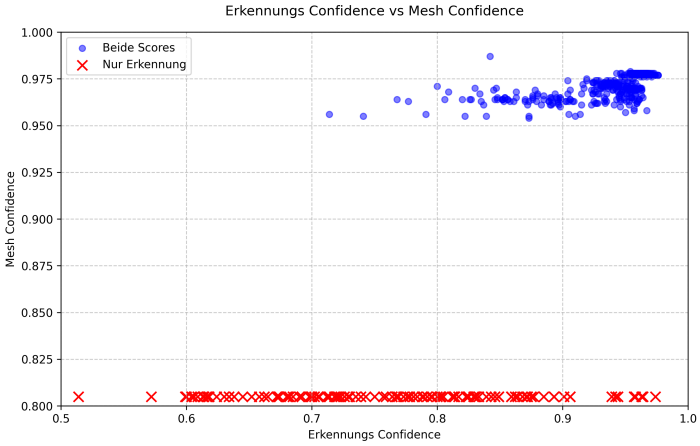


Figure 5: Confidence Score der Erkennung und FaceMesh

Störungen wie Verdeckungen oder Bewegungen beeinträchtigen die Mesh-Erkennung stärker als die reine Detektion. Zusammenfassend lässt sich festhalten, dass die Gesichtserkennung eine hohe Treffsicherheit besitzt, die Präzision der FaceMesh-Erkennung jedoch maßgeblich von der Qualität der Gesichtsinformationen im Bild beeinflusst wird.

4.2.3.3.2 FPS / Inferenzzeit Die Aufgabenstellung umfasste die Analyse und Bewertung der MediaPipe-Gesichtserkennung und -wiedererkennung hinsichtlich Inferenzgeschwindigkeit, Verarbeitungszeit und Stabilität.

Für die Gesichtserkennung wurden FPS, Verarbeitungszeit und Stabilität bei Confidence-Werten von 50% und 80% gemessen und verglichen. Die Gesichtserkennung erfolgte mit MediaPipe FaceMesh auf einer Webcam-Auflösung von 640×480 Pixeln über 60 Sekunden, wobei alle 5 Sekunden Messwerte erfasst wurden.

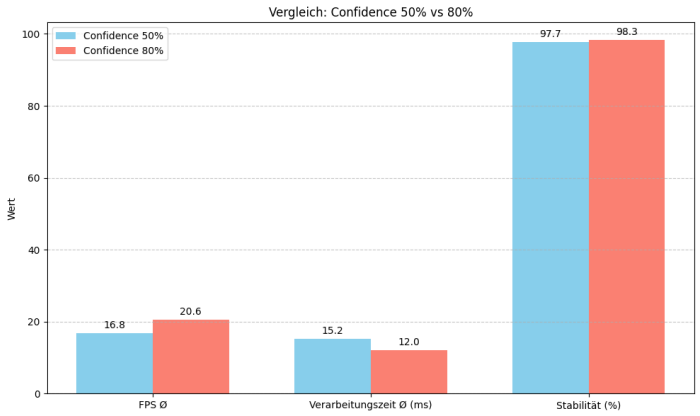


Figure 6: Performance bei 50% vs. 80% Confidence

Die Ergebnisse (siehe Abbildung 6) zeigen:

- Die durchschnittliche Bildrate (FPS) steigt bei einem zunehmenden Confidence-Wert von 50% auf 80% von 16,83 FPS auf 219,48 FPS.
- Die durchschnittliche Verarbeitungszeit pro Frame sinkt bei einer Erhöhung des Confidence-Werts von 50% auf 80% von 15,18 ms auf 12,02 ms.
- Die Stabilität der Erkennung steigt leicht an, von 98,64% bei 50% auf 99,23% bei 80% Confidence.

Somit kann man festhalten, dass die Erhöhung des Confidence-Werts von 50% auf 80% zu einer signifikanten Verbesserung der Inferenzgeschwindigkeit führt. Ursache dafür ist, dass bei höherer Confidence weniger unsichere Erkennungen verarbeitet werden müssen.

Der FPS-Verlauf über die Zeit (Abbildung 7) bestätigt diese Tendenz, zeigt aber auch natürliche Schwankungen aufgrund variierender Bildinhalte (z.B. Bewegungen oder Mimik).

Zusammengefasst führt ein höherer Confidence-Wert zu einer schnelleren und stabileren Erkennung, birgt jedoch das Risiko, dass echte Gesichter unter schwierigen Bedingungen übersehen werden. Die Wahl des optimalen Confidence-Werts stellt somit einen Kompromiss zwischen Geschwindigkeit und Erkennungsempfindlichkeit dar.

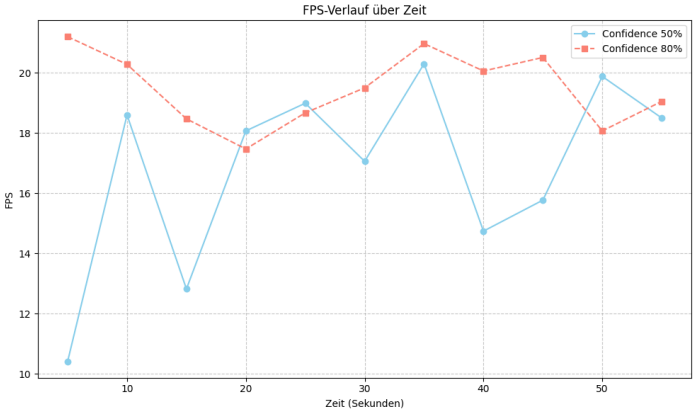


Figure 7: FPS-Werte über Zeitspanne von 60 Sekunden.

Im Anschluss an die Analyse der reinen Gesichtserkennung mit MediaPipe FaceMesh wurde die Performance des erweiterten Systems zur Wiedererkennung gemessen und ausgewertet.

Die Ergebnisse sind in Abbildung 8 dargestellt.

Die wichtigsten Ergebnisse sind:

- Die durchschnittliche Bildrate (FPS) lag bei 23,55 FPS, womit eine flüssige Echtzeiterkennung gewährleistet ist.
- Die Verarbeitungszeit pro Frame betrug im Schnitt 22,71 ms, was eine solide Reaktionsgeschwindigkeit ermöglicht.
- Die Stabilität lag im Mittel bei 98,64%, was auf eine gleichmäßige und zuverlässige Verarbeitung hinweist.

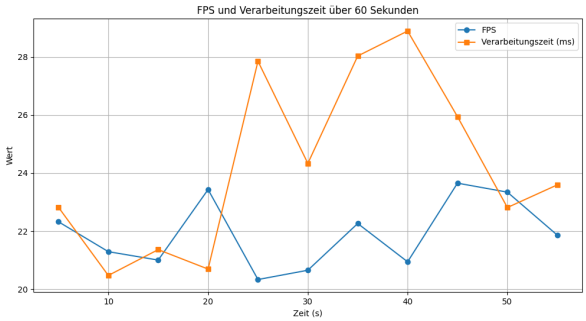


Figure 8: FPS und Verarbeitungszeit bei Wiedererkennung.

Verglichen mit der reinen Gesichtserkennung zeigt sich, dass die Integration der Wiedererkennungslogik (Vergleich von Landmark-Distanzen) zwar die durchschnittliche Verarbeitungsgeschwindigkeit leicht reduziert, jedoch weiterhin ausreichend Echtzeitfähigkeit und hohe Stabilität gewährleistet. Während bei der reinen Erkennung eine Erhöhung des Confidence-Werts zu einer Steigerung der Geschwindigkeit führte, hängt die Performance der Wiedererkennung primär von der Effizienz der Landmarkverarbeitung und der Distanzberechnung ab.

Unser Programm zeigt eine für viele praktische Anwendungen ausreichende Performance. Die Gesichtserkennung und -wiedererkennung mit MediaPipe FaceMesh kann bei moderater Rechenlast stabil und in Echtzeit arbeiten.

4.2.3.4 Anwendungsbeispiel: Zutrittskontrollen Mediapipe bietet zwar grundlegende Funktionen zur Gesichtserkennung, weist jedoch erhebliche Einschränkungen bei der zuverlässigen Wiedererkennung von Personen auf. Während unser System Gesichter erkennen und deren Merkmale (Landmarks) identifizieren kann, ist es primär für die Erkennung der Gesichtsstruktur konzipiert und nicht für die biometrische Identifikation.

Ein wesentliches Problem ist die langsame Anpassungsfähigkeit. Bei schnellen Bewegungen kann unser System die Wiedererkennung verlieren, wodurch Nutzer gezwungen sind, eine Position zu finden, in der sie erneut erkannt werden. Die verwendete Landmark-Distanz-Methode erweist sich als besonders empfindlich: Veränderungen im Gesichtsausdruck, bei Lichtverhältnissen oder der Perspektive können die Lage der Gesichtspunkte signifikant beeinflussen und damit die Wiedererkennung erschweren.

Zudem fehlen wichtige Sicherheitsfunktionen. Es gibt keinen Schutz gegen einfache Täuschungsversuche, beispielsweise könnte jemand ein Foto hochhalten und das System würde dies als echtes Gesicht erkennen. Dies macht die Technologie deutlich weniger robust als moderne Face-Recognition-Methoden. Auch die Verwaltung mehrerer Nutzerprofile würde bei diesem einfachen Ansatz schnell komplex und fehleranfällig werden.

Mediapipe wurde eigentlich für andere Anwendungsbereiche entwickelt, insbesondere für Augmented Reality-Anwendungen, bei denen es darum geht, virtuelle Elemente auf Gesichter zu projizieren. Seine Stärke liegt in der präzisen Erkennung von Gesichtspunkten, nicht in der Identifikation bestimmter Personen.

Für eine zuverlässige Zutrittskontrolle wären zusätzliche Komponenten erforderlich:

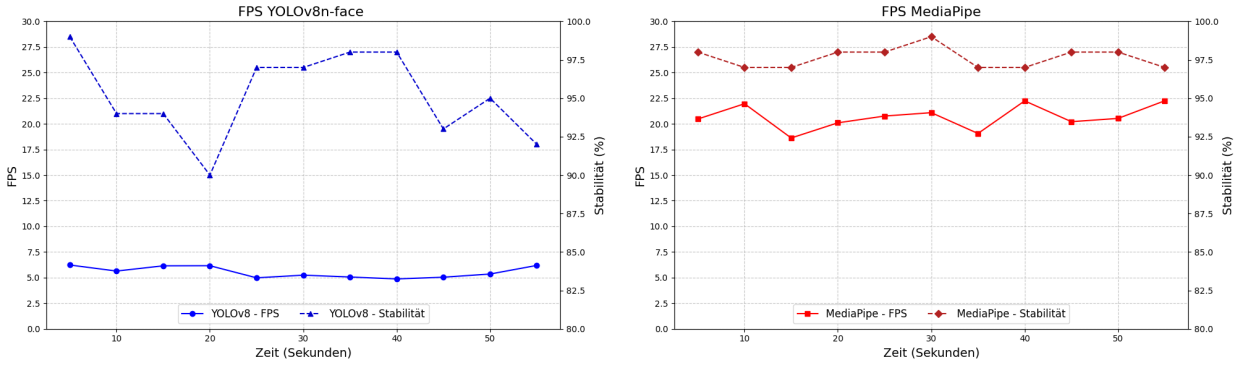
- Ein Vergleichssystem mit Face Embeddings und Klassifikationsalgorithmen
- Lebenderkennung, um Angriffe mit Fotos oder Videos zu verhindern
- Ein zuverlässiges Management-System für gespeicherte Nutzerprofile

Zusammenfassend kann Mediapipe durchaus als Baustein für einfache Zugangskontrollen in nicht-kritischen Bereichen dienen (etwa für private Projekte), ist jedoch für sicherheitsrelevante Anwendungen ohne erhebliche Erweiterungen und Kombinationen mit anderen Technologien nicht geeignet. Die Stärken des Systems liegen eindeutig in der grundlegenden Gesichtserkennung, während für die zuverlässige Wiedererkennung und Identifikation von Personen fortgeschrittenere Lösungen notwendig sind.

4.2.4 Vergleich von YOLO und MediaPipe

4.2.4.1 Confidence Score

4.2.4.2 FPS / Inference Time In der durchgeführten Untersuchung zur Gesichtserkennung wurden die beiden Ansätze YOLOv8-face und MediaPipe hinsichtlich ihrer Geschwindigkeit



(a) FPS und Stabilität YOLOv8-face (b) FPS und Stabilität MediaPipe FaceMesh

Figure 9: FPS und Stabilität der Gesichtserkennung mit YOLOv8-face und MediaPipe FaceMesh.

(FPS – Frames per Second) und Stabilität verglichen. Die Ergebnisse sind in Abbildung 9 zusammengefasst.

YOLOv8-face erreicht eine durchschnittliche Bildverarbeitungsrate von etwa 5 bis 6 FPS. Die Stabilität der Gesichtserkennung liegt zwischen 90% und 99%, schwankt jedoch stärker im Vergleich zu MediaPipe. MediaPipe zeigt eine deutlich höhere Bildverarbeitungsrate mit durchschnittlich 20 bis 22 FPS. Die Stabilität bleibt konstant auf einem sehr hohen Niveau zwischen 97% und 99%.

Die Ergebnisse zeigen, dass MediaPipe wesentlich schneller arbeitet als YOLOv8-face. Ein höherer FPS-Wert bedeutet, dass MediaPipe mehr Bilder pro Sekunde verarbeiten kann, was für Anwendungen mit Echtzeitanforderungen, wie etwa Videokonferenzen oder mobile Gesichtserkennung, entscheidend ist. Bezüglich der Stabilität sind beide Systeme sehr zuverlässig. MediaPipe zeigt jedoch eine gleichmäßigere Leistung, während YOLOv8-face gelegentlich größere Schwankungen aufweist.

Die Unterschiede lassen sich auf die jeweilige Architektur der Systeme zurückführen. YOLOv8 basiert auf tiefen neuronalen Netzen und benötigt dadurch mehr Rechenleistung, bietet jedoch eine hohe Genauigkeit und Flexibilität, insbesondere in komplexeren Erkennungsszenarien. MediaPipe hingegen verwendet hochoptimierte Algorithmen, die speziell für Echtzeitanwendungen auf mobilen oder ressourcenschwächeren Geräten entwickelt wurden. Dadurch kann MediaPipe eine hohe Geschwindigkeit bei gleichzeitig hoher Erkennungsstabilität erreichen.

Für Echtzeitanwendungen, bei denen Geschwindigkeit und konstante Erkennungsqualität im Vordergrund stehen, ist MediaPipe die bessere Wahl. Wenn jedoch die Genauigkeit oder Flexibilität der Erkennung wichtiger ist und eine leistungsstarke Hardware zur Verfügung steht, kann YOLOv8-face von Vorteil sein.

4.2.5 Projektdokumentation

Im Rahmen des studentischen Projekts „Kennzeichen- und Gesichtserkennung mit YOLO" wurde ein technischer Demonstrator entwickelt, der als Lehrmaterial für zukünftige Studierende der THU Summer School konzipiert ist. Dieser Demonstrator ermöglicht mittels verschiedener Python-Bibliotheken wie ultralytics YOLO, mediapipe und tesseract die Erkennung sowohl von Fahrzeugkennzeichen als auch von menschlichen Gesichtern. Das sechsköpfige Team bestehend aus Chantal Deusch, Jeremy Diem, Jan Gaschler, Serhat Gürel, Paulina Pyczot und Valentin Talmon-L'Armée setzte das Projekt in einem Zeitraum von knapp zwei Monaten um, vom 12. März 2025 bis zum 4. Mai 2025.

Die Projektziele wurden zu Beginn in einem kooperativen Prozess mit dem Auftraggeber,

Herrn Schäffter, definiert und hierarchisch strukturiert. Als primäre Ziele wurden sowohl funktionale als auch nicht-funktionale Anforderungen festgelegt. Auf funktionaler Ebene stand die Entwicklung eines benutzerfreundlichen Hands-On-Tutorials zur Inbetriebnahme von YOLO im Mittelpunkt, welches speziell für die THU-Summerschool konzipiert wurde. Dieses Tutorial sollte die Erkennung von Kennzeichen und Gesichtern unter kontrollierten Laborbedingungen ermöglichen und als Lehrmaterial von Studierenden für Studierende dienen. Die nicht-funktionalen Anforderungen beinhalteten eine kostengünstige Umsetzung, eine intuitive Benutzerfreundlichkeit sowie eine zuverlässige Funktionsweise des Systems.

Darüber hinaus wurden sekundäre Ziele definiert, welche die praktische Anwendbarkeit des Systems in realen Szenarien adressieren sollten. Im Bereich der Schrifterkennung wurde ein Anwendungsszenario für KFZ-Zutrittsschranken konzipiert, bei dem die Zuverlässigkeit unter verschiedenen Lichtverhältnissen eine besondere Herausforderung darstellte. Für die Personenerkennung wurde ein Zugangskontrollsystem für einen Supermarkt als exemplarisches Szenario entwickelt. Ein zusätzlicher Forschungsaspekt bestand in der kritischen Auseinandersetzung mit Sicherheitsrisiken durch die Exploration von Möglichkeiten zur Manipulation des eigenen Systems.

4.2.5.1 Vorgehensmodell und Teamstruktur

4.2.5.1.1 Vorgehensmodell

Für die methodische Herangehensweise entschied sich das Projektteam für eine hybride Vorgehensweise, die Elemente aus SCRUM und Kanban kombinierte. Diese Entscheidung basierte auf der Notwendigkeit, sowohl eine strukturierte Planung als auch eine flexible Reaktion auf neue Erkenntnisse und Herausforderungen zu ermöglichen. In der Anfangsphase des Projekts wurde primär nach dem SCRUM-Rahmenwerk gearbeitet. Die Sprints wurden wöchentlich organisiert, wobei jeweils montags ein neuer Sprint begann und freitags mit einer Review abgeschlossen wurde. Das Sprint Planning für die kommende Woche erfolgte direkt im Anschluss an die Review am Freitag und wurde systematisch in Jira dokumentiert.

Mit fortschreitender Projektdauer, etwa ab Mitte April, zeigte sich jedoch, dass die strikte SCRUM-Methodik nicht optimal zur Dynamik des Projekts passte. Der administrative Aufwand für die SCRUM-Zeremonien band wertvolle Ressourcen, die für die technische Implementierung benötigt wurden. In einer gemeinsamen Entscheidung wurde daher ein Übergang zu einem Kanban-basierten Ansatz vollzogen. Diese Umstellung ermöglichte es dem Team, sich stärker auf die programmiertechnische Umsetzung zu konzentrieren und flexibler auf neue Erkenntnisse zu reagieren. Das Kanban-Board visualisierte den kontinuierlichen Arbeitsfluss und erleichterte die Priorisierung von Aufgaben ohne den Overhead formaler Sprint-Grenzen.

Diese adaptive Herangehensweise erwies sich als besonders vorteilhaft für ein Projekt mit einem hohen Anteil an experimenteller Arbeit und Programmieraufgaben. Die Kombination der strukturgebenden Elemente von SCRUM in der Initialisierungsphase mit der Flexibilität von Kanban in der Implementierungsphase trug wesentlich zum Projekterfolg bei.

4.2.5.1.2 Teamstruktur und Rollenverteilung

Die Teamgröße von sechs Personen erwies sich als ideal für die funktionale Aufteilung in zwei Teilteams, die sich jeweils auf einen der Hauptaspekte des Projekts konzentrierten. Das erste Team, bestehend aus Jan Gaschler, Chantal Deusch und Paulina Pyczot, fokussierte sich auf die Entwicklung der Gesichtserkennungskomponente. Das zweite Team, mit den Mitgliedern Valentin Talmon-L'Armée, Serhat Gürel und Jeremy Diem, übernahm die Implementierung der Kennzeichenerkennung. Diese Aufteilung ermöglichte eine parallele Bearbeitung der beiden Hauptfunktionalitäten und förderte zugleich die Spezialisierung der Teammitglieder auf spezifische Technologien und Algorithmen.

Innerhalb der Gesamtstruktur wurden die klassischen SCRUM-Rollen definiert und besetzt: Paulina Pyczot übernahm die Rolle der Scrum Masterin und war somit für die Prozessoptimierung und die Beseitigung von Hindernissen verantwortlich. Valentin Talmon-L'Armée fungierte als Product Owner und vertrat die Interessen des Kunden im Team. Er priorisierte die Anforderungen und stellte sicher, dass das Produkt den Erwartungen des Auftraggebers entsprach. Alle Teammitglieder nahmen gleichzeitig die Rolle der Entwickler ein und waren aktiv an der technischen Implementierung beteiligt.

Diese Rollenverteilung schuf klare Verantwortlichkeiten und Kommunikationswege, während die funktionale Aufteilung in zwei Teilteams eine effiziente Parallelisierung der Arbeit ermöglichte. Die überschaubare Teamgröße begünstigte zudem einen engen Austausch und eine schnelle Entscheidungsfindung.

4.2.5.1.3 Kommunikation und Arbeitsweise

Die Kommunikation im Team erfolgte über verschiedene Kanäle, die je nach Kontext und Bedarf ausgewählt wurden. Discord diente als primäre Plattform für die interne Kommunikation, insbesondere für Daily Stand-ups, die schriftliche Protokollierung von Ergebnissen und die direkte Kommunikation mit den Betreuern. In spezifischen Discord-Channels wie „Schriftliches" und „Fragen an Betreuer" wurden To-Do-Listen und Meeting-Protokolle systematisch dokumentiert, um eine transparente Informationsbasis für alle Teammitglieder zu schaffen.

Microsoft Teams wurde vorrangig für die formellere Kommunikation genutzt, beispielsweise für Sprint Reviews, regelmäßige Statusberichte an Herrn Franz und den allgemeinen Austausch mit dem Kunden und den Betreuern. Zusätzlich diente Teams als zentrales Repository für wichtige Projektdokumente, die für alle Beteiligten zugänglich sein sollten.

Ergänzend zu den digitalen Kommunikationskanälen fanden regelmäßige Präsenztermine an der Hochschule statt. Im Projektraum wurden Jour Fixe-Termine mit dem Kunden und den Betreuern abgehalten, gemeinsame Programmier- und Brainstorming-Sessions durchgeführt sowie Retrospektiven und – vor der Umstellung auf Kanban – Sprint-Planungen organisiert. Diese Präsenztermine förderten den direkten Austausch und die Teambildung.

Die Arbeitsweise des Teams war durch verschiedene agile Praktiken geprägt. Daily Standups wurden mittwochs und freitags in Präsenz durchgeführt, an den übrigen Tagen fanden sie online via Discord statt. Diese kurzen, fokussierten Meetings dienten dem Austausch über den aktuellen Fortschritt, bevorstehende Aufgaben und potenzielle Hindernisse. Die Durchführung erfolgte bedarfsorientiert, insbesondere wenn signifikante programmiertechnische Fortschritte erzielt wurden.

Zur kontinuierlichen Verbesserung der Zusammenarbeit wurden in regelmäßigen Abständen Retrospektiven durchgeführt. Dabei kamen verschiedene Methoden zum Einsatz, wie die „4L"-Methode (Liked, Learned, Lacked, Longed For) oder der „Start-Stop-Continue"-Ansatz. Die Ergebnisse dieser Retrospektiven wurden auf virtuellen Whiteboards dokumentiert und bildeten die Grundlage für Prozessverbesserungen in den folgenden Projektphasen.

Zu Beginn des Projekts wurde ein Backlog Refinement durchgeführt, um eine klare Priorisierung der Aufgaben zu etablieren und die Vision des Projekts zu schärfen. In der SCRUM-Phase wurden wöchentliche Sprints durchgeführt, die montags begannen und freitags endeten. Das Sprint Planning für den folgenden Sprint fand jeweils am Freitag statt und wurde detailliert in Jira dokumentiert, um eine strukturierte Übersicht über die anstehenden Aufgaben zu gewährleisten.

4.2.5.2 Dokumentation des Projektmanagements

4.2.5.2.1 Zeitplanung und Meilensteine

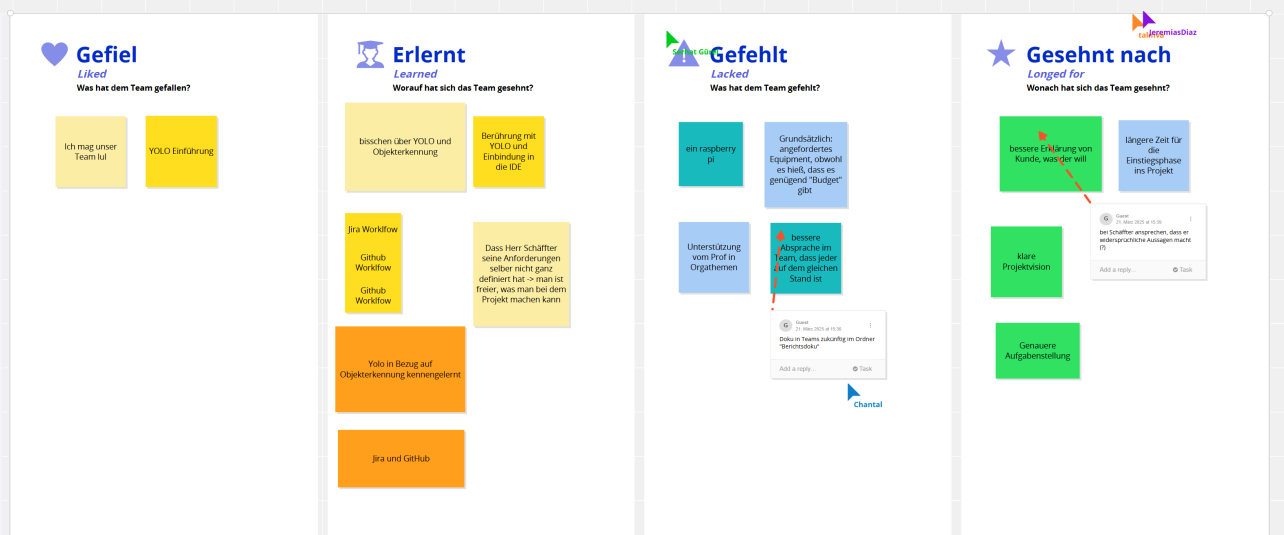


Figure 10: Retrospektive mit der 4L-Methode (durchgeführt am 21.03.2025)

Die Projektplanung basierte auf einem detaillierten Zeitplan, der die zentralen Meilensteine und Aktivitäten umfasste. Die chronologische Strukturierung des Projekts erfolgte entlang von acht wesentlichen Meilensteinen, die einen klaren Entwicklungspfad von der initialen Teamfindung bis zum Projektabschluss definierten.

Der erste Meilenstein umfasste die Phase der Teamfindung vom 6. Februar bis zum 10. März 2025, in der sich die sechs Teammitglieder zusammenfanden und erste konzeptionelle Überlegungen zum Projektumfang anstellten. Am 18. März 2025 wurde der zweite Meilenstein mit dem Abschluss der Projektplanung erreicht. In dieser Phase wurden die grundlegenden Strukturen für das Projektmanagement etabliert, Tools wie Jira und GitHub eingerichtet und erste inhaltliche Konzepte entwickelt.

Der dritte Meilenstein, die Implementierung der Kennzeichenerkennung unter Laborbedingungen, erstreckte sich vom 19. bis zum 27. März 2025. Hier lag der Fokus auf der Entwicklung eines grundlegenden Verständnisses der YOLO-Architektur und der Implementierung erster Erkennungsalgorithmen für Kennzeichen. Vom 28. März bis zum 3. April 2025 folgte der vierte Meilenstein mit der Erweiterung der Kennzeichenerkennung für das spezifische Anwendungsszenario an Mautstellen.

Der fünfte Meilenstein vom 4. bis zum 10. April 2025 widmete sich der Gesichtserkennung unter Laborbedingungen, gefolgt vom sechsten Meilenstein (11. bis 17. April 2025), der die Anwendung der Gesichtserkennung für Zutrittskontrollen adressierte. Der siebte Meilenstein vom 18. bis zum 25. April 2025 umfasste Untersuchungen zu Möglichkeiten der Manipulation von Gesichtserkennungssystemen und deren Absicherung.

Den Abschluss bildete der achte Meilenstein vom 26. April bis zum 2. Mai 2025, der die Präsentation des Projektergebnisses sowie die Finalisierung der Projektdokumentation beinhaltete. Diese strukturierte Zeitplanung ermöglichte eine kontinuierliche Fortschrittskontrolle und die frühzeitige Identifikation potenzieller Verzögerungen.

4.2.5.2.2 Backlog-Management und Fortschrittskontrolle

Das Backlog-Management wurde mithilfe von Jira realisiert, einer professionellen Software für agiles Projektmanagement. In Jira wurden sowohl das Product Backlog als auch die Sprint Backlogs systematisch gepflegt und kontinuierlich aktualisiert. In der SCRUM-Phase wurden regulär Burndown Charts erstellt, um den Fortschritt visuell zu erfassen und potenzielle Abweichungen vom geplanten Projektverlauf frühzeitig zu erkennen.

Insgesamt wurden über 90 Tickets im Jira-System erstellt und bearbeitet, die hierarchisch in

Meilenstein	Beschreibung	Zeitraum
1	Teamfindung	06.02.2025 - 10.03.2025
2	Projektplanung abgeschlossen	18.03.2025
3	Kennzeichenerkennung unter Laborbedingungen	19.03.2025 - 27.03.2025
4	Kennzeichenerkennung an Mautstellen	28.03.2025 - 03.04.2025
5	Gesichtserkennung unter Laborbedingungen	04.04.2025 - 10.04.2025
6	Gesichtserkennung für Zutrittskontrollen	11.04.2025 - 17.04.2025
7	Manipulierte Gesichtserkennung	18.04.2025 - 25.04.2025
8	Projektende mit Präsentation und Dokumentation	26.04.2025 - 02.05.2025

Table 2: Übersicht der Projektmeilensteine

acht Epic-Kategorien eingeordnet wurden, welche den definierten Meilensteinen entsprachen. Diese Struktur ermöglichte eine übersichtliche Organisation der anfallenden Aufgaben und erleichterte die Zuordnung von Verantwortlichkeiten.

Nach der Umstellung von SCRUM auf Kanban wurde das Jira-Board entsprechend angepasst, um den kontinuierlichen Arbeitsfluss transparent darzustellen. Anstelle der Sprint-gebundenen Planung trat nun die visualisierte Darstellung des Aufgabenflusses durch verschiedene Bearbeitungsstadien, was eine flexiblere Priorisierung und Bearbeitung der Aufgaben ermöglichte.

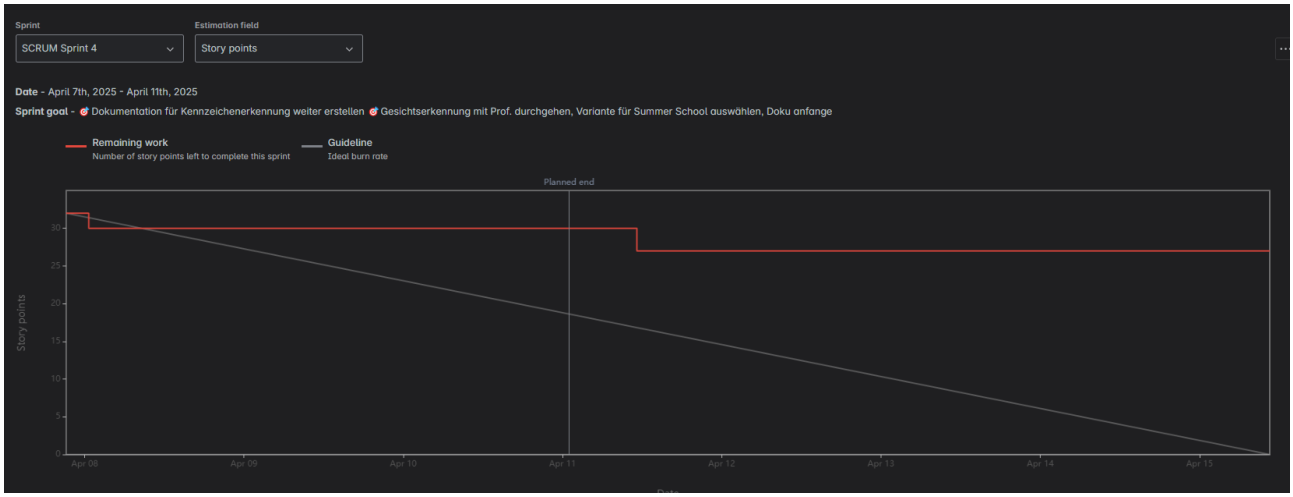


Figure 11: Burndown Chart des letzten SCRUM-Sprints vor der Umstellung auf Kanban

Die Fortschrittskontrolle erfolgte durch regelmäßige Statusberichte, die an den Betreuer, Herrn Franz, übermittelt wurden. Diese Berichte boten eine konzise Zusammenfassung des aktuellen Projektstands, der erreichten Meilensteine und der geplanten nächsten Schritte. Ergänzend wurden Sprint-Review-Dokumente erstellt, die den Fortschritt bei der Implementierung der Kennzeichenerkennung und Gesichtserkennung detailliert dokumentierten.

Ein weiteres Element der Fortschrittskontrolle waren die Protokolle der regelmäßigen Meetings, die systematisch im Discord-Channel „Schriftliches" abgelegt wurden. Diese Protokolle dienten nicht nur der Dokumentation getroffener Entscheidungen, sondern auch als Basis für die Nachverfolgung von Aktionspunkten und offenen Fragen.

4.2.5.2.3 Projektkommunikation und Dokumentationsmanagement

Die Projektkommunikation erfolgte auf verschiedenen Ebenen und wurde durch eine strukturierte Dokumentation unterstützt. Während kein klassisches Lastenheft erstellt wurde, fanden regelmäßige Abstimmungen mit dem Kunden, Herrn Schäffter, statt, um den Projektumfang agil anzupassen. Ein Beispiel für diese agile Anpassungsfähigkeit war die Erweiterung

des ursprünglich als „Personenerkennung mit YOLO" definierten Projekts um den Aspekt der Kennzeichenerkennung, um den Umfang besser an die zweimonatige Projektlaufzeit anzupassen.

To-Do-Listen wurden im Discord-Channel „Fragen an Betreuer" geführt und regelmäßig aktualisiert, um eine transparente Übersicht über anstehende Aufgaben zu gewährleisten. Diese Listen dienten als Ergänzung zum formalen Backlog in Jira und ermöglichten eine schnellere und direktere Kommunikation über kurzfristige Aufgaben.

Ein besonderer Fokus lag auf der Dokumentation der technischen Aspekte, insbesondere im Hinblick auf das zu erstellende Hands-On-Tutorial. Hier wurden detaillierte Anleitungen zur Einrichtung der Raspberry Pi-Umgebung, zur Installation der erforderlichen Bibliotheken und zur Nutzung der implementierten Erkennungsalgorithmen erstellt. Diese Dokumentation bildete die Grundlage für das Tutorial, das zukünftigen Studierenden der THU-Summerschool zur Verfügung gestellt werden sollte.

Im Bereich des Datenschutzes wurden potenzielle DSGVO-Implikationen frühzeitig mit Herrn Schäffter diskutiert, um sicherzustellen, dass die entwickelten Anwendungen den gesetzlichen Anforderungen entsprechen. Insbesondere für die Gesichtserkennung wurden Konzepte erarbeitet, die eine datenschutzkonforme Verarbeitung biometrischer Daten gewährleisteten.

4.2.5.2.4 Versionsverwaltung und kollaboratives Arbeiten

Für die Versionsverwaltung wurde GitHub als zentrales Repository genutzt. Um eine klare Trennung zwischen Entwicklung und Endprodukt zu gewährleisten, wurde das Projekt in zwei separate Repositories aufgeteilt: Ein privates Development-Repository für die Entwicklungsarbeit und ein öffentliches Repository für das Hands-on-Tutorial, das ohne direkten Zugriff auf die implementierten Lösungen genutzt werden konnte.

Die Branching-Strategie folgte einem strukturierten Ansatz, bei dem jedes Teammitglied auf einem eigenen Branch (`dev_[Name]`) arbeitete. Diese dezentrale Arbeitsweise ermöglichte parallele Entwicklungen ohne gegenseitige Beeinträchtigungen. Funktionsfähiger Code wurde auf den `development`-Branch zusammengeführt, wobei die Integration erst nach erfolgreichen Tests erfolgte. Nur vollständig fertiggestellte und getestete Funktionen wurden mittels Pull Request auf den `main`-Branch übertragen, der stets eine stabile und produktionsreife Version des Projekts repräsentierte.

Für Pull Requests wurde eine standardisierte Checkliste implementiert, die eine konsistente Qualitätssicherung gewährleistete. Diese Checkliste umfasste unter anderem die detaillierte Beschreibung der Änderung, die Kategorisierung der Art der Änderung (neue Funktion, Bugfix, Breaking Change, Dokumentationsupdate), eine Auflistung der durchgeführten Tests, die Bestätigung der Einhaltung der Stilrichtlinien, den Nachweis eines Self-Reviews, die Kommentierung komplexer Codeabschnitte, die Anpassung der zugehörigen Dokumentation, die Vermeidung neuer Warnungen, das Hinzufügen von Tests für neue Funktionen sowie die erfolgreiche lokale Ausführung aller Tests. Diese umfassende Checkliste stellte sicher, dass nur qualitativ hochwertiger und gut dokumentierter Code in das Hauptrepository integriert wurde.

Das kollaborative Arbeiten wurde durch diese strukturierte Versionsverwaltung erheblich erleichtert. Die klare Trennung der Arbeitsbereiche durch individuelle Branches verhinderte Konflikte bei der gleichzeitigen Bearbeitung von Dateien, während die zentralisierte Integration über den `development`-Branch eine kontinuierliche Integration der Einzelbeiträge zu einem kohärenten Gesamtsystem ermöglichte.

4.2.5.2.5 Reflexion und Verbesserungspotential

Die Entscheidung, von SCRUM zu Kanban zu wechseln, erwies sich im Projektverlauf als vorteilhaft und angemessen. Die anfängliche SCRUM-Struktur bot einen soliden Rahmen für die Projektinitialisierung und -planung, während Kanban in der Umsetzungsphase die erforderliche Flexibilität bereitstellte. Diese Anpassungsfähigkeit war besonders wertvoll angesichts des

experimentellen Charakters des Projekts und der Tatsache, dass alle Teammitglieder parallel zu ihren regulären Studienverpflichtungen am Projekt arbeiteten.

In der Retrospektive lassen sich dennoch verschiedene Bereiche identifizieren, in denen Verbesserungspotential bestand. Eine präzisere Einschätzung des Arbeitsaufwands zu Beginn des Projekts hätte möglicherweise zu einer realistischeren Zeitplanung geführt und damit einige der zeitlichen Herausforderungen in den späteren Projektphasen reduziert. Die anfängliche Planung erwies sich teilweise als zu optimistisch, insbesondere hinsichtlich der Komplexität der Implementierung der Gesichtserkennung.

Der Wechsel zu Kanban hätte eventuell früher erfolgen können, da der administrative Aufwand für die SCRUM-Methodik teilweise Ressourcen band, die für die technische Umsetzung hätten genutzt werden können. Eine frühere Erkenntnis der besseren Eignung des Kanban-Ansatzes für dieses spezifische Projekt hätte möglicherweise zu einer effizienteren Ressourcennutzung geführt.

Die Dokumentation der technischen Aspekte erfolgte teilweise erst gegen Projektende, was den Abschlussprozess intensivierte. Eine kontinuierlichere Dokumentation parallel zur Entwicklung hätte eine gleichmäßigere Arbeitsbelastung ermöglicht und potenziell zu einer höheren Dokumentationsqualität geführt. Zudem hätte eine intensivere Testphase, insbesondere für die Überprüfung des Hands-On-Tutorials durch unbeteiligte Dritte, möglicherweise weitere Verbesserungspotentiale aufgedeckt.

Trotz dieser Optimierungsmöglichkeiten kann das Projektmanagement insgesamt als erfolgreich betrachtet werden. Die flexible Kombination aus SCRUM und Kanban ermöglichte es dem Team, die auftretenden Herausforderungen zu bewältigen und die primären Projektziele zu erreichen. Die klare Strukturierung in Meilensteine, die transparente Kommunikation und die systematische Versionsverwaltung trugen wesentlich zum Gelingen des Projekts bei. Die entwickelten Erkennungsalgorithmen für Kennzeichen und Gesichter sowie das erstellte Hands-On-Tutorial erfüllen die anfangs definierten Anforderungen und bieten eine solide Grundlage für zukünftige Weiterentwicklungen und Anwendungen im Rahmen der THU-Summerschool.

5 Ergebnis

5.1 Zusammenfassung der Ergebnisse aus Nutzersicht

Um die Praxistauglichkeit unseres Hands-On Tutorials zur Gesichts- und Schrifterkennung zu bewerten, haben wir Studierende der Technischen Hochschule Ulm (THU) sowie Studierende der Hochschule Neu-Ulm (HNU) eingeladen, das Tutorial selbstständig zu bearbeiten. Dabei stellten wir die notwendige Hardware zur Verfügung und leisteten bewusst nur minimale Hilfestellung, um ein möglichst realistisches Bild der Benutzerfreundlichkeit zu erhalten. Für die Bearbeitungszeit wählten wir einen Zeitraum von etwa dreieinhalb Stunden. Zwar ist für die Summer School ein Zeitraum von vier Stunden für die Bearbeitung vorgesehen, dennoch haben wir die Bearbeitungszeit für unseren Testdurchlauf gekürzt da wir auch Verspätungen oder Verzögerungen im Plan abdecken möchten. Das soll den Studenten die Chance zu geben das Tutorial vollständig abschließen zu können. In diesem Kapitel fassen wir die Rückmeldungen, Beobachtungen und Verbesserungsvorschläge der Teilnehmenden zusammen.

Ein Studierender (Studiengang: BWL) ohne formale Ausbildung in einem technischen Studiengang, jedoch mit ersten Erfahrungen in Webentwicklung, bearbeitete das Tutorial. Er berichtete, dass die grundlegende Konfiguration des Raspberry Pi für ihn überraschend reibungslos verlief, was er insbesondere auf die Bereitstellung eines klar strukturierten Installationsskripts zurückführte. Besonders die erste Verbindung zur Hardware und das Starten des Systems verliefen für ihn problemlos, da das Skript es ermöglichte, die notwendigen Schritte sicher und ohne größere Fehlermöglichkeiten auszuführen. Die eigentliche Herausforderung bestand für ihn darin, sich als jemand ohne tiefere Kenntnisse in Python-Programmierung sowie in der Bearbeitung von Bildmaterial zurechtzufinden, insbesondere im Abschnitt zur Erkennung und Auswertung von Kfz-Kennzeichen. Mit Mühe und kleinen Hilfestellungen von unserer Seite gelang es ihm, fast das vollständige Tutorial innerhalb des vorgesehenen Bearbeitungszeitraums von etwa drei bis vier Stunden abzuschließen. Zwei weitere Studenten (Studiengänge: Information Management Automotive, Informationsmanagement und Unternehmenskommunikation), die ebenfalls keinen technischen Hintergrund haben und sich im Gegensatz zum ersten Teilnehmer nicht mit Webentwicklung auseinandergesetzt haben, bearbeiteten gemeinsam dieses Tutorial. Auch hier war das Einrichten des Raspberry Pi kein Problem, die Aufgaben zur Kennzeichen- und Personenerkennung jedoch herausfordernd. Trotz Bemühungen und kleinen Hilfestellungen haben die Studenten es nicht geschafft das Tutorial vollständig abzuschließen. Auch hier war die fehlende Programmiererfahrung eine Hürde. Bei den Teilnehmenden mit technischem Hintergrund, bestehend aus drei Informatikstudenten (zwei im sechsten und einer im dritten Semester) sowie einem Fahrzeugtechnikstudenten im dritten Semester, verlief die Bearbeitung der Aufgaben deutlich reibungsloser. Alle Studenten äußerten, dass das Tutorial trotz einiger technischer Herausforderungen grundsätzlich gut bewältigbar sei. Besonders hervorgehoben wurde dabei, dass die Aufgaben zwar fordernd, aber mit gegebenenfalls gegenseitiger Unterstützung gut lösbar seien.

Da wir im Rahmen der THU Summer School überwiegend mit Studierenden mit technischem Hintergrund rechnen, entschieden wir uns bewusst dafür, die Schwierigkeit des Tutorials nicht weiter zu reduzieren. Die Möglichkeit der Zusammenarbeit unter den Teilnehmerinnen und Teilnehmern soll eventuelle Hürden abmildern und gleichzeitig die Teamarbeit fördern. Auf Basis des Feedbacks aus den Testläufen entschieden wir uns außerdem, das Lehrmaterial in Form einer PowerPoint-Präsentation aufzubereiten. Diese Entscheidung stützt sich sowohl auf die Unterlagen unseres Auftraggebers aus der vergangenen Summer School als auch auf die Rückmeldungen der testenden Studierenden, die die strukturierte und übersichtliche Aufbereitung in PowerPoint-Form positiv bewerteten.

5.2 Fehlerquellen und potentielle Optimierungen

Das Hauptergebnis unseres Projekts ist die Entwicklung eines strukturierten Hands-On Tutorials zur Gesichts- und Schrifterkennung auf einem Raspberry Pi. Im Gegensatz zu klassischen Projekten, bei denen ein fertiges Produkt im Vordergrund steht, handelt es sich bei unserem Projekt um die Erstellung von Lehrmaterialien, deren tatsächlicher Erfolg erst im Einsatz durch die Studierenden der THU Summer School sichtbar wird. Zur vorläufigen Evaluation des Materials führten wir, wie bereits beschrieben erste Testläufe mit Studierenden unterschiedlicher technischer Hintergründe durch. Eine abschließende Bewertung hinsichtlich Verständlichkeit, Schwierigkeitsgrad und Lerneffekt des Tutorials kann jedoch erst erfolgen, wenn es im Rahmen der Summer School eingesetzt und von einer größeren Gruppe internationaler Studierender bearbeitet worden ist. Wir erwarten, dass das Lernmaterial den Lernprozess unterstützt.

6 Diskussion

6.1 Welche Ziele wurden erreicht?

In diesem Abschnitt wird der Abgleich der ursprünglich definierten Projektziele (siehe Abschnitt 1.2) mit den tatsächlich erzielten Ergebnissen (dargestellt in Abschnitt 5) vorgenommen.

- **Primärziel - Hands-On Tutorial:** Es wurde erfolgreich ein praxisorientiertes Hands-On-Tutorial für die THU Summer School entwickelt. Der Kern des Tutorials liegt in der kostengünstigen Umsetzung von Methoden zur Personen- und Schrifterkennung mittels eines Raspberry Pi. Es demonstriert Computer-Vision-Technologien, primär auf Basis von YOLOv5.
  - **Kennzeichenerkennung (OCR):** Das entwickelte System ist in der Lage, Kfz-Kennzeichen aus Bilddateien zu detektieren und deren alphanumerischen Inhalt auszulesen. Eine Optimierung erfolgte spezifisch für deutsche Kennzeichenformate. Die durchgeführten Testläufe illustrierten sowohl die grundsätzliche Funktionsfähigkeit als auch die typischen Fehlerquellen unter Laborbedingungen.
  - **Personenerkennung (Gesichter):** Es wurde eine KI-gestützte Personenerkennung implementiert, die das Erlernen, Unterscheiden und Wiedererkennen von Gesichtern ermöglicht. Das Tutorial vermittelt den Studierenden den Umgang mit den Modellen YOLO und MediaPipe für diese Aufgabe.
- **Sekundärziel - Anwendungsszenarien:** Die beiden Erkennungsaufgaben dienen als praxisnahe Demonstrationen für potenzielle Anwendungsfälle, wie automatisierte Mautprozesse oder Zutrittskontrollsysteme. Ziel war es hierbei, ein grundlegendes Verständnis für die technische Realisierung und die damit verbundenen Herausforderungen zu schaffen.
- **Technische Umsetzung und Plattform:** Das Tutorial wurde erfolgreich auf einem Raspberry Pi implementiert. Dies bestätigt die Machbarkeit der vorgestellten Methoden auf ressourcenbeschränkter Hardware und gibt Einblicke in das Feld des *Tiny Machine Learning*.
- **Evaluation und Lehrmaterial:** Erste Testläufe mit Studierenden lieferten wertvolles Feedback. Basierend darauf wurde das Lehrmaterial in Form einer PowerPoint-Präsentation aufbereitet und um eine optionale Zusatzaufgabe zur Vertiefung der Kennzeichenerkennung ergänzt.

6.2 Retrospektive

6.2.1 Was lief gut?

Zusätzlich zu den bereits etablierten positiven Aspekten der Teamarbeit und Dokumentation können folgende Punkte hervorgehoben werden:

- Die Kommunikation im Team war durchweg konstruktiv und effizient.
- Die vereinbarte Arbeitsteilung funktionierte gut und wurde konsequent eingehalten.
- Die Projektdokumentation ist umfassend und klar strukturiert.
- Der entwickelte Code ist gut organisiert und ausreichend kommentiert.

- **Testläufe und Feedback:** Studierende mit technischem Hintergrund konnten das Tutorial erfolgreich bearbeiten. Das bereitgestellte Installationsskript wurde als große Hilfe bei der Einrichtung des Raspberry Pi empfunden. Die Aufbereitung der Lehrmaterialien als PowerPoint-Präsentation fand Anklang.
- **Gesichtserkennung (YOLO):** Die Kombination aus YOLOv8n-face und dem `face_recognition`-Framework ermöglichte eine zuverlässige Gesichtserkennung (ca. 93% in Tests).
- **Gesichtserkennung (MediaPipe):** Die Gesichtspunkterkennung mittels MediaPipe Face Mesh erwies sich als präzise und echtzeitfähig. Das darauf aufbauende Wiedererkennungssystem zeigte trotz bekannter Limitationen eine für viele Anwendungsfälle ausreichende Performance (ca. 23 FPS im Test).

6.2.2 Was lief nicht so gut?

Neben den Erfolgen traten auch Herausforderungen und Limitationen auf, die hier reflektiert werden:

- **Kennzeichenerkennung - Universalität:** Die enorme globale Vielfalt an Kennzeichenformaten und Schriftarten verhinderte die Entwicklung einer universell einsetzbaren Lösung im Rahmen des Projekts. Der Fokus musste daher auf deutsche Kennzeichen gelegt werden.
- **Kennzeichenerkennung - Robustheit unter Realbedingungen:** Die Anpassung des Systems an stark variierende Lichtverhältnisse (Schatten, Überbelichtung) sowie an Verschmutzungen auf den Kennzeichen erwies sich als komplex. Eine robuste Lösung für alle denkbaren Umgebungsbedingungen hätte den Projektrahmen gesprengt.
- **Kennzeichenerkennung - OCR-Genauigkeit:** Die verwendete OCR-Engine (Tesseract) zeigte Schwächen bei der Unterscheidung ähnlicher Zeichen wie '0', 'G' und 'O', was zu Fehlern bei der Textextraktion führte.
- **Hardwarelimitierungen des Raspberry Pi:** Insbesondere die OCR-Komponente zur Schriftzeichenerkennung stellte hohe Anforderungen an die Rechenleistung des Raspberry Pi, was die Echtzeitfähigkeit des Gesamtsystems einschränkte. Der Einsatz von EasyOCR scheiterte an dessen hohem Speicherbedarf und den langen Inferenzzeiten auf der Zielhardware.
- **Gesichtserkennung (MediaPipe - Wiedererkennung):** MediaPipe allein ist für robuste biometrische Identifikationsaufgaben, wie sie bei Zutrittskontrollen benötigt werden, nur eingeschränkt tauglich. Die verwendete Landmarken-Distanz-Methode ist anfällig für Variationen in Mimik, Kopfhaltung und Beleuchtung. Zudem fehlt eine integrierte Lebenderkennung (*Liveness Detection*), um Täuschungsversuche (z.B. durch Vorhalten eines Fotos) abzuwehren.
- **Tutorial-Schwierigkeitsgrad:** Ein Testlauf mit einem Studierenden ohne spezifischen technischen Hintergrund zeigte, dass das Tutorial für diese Zielgruppe potenziell eine hohe Einstiegshürde darstellt und die Bearbeitung innerhalb der vorgesehenen Zeit schwierig sein kann.

6.2.3 Lessons Learned

6.2.3.1 Valentin Talmon-l'Armée Im Verlauf des Projekts habe ich wertvolle Einblicke in die praktische Anwendung von KI-Verfahren und die Bildverarbeitung gewonnen. Besonders das Arbeiten mit dem Raspberry Pi und die Nutzung von Git zur Versionskontrolle haben mein technisches Verständnis erweitert. Ich habe gelernt, dass neben der technischen Umsetzung auch eine klare und regelmäßige Abstimmung im Team entscheidend für den Projekterfolg ist. Die iterative Auseinandersetzung mit OCR- und Detektionsmethoden hat mir gezeigt, wie wichtig systematisches Testen und Anpassen im Entwicklungsprozess ist. Insgesamt konnte ich mein Wissen in mehreren Bereichen gezielt vertiefen und besser miteinander verknüpfen.

6.2.3.2 Jan Gaschler Das Projekt hat mir nicht nur technische Fähigkeiten vermittelt, sondern auch meine Teamarbeit und Kommunikationsfähigkeiten gestärkt. Die enge Zusammenarbeit mit meinen Kommilitonen war entscheidend für den Erfolg des Projekts. Ich habe gelernt, wie wichtig es ist, Feedback zu geben und zu empfangen, um gemeinsam Lösungen zu finden. Besonders die Herausforderungen bei der Implementierung der Gesichtserkennung haben mir gezeigt, wie wichtig es ist, flexibel zu bleiben und alternative Ansätze in Betracht zu ziehen. Ich habe auch ein besseres Verständnis für die ethischen Implikationen von Gesichtserkennungstechnologien entwickelt und werde diese Überlegungen in zukünftige Projekte einfließen lassen.

6.2.3.3 Chantal Deusch Im Rahmen des Projekts konnte ich wichtige Erfahrungen in der technischen Umsetzung von Gesichtserkennung und -wiedererkennung, sowohl auf Pixelebene als auch im Bereich der Biometrie, sammeln. Durch die Arbeit mit Technologien wie YOLO und Mediapipe habe ich neue Methoden und Ansätze kennengelernt. Besonders wichtig war zudem die Erkenntnis, dass Kommunikation und regelmäßige Abstimmungen im Team entscheidend für den Projekterfolg sind. Trotz sorgfältiger Planung traten immer wieder neue Herausforderungen auf, die Flexibilität und zusätzliche Zeit erforderten. Dadurch habe ich gelernt, dass technische Projekte oft komplexer und zeitintensiver sind als anfangs erwartet. Insgesamt war das Projekt eine wertvolle Erfahrung, sowohl im technischen als auch im organisatorischen Bereich.

6.2.3.4 Serhat Gürel Das Projekt bot mir die Möglichkeit, erste Kenntnisse im Bereich der Kunstlichen Intelligenz zu erwerben und ein grundlegendes Verständnis für diese komplexe Technologie zu entwickeln. Dabei wurde mir sowohl das Potenzial als auch die Grenzen intelligenter Systeme bewusst. Auch die Zusammenarbeit in einem Team an einem größeren Projekt war eine weitere lehrreiche Erfahrung. Das Arbeiten mit Git und Jira war anfangs zwar eine Hürde, aber im Laufe des Projektes und mit gegenseitiger Unterstützung im Team konnte auch diese erfolgreich überwunden werden. Besonders deutlich wurde mir, wie entscheidend regelmäßige Kommunikation und ein strukturierter Austausch im Team für den Projekterfolg sind. Abschließend kann ich sagen, dass mich das Projekt sowohl in technischer, planerischer und persönlicher Hinsicht weitergebracht hat.

7 Ausblick und zukünftige Entwicklungen

7.1 Ausblick YOLO

Da YOLO ein Open-Source-Modell mit einer aktiven und stetig wachsenden Community ist, wird es kontinuierlich weiterentwickelt und verbessert. Angesichts dieser Dynamik ist davon auszugehen, dass YOLO auch in den kommenden Jahren an Relevanz gewinnen wird. Besonders spannend ist die bereits angekündigte Version YOLOv12, die neue Features und Implementierungen einführen soll mit dem Ziel, die Erkennungsgenauigkeit und Effizienz nochmals deutlich zu steigern.

7.2 YOLOv12

Das Schlüsselkonzept von YOLOv12 liegt darin eine neuartige, aufmerksamkeitszentrierte Architektur für die Objekterkennung zu sein, die auf Echtzeitleistung mit hoher Genauigkeit kombiniert. [14]

7.2.1 Wesentliche Merkmale

- **Mechanismus der Bereichsaufmerksamkeit:** Effiziente Verarbeitung großer rezeptiver Felder durch Aufteilung der Merkmalskarten in gleich große Regionen, was die Rechenkosten senkt.
- **Resteffiziente Schichtaggregationsnetze (R-ELAN):** Verbesserte Merkmalsaggregation zur Optimierung von Trainingsprozessen bei großen Modellen.
- **Optimierte Aufmerksamkeitsarchitektur:** Minimierung des Speicherzugriffs-Overheads durch FlashAttention, Vermeidung von Positionskodierung und Anpassung der MLP-Verhältnisse.
- **Umfassende Aufgabenunterstützung:** Unterstützung von Objekterkennung, Instanzsegmentierung, Bildklassifizierung, Posenschätzung und orientierter Objekterkennung (OBB).

7.2.2 Unterstütze Aufgaben und Modi

Modell Typ	Aufgabe	Inferenz	Validierung	Ausbildung	Exportieren
YOLO12-seg	Segmentierung	✓	✓	✓	✓
YOLO12-Pose	Pose	✓	✓	✓	✓
YOLO12-obb	OBB	✓	✓	✓	✓
YOLO12-cls	Klassifizierung	✓	✓	✓	✓
YOLO12	Erkennung	✓	✓	✓	✓

Figure 12: Unterstützte Aufgaben und Modis YOLOv12

7.2.3 Leistungsmetriken

Detektionsleistung (COCO val2017): YOLO12 zeigt signifikante Verbesserungen in der mAP (mean Average Precision) über alle Modellskalen.

Modell	Größe (Pixel)	mAPval 50–95	T4 TensorRT (ms)	Params (M)	FLOPs (B)	Vergleich
YOLO12n	640	40.6	1.64	2.6	6.5	+2,1% / –9% (ggü. YOLOv10n)
YOLO12s	640	48.0	2.61	9.3	21.4	+0,1% / +42% (ggü. RT-DETRv2)
YOLO12m	640	52.5	4.86	20.2	67.5	+1,0% / –3% (vs. YOLO11m)
YOLO12l	640	53.7	6.77	26.4	88.9	+0,4% / –8% (ggü. YOLO11l)
YOLO12x	640	55.2	11.79	59.1	199.0	+0,6% / –4% (ggü. YOLO11x)

Table 3: Vergleich verschiedener YOLOv12-Modelle

7.2.4 Wichtige Verbesserungen

Das YOLOv12-Modell bringt bedeutende Fortschritte in zwei zentralen Bereichen mit sich. Zum einen wurde die **Merkmalsextraktion** optimiert, wodurch das Modell große rezeptive Felder effizienter verarbeiten kann. Gleichzeitig wird das Zusammenspiel zwischen aufmerksamkeitbasierten Mechanismen und klassischen Feedforward-Netzwerken gezielter ausbalanciert. Zum anderen überzeugt YOLOv12 durch eine gesteigerte **architektonische Effizienz**, indem es die Anzahl der Modellparameter reduziert ohne dabei Einbußen bei der Genauigkeit hinzunehmen, teilweise sogar mit Verbesserungen.

7.2.5 Zusammenfassung und Wichtige Erkenntnisse

YOLOv12 ist flexibel einsetzbar und deckt viele Bereiche der Computer Vision ab. Dank seiner überarbeiteten Architektur gelingt eine starke Balance zwischen Genauigkeit und Geschwindigkeit. Außerdem unterstützt das Modell verschiedene Modi, was den Einsatz in unterschiedlichsten Anwendungen ermöglicht von Robotik bis hin zur medizinischen Bildverarbeitung.

7.3 Mikrocomputer gewinnen an Bedeutung

Mikrocomputer gewinnen in der heutigen Technologie- und Informationsgesellschaft zunehmend an Bedeutung. Ihre kompakte Bauweise, Energieeffizienz und vielseitige Einsetzbarkeit machen sie zu Schlüsselkomponenten in zahlreichen Anwendungen.[15], [16]

Kompakte Bauweise und Energieeffizienz: Moderne Mikrocomputer sind darauf ausgelegt, hohe Rechenleistung auf kleinem Raum bereitzustellen. Durch fortschrittliche Fertigungstechnologien können sie komplexe Aufgaben effizient bewältigen und dabei den Energieverbrauch minimieren. Dies ist besonders wichtig für mobile Geräte und eingebettete Systeme, bei denen Platz und Energie begrenzt sind.

Vielseitige Einsatzmöglichkeiten: Die Anwendungsbereiche von Mikrocomputern sind breit gefächert. Sie finden Verwendung in Haushaltsgeräten, Fahrzeugen, medizinischen Geräten, industriellen Steuerungen und vielen weiteren Bereichen. Ihre Fähigkeit, spezifische Aufgaben zuverlässig und effizient zu erfüllen, macht sie in der modernen Technik unverzichtbar.

Unterstützung innovativer Technologien: Mikrocomputer spielen eine zentrale Rolle bei der Umsetzung neuer Technologien wie de3m Internet of Things (IOT), KI, und der Automatisierung. Sie ermöglichen die Verarbeitung und Analyse von Daten in Echtzeit, was für die Entwicklung intelliegenter Systeme und Anwendungen entscheidend ist.

7.4 Langlebigkeit des Tutorials

Durch die Ausarbeitung und erfolgreiche Durchführung des Tutorials mit Studierenden hoffen wir, dass die Technische Hochschule dieses in den kommenden Jahren für die Summer School einsetzen kann. Da das Tutorial bereits in Zusammenarbeit mit Studierenden getestet wurde

mit überwiegend positiven Ergebnissen sind wir zuversichtlich hinsichtlich seiner zukünftigen Anwendung. Wie in Abschnitt 7.3 erwähnt, gewinnen Mikrocomputer zunehmend an Bedeutung, was dem Projekt einen zukunftsorientierten Aspekt verleiht. Zudem wird durch die kontinuierliche Weiterentwicklung und Verbesserung des YOLO-Modells auch in den kommenden Jahren eine hohe Relevanz und Anwendungsbedeutung erwartet.

8 Quellenverzeichnis

[1] Ultralytics, *Everything you need to know about computer vision in 2025*, Online, Verfügbar unter: https://www.ultralytics.com/de/blog/everything-you-need-to-know-about-computer-vision-in-2025, 2025. Accessed: May 4, 2025.

[2] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek, "A brief introduction to opencv," in *2012 Proceedings of the 35th International Convention MIPRO*, 2012, pp. 1725–1730.

[3] J.-O. Schneppat, *Opencv*, https://gpt5.blog/opencv/, n.d.

[4] doxygen, *Image processing in opencv*, https://docs.opencv.org/4.x/d2/d96/tutorial_py_table_of_contents_imgproc.html, 2025.

[5] Google, *Mediapipe solutions guide*, Online, Verfügbar unter: https://ai.google.dev/edge/mediapipe/solutions/guide?hl=de, 2025. Accessed: May 4, 2025.

[6] Google, *Mediapipe face mesh*, Online, Verfügbar unter: https://github.com/google-ai-edge/mediapipe/blob/master/docs/solutions/face_mesh.md, 2023. Accessed: Apr. 28, 2025.

[7] Google, *Mediapipe face mesh topology*, https://developers.google.com/mediapipe/solutions/vision/face_landmarker, 2024.

[8] Ultralytics, *Evolution of yolo architecture*, Online, Verfügbar unter: https://cdn.prod.website-files.com/6479eab6eb2ed5e597810e9e/67ed560c30286ca00c535cd2_671248268bb146c11824aaf4_671245f5cf139c5400d2bc26_Evolution_fig4.png, 2023. Accessed: Apr. 25, 2025.

[9] Tesseract-OCR, *Tessdoc*, https://tesseract-ocr.github.io/tessdoc/, 2024. Accessed: Apr. 25, 2025.

[10] Roboflow, *License plate recognition dataset (roboflow universe)*, https://universe.roboflow.com/roboflow-universe-projects/license-plate-recognition-rxg4e, 2024. Accessed: Apr. 25, 2025.

[11] GeeksforGeeks, *License plate recognition with opencv and tesseract ocr*, https://www.geeksforgeeks.org/license-plate-recognition-with-opencv-and-tesseract-ocr/, 2021. Accessed: Apr. 25, 2025.

[12] M. AI, *Mastering alpr in wild environments*, https://blog.marvik.ai/2024/09/23/alpr-in-wild-environments/, 2024. Accessed: Apr. 25, 2025.

[13] C. Deusch, J. Diem, J. Gaschler, S. Gürel, P. Pyczot, and V. Talmon-l'Armée, *Yolo github repository topr-ss25*, Online, GitHub Repository vom teamorientierten Projekt TOPR-SS25; verfügbar unter: https://github.com/TOPR-yoloteam/YOLO, 2025. Accessed: May 4, 2025.

[14] G. Jocher, M. Yasin, M. R. Munawar, and Laughing, *Yolo12: Aufmerksamkeitszentrierte objekterkennung*, https://docs.ultralytics.com/de/models/yolo12/#where-can-i-find-usage-examples-and-more-detailed-documentation-for-yolo12, 2025.

[15] G. Wright and S. Shea, *What is a microcomputer?* https://https://www.techtarget.com/iotagenda/definition/microcomputer, 2024.

[16] Jer.re10542ce, R. Weemeyer, Ijbond, and PeterPaan, *Mikrocomputer*, https://de.wikipedia.org/wiki/Mikrocomputer, 2024.

9 Anhang