FH Westküste
Wirtschaft und Technik

SmartFruitVend

Sushma Sushma

Loic Merveil Tchatat Naheu

Moise Kamdem Ngnobou

Konzept für das Modul Echtzeit Bildverarbeitung

bei

M.Sc. Vitali Czymmek

FH-Westküste Master Automatisierungstechnik

21.11.2025 WiSe 25/26

Inhaltsverzeichnis

1 Einleitung 1
2 Detallierte Angaben 2
2.1 Rahmenbedingungen 2
2.2 Labeln 3
2.3 Aufgabenverteilung 3
2.4 Zeitplan 5
2.4. Klassen/Obstsorten 6
3 Projekt Durchführung 7
3.1 Konzepterstellung 7
3.1.1 Ausgangslage und Motivation 7
3.1.2 Zielsetzung und betrachtete Klassen 7
3.1.3 Nutzungskonzept (Automat) und Erweiterung durch Android-App 8
3.1.4 Systemkonzept und Ablaufkette (vom Bild zur Kalorienanzeige) 8
3.2 Implementierung 10
3.2.1 Datensatz und Labeling 10
3.2.2 Modelltraining 11
3.2.3 Android-App-Integration 14
3.3 Ergebnisse 25
3.3.1 Hyperparameter-Optimierung 29
4 Fazit und Ausblick 38
5 Literaturverzeichnis 39

Abbildungsverzeichnis

Abbildung 1: Beispiel Aufnahme Bildern 3
Abbildung 2: Struktur des Datasets 11
Abbildung 3: Code für Aufteilung der Daten 11
Abbildung 4:Befehl, um die Ultralytics Bibliothek zu installieren 12
Abbildung 5: Erstellung YAML-Datei 12
Abbildung 6: Befehl zum Starten des Trainings 13
Abbildung 7: Ergebnisse nach Training des Modells 13
Abbildung 8: Ergebnisse nach Training des Modells 14
Abbildung 9: Ergebnis des Befehls 15
Abbildung 10: Android Studio 17
Abbildung 11:Bibliotheken in der Gradle-Konfiguration 17
Abbildung 12: Ausführung des Modells(best_float16.tflite) in der APK 18
Abbildung 13:Ausführung des Labeln-Files 18
Abbildung 14: Initialisierung des CameraX-Preview-Use-Cases in startCamera 19
Abbildung 15: Ausschnitt der Inferenz-Pipeline in der Android-App 19
Abbildung 16:Einsatzt der Letterbox verfahren 20
Abbildung 17:Verarbeitung des Single-Tensor-Outputs 20
Abbildung 18: Auslesung der Tensorswerte für Bounding Boxes 21
Abbildung 19:Einschätzung der kcal von jede klasse 22
Abbildung 20:geometrische Transformation des Modells 23
Abbildung 21:Screenschot der Apps 24
Abbildung 22:Trainingsergebnisse final 25
Abbildung 23:Beispielhafte Detektionen mit Bounding Boxes 26
Abbildung 24: Normalisierte Confusion Matrix 27
Abbildung 25: Instanzen pro Klasse 28
Abbildung 26: Precision-Recall-Kurve 28
Abbildung 27:Referenztest für Hyperparameter-Optimierung 29
Abbildung 28:Ergebnis des Trainings der Datasets mit yolo11n 30
Abbildung 29:Befehl zur Ausführung der EarlyStopping 30
Abbildung 30: Ergebnis EarlyStopping 31
Abbildung 31:Ergebnis Batch Size 32 32
Abbildung 32:Ergebnis Batch Size 8 32
Abbildung 33:Ergebnis Mixup 0.2 33

Abbildung 34:Befehl des Modells 34
Abbildung 35:Ergebnis der Befehl..... 34
Abbildung 36:App-Erkennung Apple mit Kalorien 35
Abbildung 37:App-Erkennung Banana mit Kalorien 36
Abbildung 38:App-Erkennung Mango mit Kalorien 36
Abbildung 39: App-Scan: Mehrklassen-Erkennung..... 37

Tabellenverzeichnis

Tabelle 1:Aufgabenverteilung 3
Tabelle 2: Zeitplan für die Projektphase 5
Tabelle 3:Beispiele für mögliche Klassen 6
Tabelle 4:Vergleich des Modells 30
Tabelle 5:Vergleich von unterschiedlicher Metrik pro Klasse 31
Tabelle 6:Einfluss der Veränderung der Batch Size und Vergleich 33
Tabelle 7:Vergleich der MixUp-parameter .....**Fehler! Textmarke nicht definiert.**

1 Einleitung

In modernen Verkaufsautomaten werden zunehmend nicht nur Getränke, sondern auch frische Snacks wie Obst angeboten. Vor allem im Kontext einer gesundheitsbewussten Ernährung wünschen sich viele Kundinnen und Kunden einen schnellen und unkomplizierten Zugang zu frischen Lebensmitteln – idealerweise rund um die Uhr und ohne lange Wartezeiten. Der Verkaufsprozess in herkömmlichen Automaten ist jedoch meist starr: Jedes Fach ist fest einer bestimmten Obstsorte oder einem Produkt zugeordnet, und es werden keine zusätzlichen Informationen wie Nährwerte oder Kalorien angezeigt. Dadurch bleibt das Potenzial digitaler und intelligenter Systeme ungenutzt.

Das vorliegende Projekt beschäftigt sich mit der Entwicklung eines kamerabasierten Erkennungssystems für Obst in einem Verkaufsautomaten. Im Fokus stehen die drei häufig angebotenen Früchte Apfel, Banane und Mango. Eine fest über der Ablagefläche installierte Kamera erfasst die Frucht, sobald diese in den Erkennungsbereich gelegt oder vom Automaten bereitgestellt wird. Mithilfe eines neuronalen Netzes auf Basis von YOLOv8 wird in Echtzeit bestimmt, um welche Obstsorte es sich handelt.

Sobald die Frucht erfolgreich erkannt wurde, berechnet das System auf Grundlage der Obstsorte eine ungefähre Kalorienmenge und zeigt diese dem Kunden auf einem Display an. Hierbei können zur Vereinfachung typische Durchschnittswerte pro Stück oder pro Portion verwendet werden. Dadurch erhält der Benutzer nicht nur das gewünschte Produkt, sondern gleichzeitig eine direkte Rückmeldung über den ungefähren Energiegehalt seines Snacks.

2 Detallierte Angaben

2.1 Rahmenbedingungen

Die Entwicklung des Erkennungssystems basiert auf einem realitätsnahen Datensatz sowie auf den Anforderungen, die sich aus dem späteren Einsatz im Bereich des Smart-Vending-Machines ergeben. Die folgenden Bedingungen prägen den Aufbau, das Training und den späteren Einsatz des Modells.

Da das System später in einer intelligenten Verkaufsmaschine eingesetzt werden soll, stellt es nicht nur für einzelne Obststücke, sondern auch für mehrere Früchte zur gleichen Zeit, die nebeneinander liegen und gescannt werden müssen, besondere Anforderungen dar.

- Unterschiedliche Beleuchtungssituationen
    - Schatten
    - Direkte Lichtquellen
    - Tageslicht
- Verschiedene Objektlagen und Perspektiven
    - Rotationen
    - Neigungen
    - Unterschiedliche Positionen im Bild
- Unterschiedliche Anordnungen
    - Geordnet
    - überschneidend
    - Teilweise verdeckt
    - Dichtaneinander
- Reale Bildstörungen
    - Reflexionen
    - Schattenbildung
    - Kamera-Artefakte

Diese Variationen sichern, dass das Modell hinreichend robust ist, um Früchte im realen Einsatzkontext eines Smart-Vending-Systems zu erkennen.

Das System muss in einem variablen Scan-Bereich automatisch erkennen, wie viele Früchte vorhanden sind, die Sorte jeder einzelnen Frucht bestimmen und die geschätzte Gesamtkalorienanzahl berechnen.

Abbildung 1: Beispiel Aufnahme Bildern

2.2 Labeln

Für das Training des neuronalen Netzes wurde ein eigener Datensatz mit Bildern von Äpfeln, Bananen und Mangos erstellt. Diese Bilder stammen sowohl aus eigenen Aufnahmen unter verschiedenen Licht- und Hintergrundbedingungen als auch aus frei verfügbaren Online-Quellen. Anschließend wurden alle Bilder mit der Software *Labelstudio* manuell gelabelt, indem die Früchte mit sogenannten Bounding Boxes markiert und den jeweiligen Klassen (Apfel = 1, Banane = 2, Mango = 3) zugeordnet wurden. Die Labeldaten wurden im YOLO-Format gespeichert, das die Position und Größe der erkannten Objekte enthält.
Jedes Gruppenmitglied ist für das Labeln einer bestimmten Fruchtart verantwortlich: Sushma für Bananen, Loïc für Mangos und Moise für Äpfel. Nach Abschluss des Labelvorgangs wurden alle Label Dateien überprüft, vereinheitlicht und zu einem gemeinsamen Datensatz zusammengeführt, der anschließend für das Training des YOLOv8-Modells verwendet wurde.

2.3 Aufgabenverteilung

Tabelle 1:Aufgabenverteilung

Sushma	• Konzepterstellung • Bilder labeln: Banane • Training und Optimierung • Dokumentation
Loic Tchatat	• Konzepterstellung • Bilder labeln: Mango • Training und Optimierung • Dokumentation

KAMDEM MOISE	• Konzepterstellung • Bilder labeln: Apfel • Training und Optimierung • Dokumentation • Smartphone APK Programmierung

Die verschiedenen Aufgaben werden überwiegend in Gruppenarbeit bearbeitet, wobei jedes Gruppenmitglied für das Labeln einer bestimmten Fruchtart verantwortlich ist. Anschließend werden die Ergebnisse der trainierten Modelle gemeinsam ausgewertet und mögliche Anpassungen der Trainingsparameter besprochen. Abschließend wird das trainierte Modell in einer Android-APK integriert, um die Erkennung der Früchte direkt auf mobilen Geräten zu ermöglichen.

2.4 Zeitplan

Tabelle 2: Zeitplan für die Projektphase

	24.10.-28.10.	31.10.-04.11.	07.11.-11.11.	14.11.-18.11.	21.11.-25.11.	28.11.-02.12.	05.12.-09.12.	12.12.-09.12.	19.12.-23.12.	26.12.-30.12.	02.01.-06.01.	09.01.-13.01.	16.01.-20.01.
Konzepterstellung	X	X	X										
Dokumentation	X	X	X	X	X	X	X	X	X	X	X	X	X
Datensatz-Suche	X	X											
Bilder Labeln		X	X	X	X	X							
Training				X	X	X	X	X	X	X	X		
Modelle erstellen						X	X	X					
Modelle testen								X	X	X	X		
Modelle optimieren									X	X	X	X	
Endergebnis fertig											X	X	
Korrektur und Abgabe												X	X

2.4. Klassen/Obstsorten

Tabelle 3:Beispiele für mögliche Klassen

Sorte	**Label**
Apfel	1
Banane	2
Mango	3

3 Projekt Durchführung

3.1 Konzepterstellung

3.1.1 Ausgangslage und Motivation

Im Bereich „Smart Vending“ wird zunehmend erwartet, dass Automaten nicht nur Produkte ausgeben, sondern auch zusätzliche Informationen bereitstellen, die den Nutzenden bei der Auswahl helfen. Besonders bei frischen Lebensmitteln wie Obst kann eine schnelle, verständliche Rückmeldung sinnvoll sein: Welche Frucht wurde erkannt und welche Kalorieninformation ist damit ungefähr verbunden? Diese Information soll ohne komplizierte Bedienung verfügbar sein und direkt am Automaten ablesbar sein.

Das Projekt SmartFruitVend setzt genau an diesem Punkt an. Ziel ist die Umsetzung einer Fruit Recognition (Fruchterkennung), das Obst im Erfassungsbereich eines Automaten automatisch erkennt und das Ergebnis dem Nutzenden anzeigt. Als zusätzlicher Mehrwert wird nicht nur die erkannte Obstsorte ausgegeben, sondern auch eine Kalorienanzeige pro Stück. Wenn mehrere Früchte gleichzeitig erfasst werden, wird daraus zusätzlich eine Gesamtkalorienanzeige gebildet. Zusätzlich wurde das trainierte Modell in eine Android-APK integriert, um die Erkennung auf einem mobilen Gerät in Echtzeit zu demonstrieren. Die App dient dabei als praktische Test- und Visualisierungsoberfläche (Live-Kamerabild mit Bounding Boxes und Kalorien-Overlay) und ergänzt damit das Gesamtsystem, ohne den Einsatz im Automatenkontext zu ersetzen.

3.1.2 Zielsetzung und betrachtete Klassen

Für SmartFruitVend wurden drei Obstklassen ausgewählt:

- Apfel
- Banane
- Mango

Das System soll diese Früchte in Kamerabildern zuverlässig erkennen. Wichtig ist dabei, dass der Automat nicht nur „eine“ Frucht klassifiziert, sondern auch Situationen unterstützt, in denen mehrere Früchte gleichzeitig im Scanbereich liegen. Daraus ergeben sich zwei zentrale Anforderungen:

1. Die Lösung muss mehrere Objekte in einem Bild erkennen können.
2. Die Lösung muss die Anzahl pro Klasse bestimmen, damit eine Gesamtkalorienberechnung möglich ist.

Im Konzept wird daher nicht nur die reine Klassifikation betrachtet, sondern ein Ablauf, der Erkennung, Zählung, Kalorienlogik und Anzeige als zusammenhängendes System beschreibt.

3.1.3 Nutzungskonzept (Automat) und Erweiterung durch Android-App

In der Konzeptphase wurden zwei Nutzungsszenarien beschrieben, da SmartFruitVend nicht nur am Automaten funktionieren soll, sondern auch über eine mobile Scan-Lösung.

A) Bedienablauf am Automaten

Die Nutzenden legen eine oder mehrere Früchte in einen definierten Erfassungsbereich. Eine fest installierte Kamera erfasst den Bereich und nimmt ein Bild auf. Anschließend wird die Fruit Recognition ausgeführt und das Ergebnis am Display angezeigt. Die Ausgabe ist so konzipiert, dass sie schnell erfasst werden kann und keine zusätzliche Bedienlogik benötigt.

Typische Anzeigeelemente sind:

- erkannte Obstsorte(n) (Apfel, Banane, Mango)
- Anzahl pro Obstsorte (z. B. „2× Banane“)
- Kalorien pro Stück (je Klasse)
- Gesamtkalorien als Summe

B) Scan über die Android-App (APK)

Zusätzlich wird eine Android-App verwendet, die eine Scan-Funktion über die Smartphone-Kamera bereitstellt. Nach dem Scazn zeigt die App das Erkennungsergebnis und die Kalorieninformation an. Die App ist im Konzept Prototyp als Erweiterung gedacht, um die Funktion unabhängig vom Automaten demonstrieren und mobil nutzen zu können. Gleichzeitig bleibt die Ergebnisdarstellung identisch zur Automatenanzeige, damit das System einheitlich wirkt.

3.1.4 Systemkonzept und Ablaufkette (vom Bild zur Kalorienanzeige)

Damit die Umsetzung strukturiert erfolgen kann, wurde das System konzeptionell in mehrere Schritte unterteilt. Jeder Schritt erfüllt eine klar definierte Aufgabe und kann später separat getestet werden.

1) Bilderfassung

Ausgangspunkt ist ein Kamerabild des Scanbereichs. Im Automaten ist die Kameraposition fest, wodurch Aufnahmebedingungen stabil sind. In der App ist die Aufnahme variabler, weshalb eine robuste Erkennung erforderlich ist.

2) Fruit Recognition (Objekterkennung)

Für die Erkennung wird Ultralytics YOLOv8 genutzt. Der Vorteil dieses Ansatzes besteht darin, dass nicht nur eine Gesamtklasse pro Bild bestimmt wird, sondern mehrere Objekte gleichzeitig erkannt und lokalisiert werden können. Das ist entscheidend für den Automatenfall, in dem mehrere Früchte gleichzeitig im Bild liegen können. Als Ergebnis liefert das System erkannte Objekte inklusive Klasse, Position und Erkennungssicherheit.

3) Zählung und Ergebnisaufbereitung

Die erkannten Objekte werden pro Klasse zusammengefasst, sodass eine verständliche Ergebnisstruktur entsteht, z. B. „Apfel: 1 Stück“ oder „Banane: 2 Stück“. Diese Struktur ist die Basis für eine klare Anzeige am Automaten und in der App.

4) Kalorienlogik (pro Stück + Gesamtsumme)

Im Konzept ist die Kalorienanzeige als „pro Stück“-Ausgabe definiert. Für jede Klasse existiert ein hinterlegter Kalorienwert pro Stück. Aus der Stückzahl pro Klasse werden Teilwerte berechnet und anschließend zu einer Gesamtsumme addiert. Dadurch kann das System sowohl Einzelwerte (pro Frucht) als auch die Gesamtkalorien direkt darstellen.

5) Ausgabe am Display / in der App

Die Ausgabe erfolgt nach erfolgreicher Erkennung als Kalorieninformation zur jeweiligen Obstsorte. Grundlage dafür ist, die vom Modell gelieferte Klassen-ID, der ein fester Kalorienwert zugeordnet wird. Die Anzeige wird nur für ausreichend sichere Detektionen genutzt (Konfidenzschwelle), um Fehlanzeigen zu vermeiden.

Damit ergibt sich eine klare Ablaufkette:

Bild → Erkennung → Stückzahl → Kalorien pro Stück → Gesamtkalorien → Anzeige

3.2 Implementierung

3.2.1 Datensatz und Labeling

Um das Training des Objekterkennungsmodells zu ermöglichen, wurde ein eigener Datensatz erstellt, der die drei Obstsorten Apfel, Banane und Mango umfasst. Ziel war es, einen möglichst realitätsnahen Datensatz zu erzeugen, der typische Einsatzbedingungen eines Smart-Vending-Systems widerspiegelt. Sämtliche Bilddaten stammen aus frei verfügbaren Online-Quellen. Dabei wurde darauf geachtet, sowohl ähnliche Hintergründe als auch unterschiedliche Beleuchtungssituationen, Kamerawinkel und variierende Objektgrößen abzudecken, um eine ausreichende Vielfalt innerhalb des Datensatzes sicherzustellen.

Für die Annotation der Bilddaten wurde zunächst eine geeignete Arbeitsumgebung mithilfe der Anaconda Prompt eingerichtet, über welche das Tool Label Studio importiert und verwendet wurde (Label-Studio, n.d.). Die Annotationen erfolgten manuell, wobei jedes sichtbare Obstobjekt im Bild mit einer rechteckigen Bounding Box markiert und eindeutig einer der definierten Klassen zugeordnet wurde. Jedes Gruppenmitglied war für das Labeling einer eigenen Klasse verantwortlich, wodurch eine parallele Bearbeitung des Datensatzes ermöglicht wurde. Um eine einheitliche Struktur sicherzustellen, wurden die Klassennamen und -zuordnungen vorab festgelegt und konsequent eingehalten.

Die erstellten Labels wurden im „YOLOv8-Format+Image“ gespeichert, das für jedes Objekt die normierten Koordinaten der Bounding Box sowie die zugehörige Klassen-ID enthält. Dieses Format ist speziell auf die Anforderungen des eingesetzten YOLOv8-Modells abgestimmt und erlaubt eine effiziente Weiterverarbeitung im Trainingsprozess.

Zur Sicherstellung einer hohen Annotierungsqualität wurde besonderer Wert auf konsistente und präzise Bounding-Boxes gelegt. Dabei wurde darauf geachtet, dass die markierten Boxen ausschließlich des relevanten Objekts umfassen und alle im Bild sichtbaren Früchte vollständig annotiert sind. Nach Abschluss des Labeling-Prozesses wurden die Annotationen stichprobenartig überprüft und bei Bedarf korrigiert, um Fehlklassifikationen oder ungenaue Markierungen zu vermeiden.

Der finale Datensatz umfasst insgesamt etwa 320 annotierte Objekt und wurde zufällig in Trainings- und Validierungsdaten aufgeteilt. Diese Aufteilung ist essenziell, da sie eine Bewertung der Modellleistung sowohl auf bekannten als auch auf zuvor ungesehenen Bildern ermöglicht. Die Qualität und Konsistenz des Datensatzes bilden die Grundlage, für die im späteren Verlauf erzielten, stabilen Trainingsergebnisse sowie die hohe Erkennungsleistung des Modells.

3.2.2 Modelltraining

Das Modelltraining erfolgte mit einem Convolutional Neural Network (CNN), das auf die Verarbeitung von Bilddaten ausgelegt ist. Bevor dem YOLOv8-Modelltraining anzufangen, haben wir alle den gelabelten Bildern in einem gleichen Ordner (dateset_final) hinzugefügt. Ziel dieser Vorbereitung war es, eine einheitliche und strukturierte Datenbasis für das Training sicherzustellen, wie es in Abbildung 2 dargestellt ist.

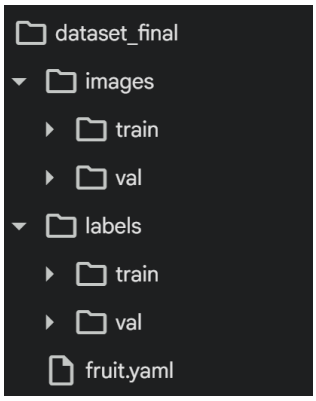

Abbildung 2: Struktur des Datasets

Um das Training durchzuführen, wurde ein Python Code implementiert, die die Daten zufällig in Trainings und Validierungsdaten aufteilt. 80% der Daten für das Training und 20% für die Validierung. Wie zeigt die Abbildung 3, wird die Bibliothek ,,random'' benutzt, um sicherzustellen, dass die Daten genau zufällig aufgeteilt werden.

```python
from pathlib import Path
import random
import shutil

ROOT = Path('c:/Users/kamde/OneDrive/Dokumente/bildverarbeitung/dataset')
IMAGE_DIRS = [
    ROOT / 'data_M' / 'images',
    ROOT / 'Label_sussus' / 'images',
    ROOT / 'Data_Loic' / 'images',
]
LABELS_DIR = ROOT / 'labels_final'
DEST = ROOT / 'dataset_final'
RATIO_TRAIN = 0.8
```

Abbildung 3: Code für Aufteilung der Daten

Aufgrund sowohl der Unverfügbarkeit von Linux auf unsere Rechner als auch mehr Flexibilität, haben wir die Entscheidung getroffen, Google-Colab zu verwenden, um das Modelltraining durchzuführen. Diese Plattform bietet kostenlosen Zugriff auf GPU-beschleunigte Rechenressourcen beispielweise Tesla 4 (T4), wodurch die Trainingszeit im Vergleich zur lokalen CPU-Ausführung deutlich reduziert werden konnte. Zudem ermöglicht Google Colab eine einfache

Einrichtung der Trainingsumgebung sowie eine gute Reproduzierbarkeit der Experimente. Durch die Integration mit Google Drive konnte der Datensatz zentral verwaltet und von allen Gruppenmitgliedern genutzt werden, was insbesondere für die Teamarbeit von Vorteil ist. Darüber hinaus liefert diese Plattform viele andere Vorteile bzw. gute Visualisierung der Trainingsergebnisse, schnelle Hyperparameternänderung und Verwendung von Python.

In Folgenden werden die wichtigsten Codelinien dargestellt, die im Rahmen des Trainings verwendet wurden.

Vor dem Start ist es erforderlich, die Bibliothek Ultralytics (siehe Abbildung 4) in Google-Colab zu installieren. Dafür wurde der direkte unter angezeigten Befehl verwendet. Diese Bibliothek ist sehr wichtig für das Modelltraining in Google-Colab.

```
!pip install ultralytics
```

Abbildung 4:Befehl, um die Ultralytics Bibliothek zu installieren

Ein weiterer wesentlicher Schritt ist die Erstellung der YAML-Datei für die Ultralytics-Trainingskonfiguration. Die Abbildung 5 veranschaulicht die Struktur des Codes. Des Weiteren enthält diese Datei Informationen zum Speicherort der Trainings- und Validierungsdaten sowie zur Klassendefinition des Modells.

```
%%writefile dataset_final/fruit.yaml
path: /content/dataset_final
train: images/train
val: images/val
nc: 3
names: ['Apple', 'Banana', 'Mango']
```

Abbildung 5: Erstellung YAML-Datei

In Abbildung 6 wird dargestellt den Befehl, der eigentlich das Modelltraining startet. Für das Modelltraining wurde die Anzahl der Epochen auf 50 festgelegt. Dieser Wert stellt einen sinnvollen Kompromiss zwischen ausreichender Lernzeit und der Vermeidung von Overfitting dar. Die während des Trainings beobachteten Loss-Verläufe zeigten eine stabile Konvergenz innerhalb dieses Bereichs. Die Eingabebildgröße wurde auf 640×640 Pixel gesetzt, da diese Auflösung den Standardvorgaben von YOLOv8 entspricht und einen guten Ausgleich zwischen Erkennungsgenauigkeit und Rechenaufwand bietet. Zudem ist diese Bildgröße besonders geeignet für Echtzeitanwendungen und eine spätere Nutzung auf mobilen Endgeräten.

```
from ultralytics import YOLO

model = YOLO('yolov8n.pt')  # kleines Modell
model.train(data='dataset_final/fruit.yaml', epochs=50, imgsz=640)
```

Abbildung 6: Befehl zum Starten des Trainings

Mit Abbildung 7 sind die Ergebnisse zu erkennen. Die Abbildung zeigt die Ergebnisse des Modells für die verschiedenen Obstsorten Appel, Banane und Mango. Es ist zu berücksichtigen:

- **Box (P):** Dies ist die Präzision der Vorhersagen für das Modell für die Bounding-Boxen

$$Precision = \frac{true\ positive}{true\ positive\ + false\ positive}$$

- **R:** Zeigt auf den Recall hin und wird bestimmt wie folgt:

$$Recall = \frac{true\ positive}{true\ positive + false\ negative}$$

- **mAP50 (Mean Average Precision @ 50%):** Gibt an, wie genau das Modell bei einer niedrigen Schwelle arbeitet. Das Modell hat hier für alle Klassen eine Genauigkeit von 98,2% erreicht.
- **mAP50-95**: Dies ist ein strengerer Maßstab, der die Genauigkeit bei verschiedenen Schwellen bewertet. Der Wert für alle Klassen beträgt 72,2%.
- **Individuelle Leistung der Klassen:**
    - Appel: mAP50= 99,3%, mAP50-95= 63,1%
    - Banana: mAP50= 95,9%, mAP50-95= 68,2%
    - Mango: mAP50= 99,2%, mAP50-95= 85,3%

```
50 epochs completed in 0.091 hours.
Optimizer stripped from /content/runs/detect/train2/weights/last.pt, 6.2MB
Optimizer stripped from /content/runs/detect/train2/weights/best.pt, 6.2MB

Validating /content/runs/detect/train2/weights/best.pt...
Ultralytics 8.3.236 🚀 Python-3.12.12 torch-2.9.0+cu126 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,006,233 parameters, 0 gradients, 8.1 GFLOPs
               Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% ━━━━━━━━━━━━ 3/3 2.3it/s 1.3s
                 all         68        268      0.997      0.942      0.982      0.722
               Apple         24        138       0.99      0.993      0.993      0.631
              Banana         22         68          1      0.865      0.959      0.682
               Mango         22         62          1      0.969      0.992      0.853
Speed: 0.2ms preprocess, 3.5ms inference, 0.0ms loss, 5.2ms postprocess per image
Results saved to /content/runs/detect/train2
ultralytics.utils.metrics.DetMetrics object with attributes:
```

Abbildung 7: Ergebnisse nach Training des Modells

Basierend auf die verfügbaren Ergebnisse ist es zu verstehen, dass das Modell bereits in der Lage ist, die meisten Objekte gut zu erkennen, jedoch bei bestimmten Obstsorten, wie z. B. der Banane, noch Optimierungsbedarf besteht.

D er in Abbildung 8 dargestellte Code dient der Anwendung des trainierten YOLOv8-Modells auf den Validierungsdatensatz zur visuellen Bewertung der Erkennungsleistung.

```
model = YOLO('runs/detect/train2/weights/best.pt')
results = model.predict(source='dataset_final/images/val', save=True ,conf=0.25)
```

Abbildung 8: Ergebnisse nach Training des Modells

Zur qualitativen Bewertung des trainierten Modells wurde das während des Trainings beste YOLOv8-Gewicht geladen und auf den Validierungsdatensatz angewendet. Mithilfe der Predict-Funktion wurden die im Validierungsordner enthaltenen Bilder analysiert und erkannte Objekte mit Bounding Boxes und Klassenlabels visualisiert. Die Ergebnisse wurden automatisch gespeichert und dienten der visuellen Überprüfung der Erkennungsqualität sowie der Identifikation möglicher Fehlklassifikationen.

3.2.3 Android-App-Integration

Dieser Schritt bildet die entscheidende Brücke zwischen dem trainierten KI-Modell und der Benutzeroberfläche. Ziel ist eine Echtzeit-Erkennung der Früchte sowie die Anzeige der Kalorieninformationen in einer intuitiven und reaktionsschnellen Anwendung.

3.2.3.1 Export nach TensorFlow Lite

Nach dem Training des YOLOv8-Modells wurde es ins TFLite-Format konvertiert, um die Inferenz auf mobilen Geräten zu ermöglichen. Der folgende Befehl wurde verwendet, um das trainierte YOLOv8-Modell für die Android-Integration vorzubereiten

```
!yolo export model=runs/detect/train2/weights/best.pt format=tflite nms=True
```

Dieser Befehl erfüllt zwei entscheidende Funktionen:

a) Aktivierung des integrierten Non-Maximum Suppression (NMS): Durch die Verwendung der Exportoption nms=True wird der Algorithmus zur Non-Maximum Suppression direkt in das exportierte TensorFlow-Lite-Modell integriert. Die Unterdrückung redundanter Bounding Boxes erfolgt somit bereits innerhalb des neuronalen Netzes und nicht erst in der Android-Applikation. Dies bietet mehrere Vorteile für den Einsatz auf mobilen

Endgeräten. Zum einen entfällt die Notwendigkeit, einen eigenen NMS-Algorithmus in der App zu implementieren, wodurch der Umfang und die Komplexität des Anwendungscodes deutlich reduziert werden. Zum anderen liefert das Modell bereits bereinigte Erkennungsergebnisse, bei denen pro Objekt nur eine eindeutige Bounding Box ausgegeben wird. Dies führt zu einer stabileren Objekterkennung und verbessert gleichzeitig die Laufzeitperformance der Anwendung, da zusätzliche Rechenoperationen im Post-Processing vermieden werden.

b) Analyse der Eingabe- und Ausgabestruktur des Modells (siehe Abbildung 9): Beim Export des YOLOv8-Modells mit Ultralytics werden die Schnittstellen (Endpoints) des TensorFlow-Lite-Modells definiert und ausgegeben. Diese sind für die korrekte Implementierung der Inferenzpipeline in der Android-Anwendung von zentraler Bedeutung.

- Eingabestruktur: inputs_0 in der Form: (1, 640, 640, 3). Dies bedeutet, dass das neuronale Netz ein einzelnes RGB-Bild mit einer festen Auflösung von 640 × 640 Pixeln erwartet. Für die Verwendung in der Android-Anwendung müssen die von der Kamera erfassten Bilder entsprechend vorverarbeitet werden. Dazu gehören insbesondere das Skalieren auf die erforderliche Eingabegröße sowie die Normalisierung der Pixelwerte, bevor die Daten an das Modell übergeben werden.
- Ausgabestruktur: Die Ausgabe des Modells erfolgt über einen einzelnen Tensor mit folgender Form: (1, 300, 6). Das Modell gibt somit maximal 300 Erkennungen pro Bild zurück. Jede einzelne Erkennung besteht aus sechs Werten, die folgende Informationen enthalten: Position der Bounding Box (x- und y-Koordinate); Abmessungen der Bounding Box (Breite und Höhe); Konfidenzwert der Erkennung; Klassenindex des erkannten Objekts

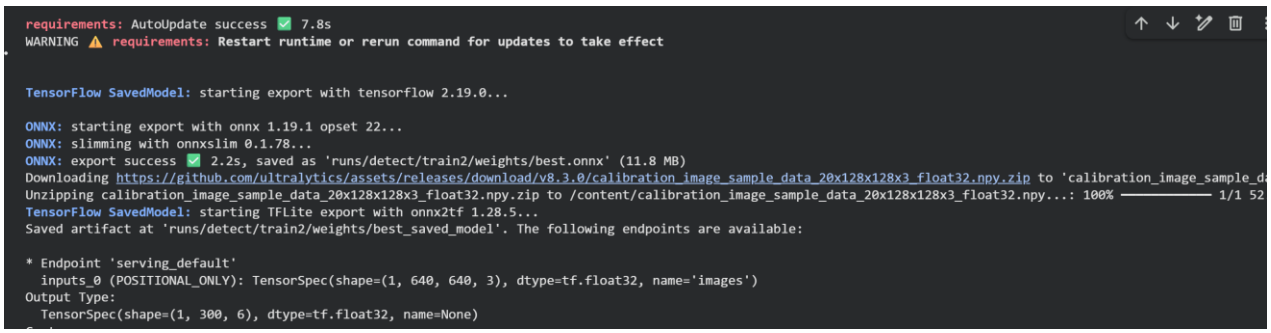

Abbildung 9: Ergebnis des Befehls

Nach dem Training und dem Speichern des besten Modells im Ordner „best_saved_model/“ werden beim Export nach TensorFlow Lite typischerweise mehrere Varianten erzeugt: „best_float32.tflite“ und „best_float16.tflite.“

3.2.3.2 Android Studio: Überblick und Konfiguration

Android Studio ist die offizielle integrierte Entwicklungsumgebung (IDE) für die Android-Plattform, entwickelt von Google. Sie basiert auf IntelliJ IDEA und bietet eine umfassende Umgebung für die Entwicklung, das Testen und die Bereitstellung von Android-Anwendungen. Die IDE integriert alle notwendigen Werkzeuge wie Code-Editor, Emulator, Debugging-Tools sowie Funktionen für die Benutzeroberflächengestaltung.

Android Studio ist die offizielle IDE zur Entwicklung von Android-Anwendungen. Android Studio unterstützt in erster Linie die Programmiersprachen Java und Kotlin. Seit 2019 gilt Kotlin als bevorzugte Programmiersprache für die Android-Programmentwicklung (wikipedia, 2026). Android Studio verwendet das flexible Build-System Gradle, welches eine modulare Projektstruktur (mehrere Module, z. B. App-Module, Bibliotheksmodule, und der Jetpack Komponenten wie CameraX) unterstützt und eine effiziente Verwaltung von Abhängigkeiten erlaubt. Für die Gestaltung der Benutzeroberfläche stehen ein visueller Layout-Editor sowie XML-Konfigurationsdateien zur Verfügung. Ein integrierter Android-Emulator ermöglicht das Testen der Anwendung auf virtuellen Geräten mit beliebigen Hardwarekonfigurationen und Betriebssystem-Versionen.

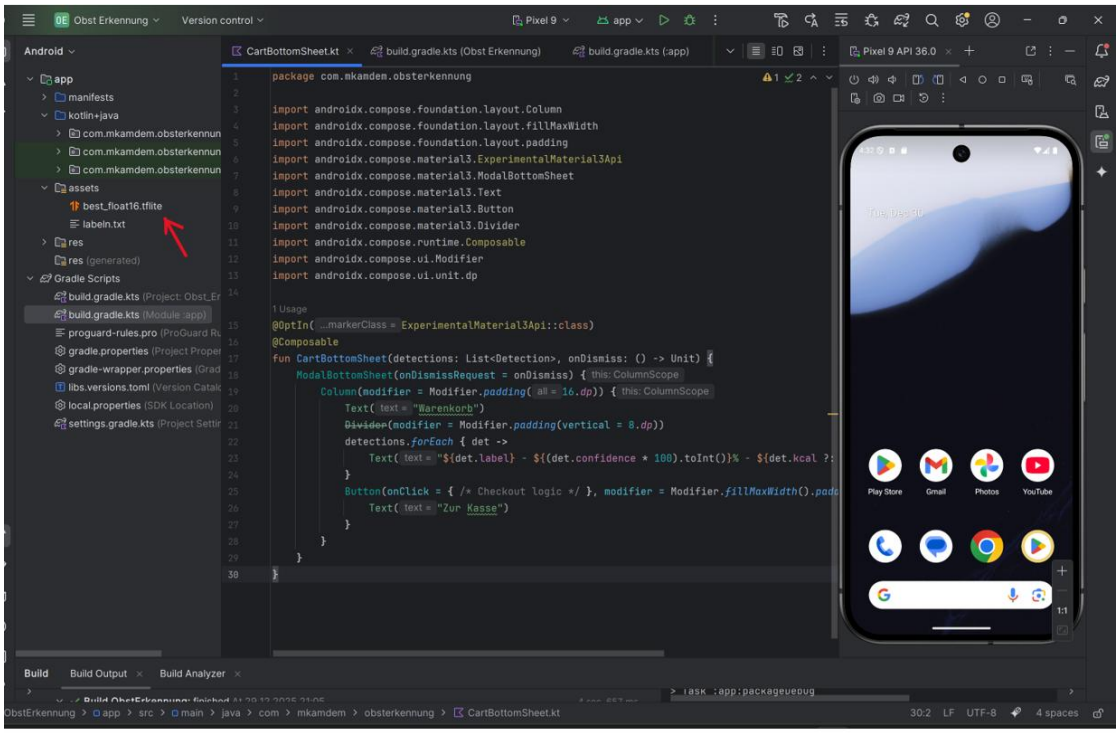

Abbildung 10: Android Studio

Für die Integration des trainierten YOLOv8-Modells in die Android-Applikation ist eine geeignete Projektstruktur sowie die korrekte Einbindung der benötigten Bibliotheken erforderlich. Wie stellt die Abbildung 10 dar, wird das exportierte TensorFlow-Lite-Modell(best_float16.tflite) im Android-Projekt im Verzeichnis „app/src/main/assets/“ abgelegt und steht der Anwendung somit zur Laufzeit zur Verfügung (surendramaran, 2025). Zusätzlich wird eine Textdatei zur Zuordnung der Klassenlabels verwendet. Diese Datei wird im selben Verzeichnis gespeichert und enthält eine eindeutige Zuordnung zwischen Klassenindex und Klassenname. Die Verwendung einer separaten Labeldatei erhöht die Flexibilität des Systems, da Änderungen an den Klassenbezeichnungen ohne erneutes Training oder Export des Modells vorgenommen werden können

```
// CameraX
implementation(libs.androidx.camera.core)
implementation(libs.androidx.camera.camera2)
implementation(libs.androidx.camera.lifecycle)
implementation(libs.androidx.camera.view)

// TensorFlow Lite
implementation(libs.tensorflow.lite)
implementation(libs.tensorflow.lite.support)
```

Abbildung 11:Bibliotheken in der Gradle-Konfiguration

Um das TensorFlow-Lite-Modell sowie die Kamerafunktionalität in der Android-Applikation nutzen zu können, müssen entsprechende Bibliotheken in der Gradle-Konfiguration des App-Moduls eingebunden werden (siehe Abbildung 11).

Zur Ausführung des trainierten YOLOv8-Modells auf dem Android-Endgerät wird der TensorFlow-Lite-Interpreter verwendet. Dieser stellt die zentrale Schnittstelle zwischen der mobilen Anwendung und dem neuronalen Netz dar und ermöglicht die Durchführung der Inferenz direkt auf dem Gerät.

```
init {
    // Load model from assets with options
    viewModelScope.launch { this: CoroutineScope
        try {
            val model = FileUtil.loadMappedFile( context = getApplication(), filePath = "best_float16.tflite")
            val options = Interpreter.Options().apply { this: Interpreter.Options
                numThreads = maxOf( a = 2, b = Runtime.getRuntime().availableProcessors() / 2)
            }
```

Abbildung 12: Ausführung des Modells(best_float16.tflite) in der APK

Wie zeigt die Abbildung 12, erfolgt die Initialisierung des TensorFlow-Lite-Modells asynchron innerhalb des viewModelScopes, um die Benutzeroberfläche nicht zu blockieren. Dieser Ansatz stellt sicher, dass die Anwendung während des Ladevorgangs reaktionsfähig bleibt, (Edge, 2025).

Beim Laden des Modells wird zusätzlich eine dynamische Konfiguration des Multi-Threadings durchgeführt. Die Anzahl der Threads wird dabei automatisch an die Leistungsfähigkeit des jeweiligen Smartphone-Prozessors angepasst. Dadurch wird die Inferenzgeschwindigkeit optimiert, ohne die Stabilität der Anwendung zu beeinträchtigen.

```
try {
    val loaded =
        try { FileUtil.loadLabels( context = getApplication(), filePath = "labeln.txt") ^launch }
        catch (_: Exception) { emptyList<String>() ^launch }
    labels = if (loaded.isNotEmpty()) loaded else listOf("Apple", "Banana", "Mango")
    Log.d( tag = "CameraVM", msg = "Labels loaded: ${labels.size}")
} catch (_: Exception) {
    labels = listOf("Apple", "Banana", "Mango")
}
```

Abbildung 13:Ausführung des Labeln-Files

Zusätzlich wurde eine Fallback-Strategie für die Klassenlabels implementiert. Diese stellt sicher, dass das System auch dann funktionsfähig bleibt, wenn die Konfigurationsdatei mit den Klassenbezeichnungen fehlt oder beschädigt ist. In diesem Fall werden Standardlabels aus der internen Liste geladen, sodass die Erkennung weiterhin korrekt durchgeführt werden kann. Der code dafür ist in Abbildung 13 dargestellt.

3.2.3.3 Kameravorschau und Bildanalyse

Die Anwendung nutzt eine Kameravorschau, die dem Nutzer das Live-Bild der Kamera anzeigt. Parallel dazu wird ein Analyse-Use-Case konfiguriert (siehe Abbildung 14), der einzelne Kameraframes für die KI-Verarbeitung bereitstellt. Um eine stabile Bildrate und geringe Latenz zu

gewährleisten, wird stets nur das aktuelle Kamerabild verarbeitet, während ältere Frames verworfen werden.

```
private fun startCamera(previewView: PreviewView, lifecycleOwner: androidx.lifecycle.LifecycleOwner, viewModel: CameraViewModel) {
    val cameraProviderFuture = ProcessCameraProvider.getInstance( context = previewView.context)
    cameraProviderFuture.addListener( listener = {
        val cameraProvider = cameraProviderFuture.get()

        val preview = Preview.Builder()
            .setTargetAspectRatio(AspectRatio.RATIO_4_3)
            .setTargetRotation(previewView.display.rotation)
            .build().also { it: Preview
            it.setSurfaceProvider(previewView.surfaceProvider)
        }
```

Abbildung 14: Initialisierung des CameraX-Preview-Use-Cases in startCamera

Die Bildanalyse erfolgt in einem separaten Hintergrundthread, um die Benutzeroberfläche nicht zu blockieren. Nach Abschluss der Inferenz werden die Erkennungsergebnisse an die UI-Ebene übergeben und dort visualisiert. Die Darstellung der Bounding Boxes erfolgt über eine separate Overlay-View, die über der Kameravorschau liegt.

```
fun onFrame(imageProxy: ImageProxy) {
    frameCount++
    try {
        val bmp = imageProxyToBitmap(imageProxy)
        val rot = imageProxy.imageInfo.rotationDegrees
        val dets = runYolo( src = bmp,  srcW = imageProxy.width,  srcH = imageProxy.height,  rotationDegrees = rot,  mirrorX = false)
        val refined = refineByClassHeuristics(dets)
        val smoothed = smoothDetections( prev = lastFrameDetections,  curr = refined)
        val gated = gateWithHysteresis( dets = smoothed)
        _detections.value = gated
        lastFrameDetections = smoothed
    } catch (e: Exception) {
        Log.w( tag = "CameraVM",  msg = "Inference failed",  tr = e)
        _detections.value = emptyList()
    } finally {
        imageProxy.close()
    }
}
```

Abbildung 15: Ausschnitt der Inferenz-Pipeline in der Android-App

Die Funktion onFrame() verarbeitet jedes von CameraX gelieferte Kamerabild, führt eine YOLO-Erkennung aus, korrigiert Rotation, glättet die Ergebnisse über mehrere Frames und stellt die finalen Bounding Boxes der Benutzeroberfläche zur Verfügung (siehe Abbildung 15).

3.2.3.4 Preprocessing (Resize & Normalisierung)

Bevor ein Kamerabild an das neuronale Netz übergeben werden kann, ist eine geeignete Vorverarbeitung erforderlich. Die von CameraX gelieferten Bilddaten liegen in einem kamerainternen Format vor und müssen zunächst in ein RGB-Bild umgewandelt werden (Docs, 2025).

```
private fun letterbox(src: Bitmap, inW: Int, inH: Int): Pair<Bitmap, LetterboxInfo> {
    val scale = minOf( a = inW / src.width.toFloat(), b = inH / src.height.toFloat())
    val newW = (src.width * scale).toInt()
    val newH = (src.height * scale).toInt()
    val dx = ((inW - newW) / 2).coerceAtLeast( minimumValue = 0)
    val dy = ((inH - newH) / 2).coerceAtLeast( minimumValue = 0)
    val dst = createBitmap( width = inW, height = inH)
    val canvas = Canvas( bitmap = dst)
    canvas.drawColor(Color.rgb( red = 114, green = 114, blue = 114))
    val scaled = src.scale( width = newW, height = newH)
    canvas.drawBitmap( bitmap = scaled, left = dx.toFloat(), top = dy.toFloat(), paint = null)
    return dst to LetterboxInfo(scale, dx, dy, inW, inH)
}
```

Abbildung 16:Einsatzt der Letterbox verfahren

Anschließend wird das Bild, auf die vom Modell erwartete, Eingabegröße von 640 × 640 Pixeln skaliert. Dabei kommt ein Letterbox-Verfahren (siehe Abbildung 16) zum Einsatz, um das ursprüngliche Seitenverhältnis beizubehalten und Verzerrungen zu vermeiden. Das Modell „best_float16.tflite“ benutzt der Single Tensor Output Format [1,300,6].

```
// Path C: Single Tensor NMS [1, N, 6]
 // Format probable: [ymin, xmin, ymax, xmax, score, class] OR [xmin, ymin, xmax, ymax, score, class]
 if (outCount == 1) {
     val outTensor = inter.getOutputTensor( outputIndex = 0)
     val shape = outTensor.shape() // [1, 300, 6]
     Log.d( tag = "CameraVM", msg = "Single tensor shape: ${shape.joinToString( separator = "x")}")

     // Relaxed check: just need last dim to be 6
     if (shape.size >= 3 && shape[shape.size - 1] == 6) {
         val numBoxes = shape[1]
         Log.d( tag = "CameraVM", msg = "Processing $numBoxes boxes from single tensor")

         val outputBuffer = java.nio.ByteBuffer.allocateDirect( capacity = 1 * numBoxes * 6 * 4) // float=4bytes
         outputBuffer.order( bo = java.nio.ByteOrder.nativeOrder())
         inter.run( input = inputBuffer, output = outputBuffer)

         outputBuffer.rewind()
         val floatBuffer = outputBuffer.asFloatBuffer()

         val dets = mutableListOf<Detection>()
         var maxScore = 0f
         var countAbove = 0
```

Abbildung 17:Verarbeitung des Single-Tensor-Outputs

Der Code, der in der Abbildung 17 dargestellt ist, erkennt, dass das Modell nur einen Ausgabetensor liefert (outputTensorCount == 1) und aktiviert die sequenzielle Auslesung des Speicherpuffers (ByteBuffer).

```
val floatBuffer = outputBuffer.asFloatBuffer()

val dets = mutableListOf<Detection>()
var maxScore = 0f
var countAbove = 0

for (i in 0 ≤ until < numBoxes) {
    // Lecture de 6 valeurs

    val v0 = floatBuffer.get() // xmin
    val v1 = floatBuffer.get() // ymin
    val v2 = floatBuffer.get() // xmax
    val v3 = floatBuffer.get() // ymax
    val score = floatBuffer.get()
    val cls = floatBuffer.get()
```

Abbildung 18: Auslesung der Tensorswerte für Bounding Boxes

Der rohe Ausgabetensor des Modells besteht aus einer Abfolge von Gleitkommazahlen. Wie zeigt der Abbildung 18, wird Jede Detektion durch sechs Werte beschrieben, die in einer festen Reihenfolge aus dem Speicherpuffer (FloatBuffer) gelesen werden. Die korrekte Lesereihenfolge ist entscheidend, um die Bounding Box an der richtigen Position darzustellen.

3.2.3.5 UI-Design und Kalorienanzeige

Dieses Kapitel beschreibt die Aufbereitung und Darstellung der vom KI-Modell gelieferten Erkennungsergebnisse in der Android-Applikation. Ziel ist es, die rohen Outputs der Objekterkennung in brauchbare, aus Sicht des Nutzers interessante Informationen zu verarbeiten und in einer brauchbar strukturierten Benutzeroberfläche anzuzeigen. Besonders Augenmerk liegt dabei auf einer intuitiven Bedienung und Darstellung, wie sie insbesondere auch für die Verwendung in der Smart-Vending-Machine nötig ist.

a) Regelbasiertes System zur Kalorienabschätzung

Zur Ermittlung des Kaloriengehalts (siehe Abbildung 19) wird ein regelbasiertes System eingesetzt, das jede erkannte Objektklasse einer durchschnittlichen Nährwertangabe zuordnet. Die Kalorienwerte basieren auf allgemein anerkannten ernährungswissenschaftlichen Referenzdaten und stellen eine realistische Näherung für einzelne Früchte dar (CalculatorChamp, 2026).

```
private fun kcalForLabel(label: String): Int? {
    return when (label) {
        "Apple" -> 52
        "Banana" -> 89
        "Mango" -> 60
        else -> null
    }
}
```

Abbildung 19:Einschätzung der kcal von jede klasse

Nach der erfolgreichen Objekterkennung wird die vom Modell ausgegebene Klassen-ID verwendet, um den entsprechenden Kalorienwert zu bestimmen.

b) Augmented-Reality-Darstellung (Grafisches Overlay)

Die Objekterkennung erfolgt im Koordinatensystem des neuronalen Netzes, welches mit einer normierten Eingabegröße von 640 × 640 Pixeln arbeitet. Da die Kameravorschau des Smartphones ein anderes Seitenverhältnis und eine höhere Auflösung besitzt, ist eine geometrische Transformation der erkannten Bounding-Boxes erforderlich. Hierzu werden folgende Schritte durchgeführt: Korrektur des Letterbox-Paddings, Skalierung der Bounding-Box-Koordinaten auf die tatsächliche Bildschirmauflösung, Anpassung an die aktuelle Geräteausrichtung und Kamerakonfiguration.

i

```
2 Usages
private fun unletterboxToOriginal(rectNormInput: RectF, lb: LetterboxInfo, srcW: Int, srcH: Int): RectF {
    // 1. Dénormaliser par rapport à l'entrée du modèle (inW, inH)
    val leftIn = rectNormInput.left * lb.inW
    val topIn = rectNormInput.top * lb.inH
    val rightIn = rectNormInput.right * lb.inW
    val bottomIn = rectNormInput.bottom * lb.inH

    // 2. Retirer le padding (dx, dy)
    val leftNoPad = leftIn - lb.dx
    val topNoPad = topIn - lb.dy
    val rightNoPad = rightIn - lb.dx
    val bottomNoPad = bottomIn - lb.dy

    // 3. Mettre à l'échelle inverse pour retrouver les pixels de l'image source redimensionnée
    //    (rappel: newW = srcW * scale, donc srcW = newW / scale)
    val leftSrcPx = leftNoPad / lb.scale
    val topSrcPx = topNoPad / lb.scale
    val rightSrcPx = rightNoPad / lb.scale
    val bottomSrcPx = bottomNoPad / lb.scale

    // 4. Normaliser par rapport à la taille source originale (srcW, srcH)
    //    Ceci permet d'avoir des coordonnées [0,1] valides pour l'affichage
    var l = leftSrcPx / srcW
    var t = topSrcPx / srcH
    var r = rightSrcPx / srcW
    var b = bottomSrcPx / srcH

    // 5. Clamper pour éviter les débordements (padding)
    l = l.coerceIn(0f, 1f)
    t = t.coerceIn(0f, 1f)
    r = r.coerceIn(0f, 1f)
    b = b.coerceIn(0f, 1f)

    return RectF( left = minOf( a = l,  b = r),  top = minOf( a = t, b),  right = maxOf( a = l,  b = r),  bottom = maxOf( a = t, b))
}
```

Abbildung 20:geometrische Transformation des Modells

Die Funktion unLetterboxToOriginal (), die in Abbildung 20 dargestellt ist, wandelt die von YOLO im normalisierten Modellraum ausgegebenen Bounding-Box-Koordinaten zurück in das Koordinatensystem des Original-Kamerabildes. Dabei werden Letterbox-Padding, Skalierungsfaktoren sowie die ursprüngliche Bildgröße berücksichtigt, um präzise und visuell korrekte Boxen in der App darzustellen.Durch diese inverse Transformation wird sichergestellt, dass die dargestellten Rahmen exakt mit den realen Früchten im Kamerabild übereinstimmen. Das Ergebnis ist eine präzise Augmented-Reality-Überlagerung, bei der jedes erkannte Objekt visuell eindeutig identifizierbar ist.

c) Aggregation und Ergebnisdarstellung

Werden mehrere identische Früchte gleichzeitig erkannt, erfolgt eine Aggregation der Ergebnisse, um die Übersichtlichkeit der Benutzeroberfläche zu gewährleisten. Anstatt jede einzelne Erkennung separat darzustellen, werden gleiche Objektklassen zusammengefasst. Mögliche Darstellungsformen sind: Anzeige der Anzahl pro Fruchtart, Berechnung und Darstellung des kumulierten Kalorienwerts. Diese Aggregation reduziert die visuelle Komplexität und ermöglicht dem Nutzer eine schnelle Erfassung der relevanten Informationen, insbesondere in zeitkritischen Anwendungsszenarien wie in einem Verkaufsautomaten.

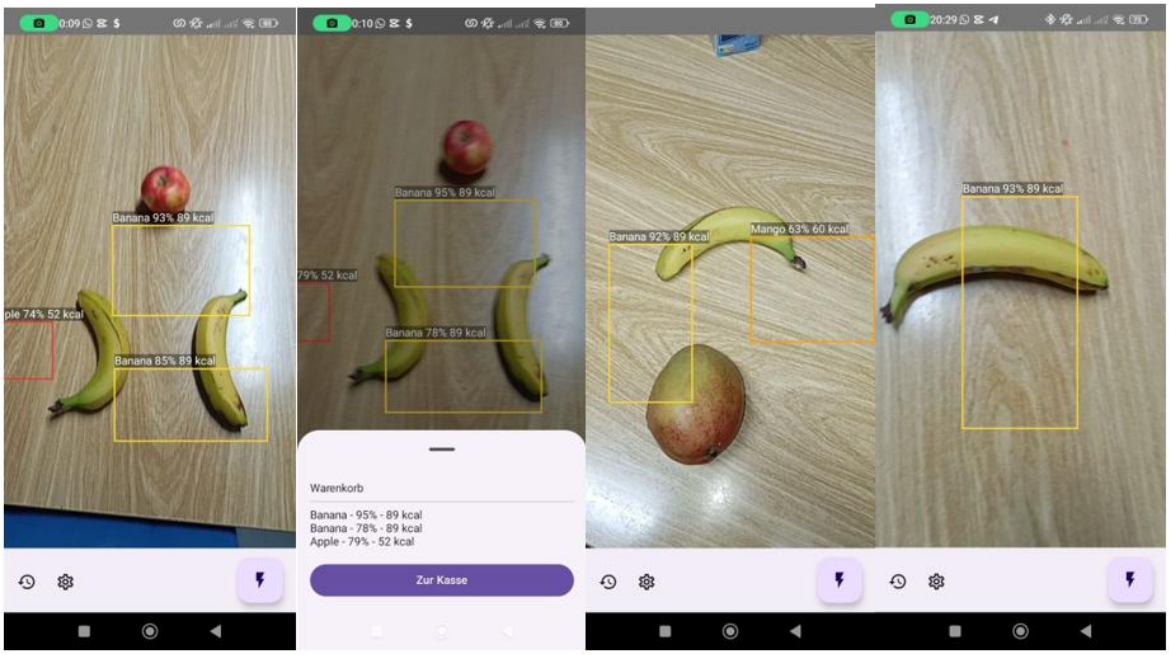

Abbildung 21:Screenschot der Apps

Die Screenshots, die in Abbildung 21 dargestellt wurde, zeigen die Benutzeroberfläche der Android-App nach erfolgreicher Integration des YOLO-Modells (TFLite) für die Echtzeit-Objekterkennung. Die App erkennt Früchte (Apfel, Banane) in der Kameravorschau und berechnet zusätzlich die Kalorien pro erkannte Frucht. Die Ergebnisse werden sowohl als Overlay auf dem Kamerabild als auch in einer Warenkorb-Ansicht dargestellt. Auffällig ist jedoch, dass die Bounding Boxes nicht exakt mit den erkannten Objekten übereinstimmen. Dies könnte auf eine ungenaue Rücktransformation der Modellkoordinaten in die Vorschauansicht hinweisen oder auf modellbedingte Faktoren wie suboptimale Hyperparameter oder eine zu geringe Modellkomplexität (z. B. die Verwendung einer sehr leichten YOLO-Variante). Diese Aspekte können die Präzision der Bounding Boxes erheblich beeinträchtigen. Die Optimierung dieser Parameter wird in Kapitel Hyperparameter-Optimierung ausführlich behandelt.

3.3 Ergebnisse

```
50 epochs completed in 0.090 hours.
Optimizer stripped from /content/runs/detect/train5/weights/last.pt, 6.2MB
Optimizer stripped from /content/runs/detect/train5/weights/best.pt, 6.2MB

Validating /content/runs/detect/train5/weights/best.pt...
Ultralytics 8.3.248 🚀 Python-3.12.12 torch-2.9.0+cu126 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,006,233 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% ━━━━━━━━━━━━ 3/3 4.1
                   all         68        268      0.997      0.942      0.982      0.722
                 Apple         24        138       0.99      0.993      0.993      0.631
                Banana         22         68          1      0.865      0.959      0.682
                 Mango         22         62          1      0.969      0.992      0.853
Speed: 0.2ms preprocess, 2.0ms inference, 0.0ms loss, 2.7ms postprocess per image
Results saved to /content/runs/detect/train5
ultralytics.utils.metrics.DetMetrics object with attributes:
```

Abbildung 22:Trainingsergebnisse final

Nach Abschluss des Trainings (50 Epochen) wurde das beste gespeicherte Modell (best.pt) auf dem Validierungsdatensatz überprüft. Für die Auswertung wurden 68 Validierungsbilder mit insgesamt 268 gelabelten Objekten (Instanzen) herangezogen. Insgesamt erreicht das Modell eine sehr hohe Präzision von P = 0,997 sowie eine Recall von R = 0,942, was auf sehr wenige Fehlalarme und eine zuverlässige Detektion der Früchte hinweist. Die Gesamtleistung beträgt mAP0.5 = 0,982 und mAP0.5-0.95 = 0,722. In der klassenweisen Betrachtung ergeben sich Apple: mAP0.5 = 0,993, Banana: mAP0.5 = 0,959 und Mango: mAP0.5 = 0,992.

Die niedrigste Wiedererkennungsrate zeigt die Klasse Banana mit R = 0,865. In der Abbildung 23 ist erkennbar, dass Bananen häufig dicht nebeneinander liegen oder sich teilweise überlappen, wodurch einzelne Früchte nicht immer als separate Instanzen erkannt werden. Zusätzlich erschweren Reflexionen und Glanz auf dem metallischen Tray sowie randnahe bzw. teilweise abgeschnittene Bananen die Abgrenzung der Objektkonturen. Dadurch werden einzelne Bananen im Validierungsdatensatz gelegentlich nicht detektiert, was die geringere Recall erklärt.

Die Abbildung 23 zeigt beispielhafte Detektionsergebnisse des trainierten Modells auf mehreren Test-/Validierungsbildern. Die erkannten Früchte werden durch Bounding Boxes markiert und zusätzlich mit der jeweiligen Klassen-ID (z. B. 0/1/2) angezeigt. Anhand der Beispiele ist erkennbar, dass das Modell sowohl Einzelobjekte (z. B. eine Banane) als auch Mehrfachobjekte (z. B. mehrere Äpfel oder mehrere Bananen in einem Bild) zuverlässig erkennen kann. Gleichzeitig werden typische Herausforderungen sichtbar, z. B. Überlappungen, Randlagen sowie Reflexionen auf dem Untergrund, die insbesondere bei Bananen zu einzelnen verpassten Detektionen beitragen können.

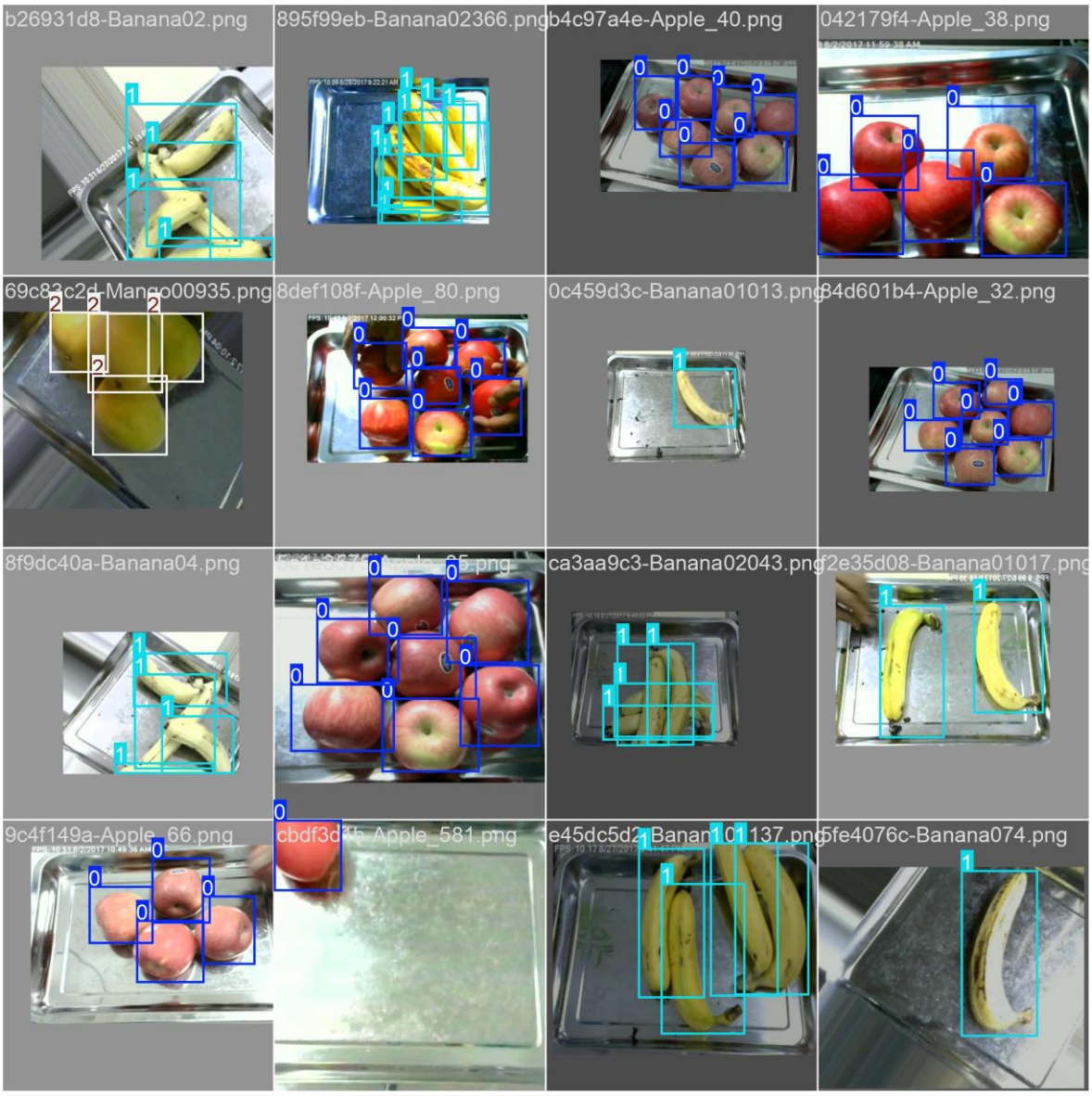

Abbildung 23:Beispielhafte Detektionen mit Bounding Boxes

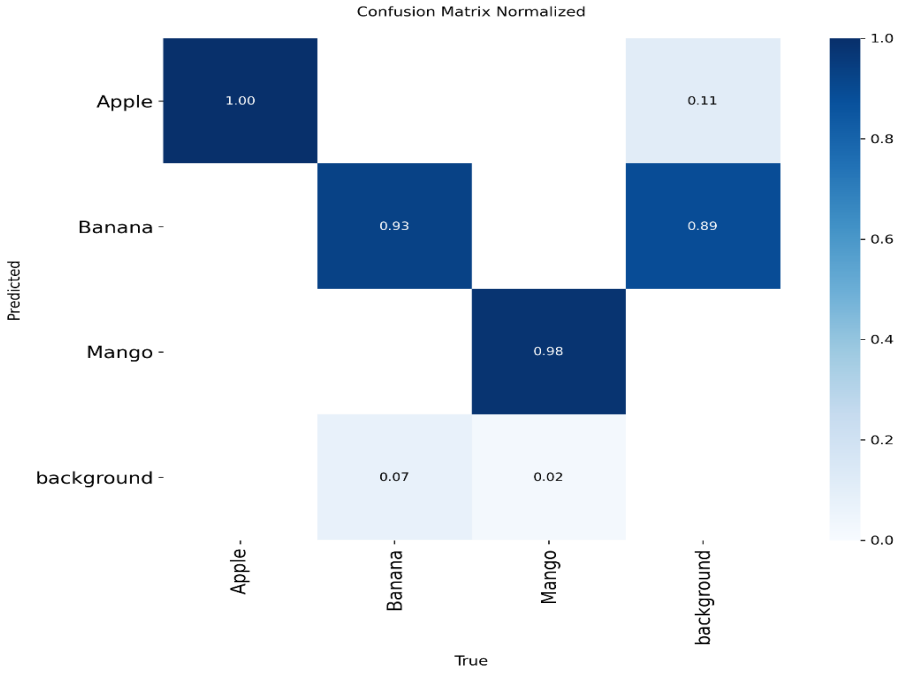

Abbildung 24: Normalisierte Confusion Matrix

Die Abbildung 24 zeigt die normalisierte Confusion Matrix des Modells für die Klassen Apple, Banana und Mango. Hohe Werte auf der Diagonalen bedeuten, dass die jeweilige Obstsorte überwiegend korrekt erkannt wird. In den Ergebnissen ist dies besonders deutlich bei Apple (1,00) und Mango (0,98), was auf eine sehr zuverlässige Klassifikation dieser beiden Klassen hinweist. Für Banana liegt der Diagonalwert bei 0,93, wodurch erkennbar ist, dass Bananen im Vergleich etwas häufiger verwechselt oder nicht eindeutig erkannt werden. Die außerhalb der Diagonale auftretenden Werte zeigen die verbleibenden Fehlzuordnungen bzw. Fälle, in denen Objekte nicht korrekt der richtigen Klasse zugeordnet werden konnten. Auch wenn bereits eine hohe Precision und ein hoher Recall bei allen Klassen vorhanden ist, könnte hier durch zusätzliche Bilder das Ergebnis der Bananen weiter verbessert werden.

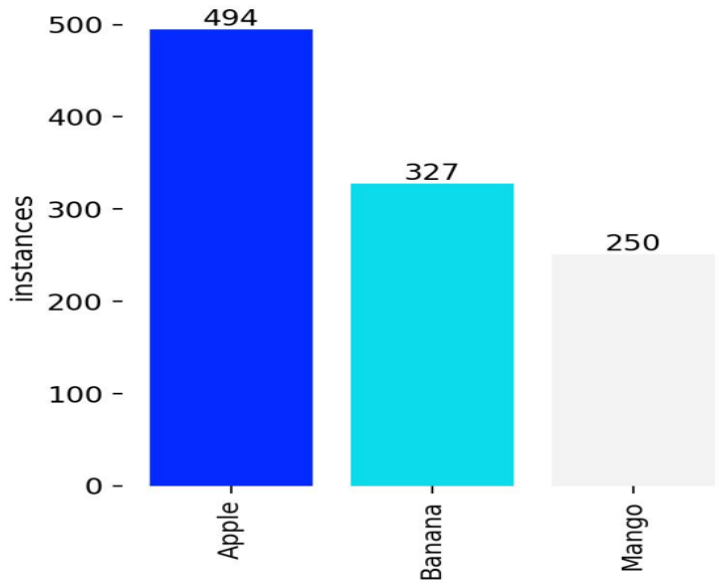

Abbildung 25: Instanzen pro Klasse

Die Abbildung 25 zeigt die Anzahl der gelabelten Instanzen pro Klasse im Datensatz. Dabei ist erkennbar, dass Apple mit 494 Instanzen am häufigsten vertreten ist, gefolgt von Banana mit 327 und Mango mit 250 Instanzen. Diese ungleichmäßige Klassenverteilung ist wichtig für die Interpretation der Modellleistung, da Klassen mit weniger Beispielen tendenziell schwieriger zu lernen sind. Gleichzeitig kann ein stärker vertretenes Objekt (hier Apple) oft stabilere Ergebnisse liefern, weil das Modell mehr Variationen dieser Klasse während des Trainings gesehen hat.

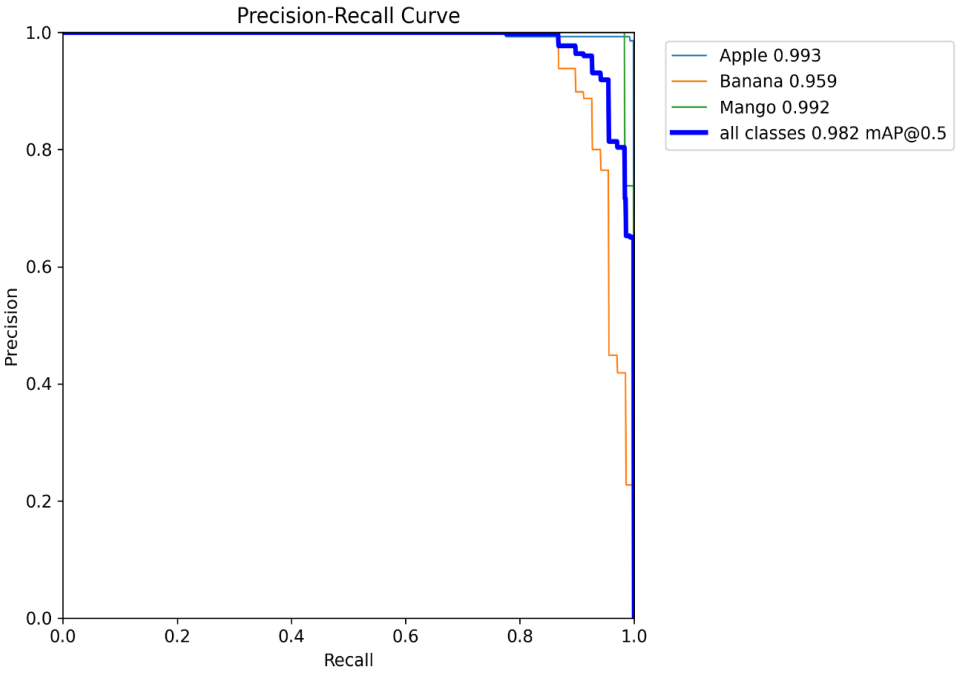

Abbildung 26: Precision-Recall-Kurve

Die Abbildung 26 zeigt die Precision-Recall-Kurve des trainierten Modells. Diese Kurve beschreibt den Zusammenhang zwischen Präzision (wie viele der erkannten Objekte korrekt sind) und Recall (wie viele der tatsächlich vorhandenen Objekte gefunden werden), wobei unterschiedliche Konfidenzschwellen betrachtet werden. Die Kurven liegen für alle Klassen überwiegend im oberen Bereich, was auf eine insgesamt sehr hohe Detektionsqualität hinweist. Aus der Legende ergeben sich die AP-Werte bei IoU = 0,5: Apple = 0,993, Banana = 0,959 und Mango = 0,992. Über alle Klassen ergibt sich mAP0.5 = 0,982, womit die Obsterkennung im gewählten Setup sehr zuverlässig arbeitet. Die Banana-Kurve fällt im Vergleich stärker ab, was zur etwas geringeren Leistung dieser Klasse passt.

3.3.1 Hyperparameter-Optimierung

Die Hyperparameter-Optimierung verfolgt das Ziel, die Gesamtleistung des Modells zu maximieren, seine Robustheit gegenüber variierenden Datenbedingungen zu erhöhen und das Risiko von Fehlklassifikationen zu reduzieren. Dabei werden jede Hyperparameter-Kombinationen identifiziert, die auf dem Validierungsdatensatz die bestmögliche Performance erzielen.

```
50 epochs completed in 0.090 hours.
Optimizer stripped from /content/runs/detect/train5/weights/last.pt, 6.2MB
Optimizer stripped from /content/runs/detect/train5/weights/best.pt, 6.2MB

Validating /content/runs/detect/train5/weights/best.pt...
Ultralytics 8.3.248 🚀 Python-3.12.12 torch-2.9.0+cu126 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,006,233 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% ━━━━━━━━━━━━ 3/3 4.1
                   all         68        268      0.997      0.942      0.982      0.722
                 Apple         24        138       0.99      0.993      0.993      0.631
                Banana         22         68          1      0.865      0.959      0.682
                 Mango         22         62          1      0.969      0.992      0.853
Speed: 0.2ms preprocess, 2.0ms inference, 0.0ms loss, 2.7ms postprocess per image
Results saved to /content/runs/detect/train5
ultralytics.utils.metrics.DetMetrics object with attributes:
```

Abbildung 27:Referenztest für Hyperparameter-Optimierung

Die in Abbildung 27 dargestellten Werte entsprechen den Standard-Hyperparametern, die während des Trainings übernommen wurden. Es wurden keine individuellen Anpassungen vorgenommen, um eine neutrale Ausgangsbasis für die Modellbewertung zu gewährleisten.

Analyse der Modelle

Es wurden zwei „Nano“ Modelle getestet: YOLOv8n und YOLO11n. Beide Varianten wurden über 50 Epochen mit imgsz=640 auf dem projektspezifischen Obst-Datensatz trainiert. Das Ergebnis ist in Abbildung 28 dargestellt

```
50 epochs completed in 0.100 hours.
Optimizer stripped from /content/runs/detect/train6/weights/last.pt, 5.5MB
Optimizer stripped from /content/runs/detect/train6/weights/best.pt, 5.5MB

Validating /content/runs/detect/train6/weights/best.pt...
Ultralytics 8.3.248 🚀 Python-3.12.12 torch-2.9.0+cu126 CUDA:0 (Tesla T4, 15095MiB)
YOLO11n summary (fused): 100 layers, 2,582,737 parameters, 0 gradients, 6.3 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%
                   all         68        268      0.997      0.946      0.978      0.726
                 Apple         24        138      0.992          1      0.994      0.624
                Banana         22         68          1      0.894      0.952      0.703
                 Mango         22         62          1      0.944      0.989       0.85
Speed: 0.2ms preprocess, 3.5ms inference, 0.0ms loss, 4.5ms postprocess per image
Results saved to /content/runs/detect/train6
```

Abbildung 28:Ergebnis des Trainings der Datasets mit yolo11n

	YOLO11n	YOLOv8n
Gesamt mAP50	0.978	0.982
Gesamt mAP50–95	0.726	0.722
Precision (P)	0.997	0.997
Recall (R)	0.946	0.942
Apple (mAP50–95)	0.624	0.631
Banana (mAP50–95)	0.703	0.682
Mango (mAP50–95)	0.850	0.853
Inference Speed (T4 GPU)	3.5 ms	2.0 ms
Postprocessing Speed	4.5 ms	2.7 ms

Tabelle 4:Vergleich des Modells

Laut der Tabelle 4 ist YOLOv8n deutlich schneller und erzielt einen etwas höheren mAP50, wohingegen YOLO11n beim strengeren mAP50–95 und beim Recall, insbesondere bei Banana geringfügig besser abschneidet. Für eine APK-Integration mit Live-Kamera stream ist das Geschwindigkeitsplus von YOLOv8n praxisrelevant

Early Stopping, um Epochs besser zu bestimmen

Der verwendete Trainingsbefehl, der in Abbildung 29 dargestellt, ist im Folgenden aufgeführt.

```
model = YOLO('yolov8n.pt')  # kleines Modell
model.train(data='dataset_final/fruit.yaml', epochs=100 ,imgsz=640, seed=42, patience=12 )
```

Abbildung 29:Befehl zur Ausführung der EarlyStopping

Durch die Festlegung von Patience = 12 wird das Early-Stopping-Verfahren aktiviert. Dabei wird das Training vorzeitig beendet, sofern sich die Validierungsmetrik (mAP) über einen Zeitraum von 12 aufeinanderfolgenden Epochen nicht weiter verbessert. Dieses Vorgehen trägt dazu bei, eine

Überanpassung des Modells an die Trainingsdaten zu vermeiden und gleichzeitig die Rechenzeit effizient zu reduzieren. „Seed“ steht für Zufalls-Startwert (random seed). Er sorgt dafür, dass alle Zufallsprozesse im Training gleich ablaufen, jedes Mal, wenn den Code gestartet wird.

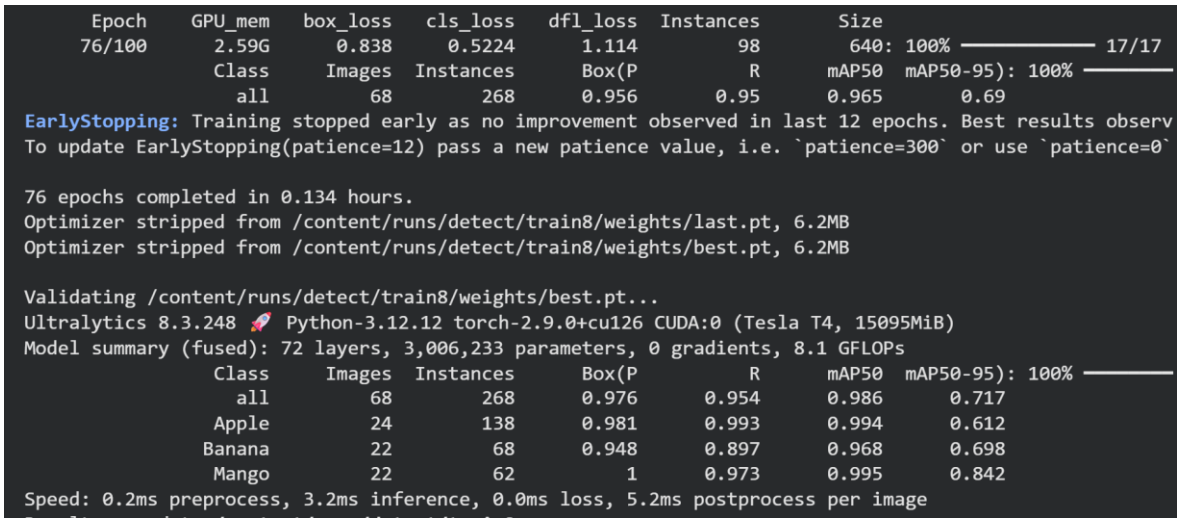

Abbildung 30: Ergebnis EarlyStopping

Zur Bestimmung des optimalen Trainingszeitpunkts, an dem das Modell die relevanten Muster der zugrunde liegenden Daten erlernt, ohne eine Überanpassung an das Rauschen der Trainingsdaten zu zeigen, wurden zwei Trainingsläufe vergleichend analysiert. Der erste Lauf diente als Referenz und wurde über 50 Epochen trainiert (siehe Abbildung 27). Der zweite Trainingslauf nutzte Early Stopping und wurde, wie es in Abbildung 30 dargestellt, automatisch nach 76 Epochen beendet. Die Ergebnisse sind in Tabelle 5 dargestellt und zeigen die Entwicklung der wichtigsten Metriken pro Klasse.

Klasse	Metrik	50 Epochs	76 Epochs	Δ (76 − 50)
Apple	mAP0.5	0.993	0.994	+0.001
	Recall	0.993	0.993	0.000
	mAP50–95	0.631	0.612	−0.019
Banana	mAP0.5	0.959	0.968	+0.009
	Recall	0.865	0.897	+0.032
	mAP50–95	0.682	0.698	+0.016
Mango	mAP0.5	0.992	0.995	+0.003
	Recall	0.969	0.973	+0.004
	mAP50–95	0.853	0.842	−0.011

Tabelle 5:Vergleich von unterschiedlicher Metrik pro Klasse

Besonders die Klasse Banane zeigte einen Zuwachs beim Recall (+0.032) und mAP50–95(+0.016), was auf eine bessere Erfassung überlappender Instanzen hindeutet. Für die produktive Nutzung wird daher ein Trainingsfenster von etwa 70 Epochen als optimal festgelegt, um eine ausgewogene Balance zwischen Genauigkeit und Effizienz zu gewährleisten.

Analyse *batch size:*

Die Batch-Größe beeinflusst, wie viele Datenpunkte gleichzeitig verarbeitet werden.

Getestet wurden Batch-Größen von 8; 32 und die Referenztest (16)

```
50 epochs completed in 0.078 hours.
Optimizer stripped from /content/runs/detect/train6/weights/last.pt, 6.2MB
Optimizer stripped from /content/runs/detect/train6/weights/best.pt, 6.2MB

Validating /content/runs/detect/train6/weights/best.pt...
Ultralytics 8.3.252 🚀 Python-3.12.12 torch-2.9.0+cu126 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,006,233 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% ━━━━━━━━━━━━ 2/2 3.6it/s 0.6s
                   all         68        268      0.986      0.961      0.977      0.732
                 Apple         24        138       0.99          1      0.991      0.629
                Banana         22         68      0.969      0.919      0.946       0.71
                 Mango         22         62          1      0.964      0.995      0.859
Speed: 0.2ms preprocess, 2.6ms inference, 0.0ms loss, 1.3ms postprocess per image
```

Abbildung 31:Ergebnis Batch Size 32

Mit einer Batch-Size von 32 erreichte das Modell (siehe Abbildung 31) eine mAP50-95 von (0.732) und eine mAP50 von (0.977), was für eine stabilere Boxenqualität Hinweis. Außerdem wurde die Trainingszeit geringer.

```
50 epochs completed in 0.104 hours.
Optimizer stripped from /content/runs/detect/train5/weights/last.pt, 6.2MB
Optimizer stripped from /content/runs/detect/train5/weights/best.pt, 6.2MB

Validating /content/runs/detect/train5/weights/best.pt...
Ultralytics 8.3.252 🚀 Python-3.12.12 torch-2.9.0+cu126 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,006,233 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% ━━━━━━━━━━━━ 5/5 1.8it/s 2.7s
                   all         68        268      0.987      0.947      0.975      0.719
                 Apple         24        138      0.993      0.995      0.994      0.621
                Banana         22         68      0.968      0.876      0.937      0.682
                 Mango         22         62          1       0.97      0.994      0.853
Speed: 0.4ms preprocess, 9.3ms inference, 0.0ms loss, 10.2ms postprocess per image
```

Abbildung 32:Ergebnis Batch Size 8

Eine Batch-Größe von 8 (siehe Abbildung 32) führte zu einer mAP50-95 von (0.719) und eine mAP50 von (0.975). Die ist deutlich langsamer als die andere

Kennzahl	Batch 8	Batch 16 (Referenz)	Batch 32
mAP0.5 (all)	0.975	0.982	0.977
mAP0.5–0.95 (all)	0.719	0.722	0.732
Recall (all)	0.947	0.942	0.961
Apple mAP0.5	0.994	0.993	0.991
Apple mAP0.5–0.95	0.621	0.631	0.629
Banana mAP0.5	0.937	0.959	0.949
Banana mAP0.5–0.95	0.682	0.682	0.711
Mango mAP0.5	0.994	0.992	0.995
Mango mAP0.5–0.95	0.853	0.853	0.859

Tabelle 6:Einfluss der Veränderung der Batch Size und Vergleich

Wie es in Tabelle 6 dargestellt ist, weist der Batch 32 den höchsten Recall-Wert (0.961) auf, was bedeutet, dass mehr wahre Objekte erkannt werden und die Anzahl der False Negatives reduziert wird. Besonders die Klasse „Banane“ profitiert deutlich von dieser Einstellung, während die Klassen „Apfel“ und „Mango“ über alle getesteten Batch-Größen hinweg weitgehend stabil bleiben. Da die Stabilität über mehrere IoU-Schwellen im Vordergrund steht, wird Batch 32 als bevorzugte Konfiguration verwendet.

Analyse der MixUp

```
Validating /content/runs/detect/train/weights/best.pt...
Ultralytics 8.3.253 🚀 Python-3.12.12 torch-2.9.0+cu126 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,006,233 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% ──────────
                   all         68        268      0.954      0.973      0.981      0.729
                 Apple         24        138       0.99          1       0.99      0.619
                Banana         22         68      0.888      0.941      0.959      0.722
                 Mango         22         62      0.984      0.977      0.994      0.848
Speed: 0.2ms preprocess, 2.7ms inference, 0.0ms loss, 2.9ms postprocess per image
```

Abbildung 33:Ergebnis Mixup 0.2

MixUp kombiniert zwei Bilder und deren Labels, um die Robustheit des Modells gegenüber überlappenden Objekten und komplexen Szenen zu erhöhen. Wie in Abbildung 33 dargestellt, profitiert insbesondere die Klasse Banane von diesem Parameter, mit einem Anstieg des mAP50–95 um +0.034 und des Recall um +0.017. Dies ist besonders relevant, da diese Klasse zuvor schwächere Ergebnisse aufwies. Die Klasse Apple bleibt weitgehend stabil, während Mango eine leichte Verbesserung beim mAP@50–95 zeigt.

Parameter	MixUp = 0.0	MixUp = 0.2	Δ (0.2 − 0.0)
mAP50 (Gesamt)	0.982	0.981	Fast gleich
mAP50–95 (Gesamt)	0.722	0.735	+0.013

Tabelle 7:Vergleich der MixUp-paramete

Wie zeigt der **Fehler! Verweisquelle konnte nicht gefunden werden.** , trägt Die Verwendung von MixUp = 0.2 zur Verbesserung der Generalisierung des Modells bei und führt zu einer signifikanten Steigerung der Gesamtleistung. Deshalb ist diese Parameter relevant für unseres Modell.

Nach der systematischen Untersuchung zentraler Hyperparameter wurde ein finaler Trainingslauf definiert, der die zuvor gewonnenen Erkenntnisse optimal vereint. Der abschließende Trainingsbefehl ist in Abbildung 34 dargestellt:

```
from ultralytics import YOLO

model = YOLO('yolov8n.pt')  # kleines Modell
model.train(data='dataset_final/fruit.yaml',
            epochs=70,imgsz=640,batch=32 ,
            mixup=0.2,
            seed=42)
```

Abbildung 34:Befehl des Modells

Durch die Kombination dieser Hyperparameter wurde ein Modelltrainingsprozess definiert, der sowohl leistungsstark als auch stabil ist und sich gleichzeitig gut für die mobile Integration eignet. Das Ergebnis den Befehl ist in Abbildung 35 dargestellt.

```
70 epochs completed in 0.119 hours.
Optimizer stripped from /content/runs/detect/train/weights/last.pt, 6.2MB
Optimizer stripped from /content/runs/detect/train/weights/best.pt, 6.2MB

Validating /content/runs/detect/train/weights/best.pt...
Ultralytics 8.3.253 🚀 Python-3.12.12 torch-2.9.0+cu126 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,006,233 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% ━━━━━━━━━━━━ 2/2 2.9it/s 0.7s
                   all         68        268      0.985      0.954      0.982       0.73
                 Apple         24        138      0.993      0.997      0.994      0.617
                Banana         22         68      0.965      0.897      0.958      0.714
                 Mango         22         62      0.997      0.968      0.995      0.859
Speed: 0.2ms preprocess, 3.1ms inference, 0.0ms loss, 1.4ms postprocess per image
```

Abbildung 35:Ergebnis der Befehl

Das vollständige Training konnte innerhalb von etwa 0,12 Stunden abgeschlossen werden. Die automatische Validierung des besten Modells („best.pt“) auf dem Validierungsdatensatz zeigt eine sehr hohe Gesamtleistung. Über alle Klassen hinweg wird eine Precision von 0.985 sowie ein Recall von 0.954 erreicht. Die mittlere Average Precision beträgt mAP0.5 = 0.982 und mAP0.5–0.95 = 0.730, was auf eine präzise und robuste Objekterkennung hinweist. In der klassenweisen Betrachtung zeigt sich, dass Apfel und Mango besonders hohe Genauigkeitswerte erzielen. Die Klasse Banana erreicht im Vergleich eine geringere Precision, profitiert jedoch deutlich von der gewählten Kombination aus Batch-Größe und MixUp, insbesondere im strengeren

Bewertungsbereich mAP0.5–0.95. Nach dem Export des trainierten YOLOv8n-Modells in das TensorFlow-Lite-Format und dessen erfolgreicher Integration in die Android-Applikation wurde die Objekterkennungsleistung unter realen Einsatzbedingungen evaluiert. Die Erkennung erfolgte dabei direkt auf dem mobilen Endgerät über einen Liven-Kamera-stream.

Die Abbildung 36 zeigt die Ergebnisse der Android-App für die Klasse Apple. In den Aufnahmen ist zu sehen, dass die Äpfel bei unterschiedlichen Blickwinkeln und verschiedenen Stückzahlen erkannt werden. Jede erkannte Frucht ist mit einer Bounding Box markiert, und direkt am Objekt wird der Kalorienwert pro Stück eingeblendet. Dadurch wird sichtbar, dass die App sowohl Einzeläpfel als auch Mehrfacherkennung (mehrere Äpfel gleichzeitig) unterstützt.

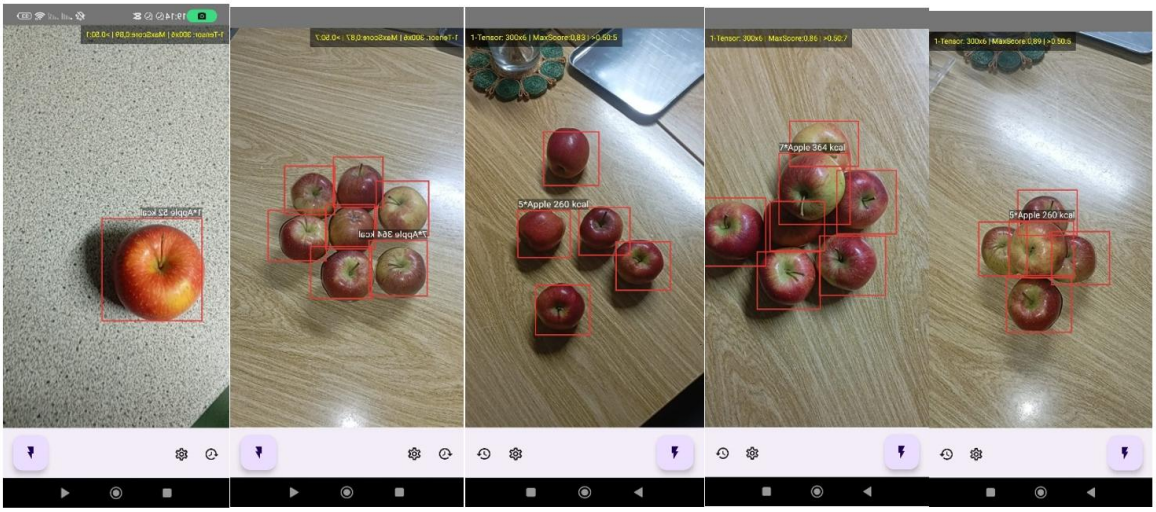

Abbildung 36:App-Erkennung Apple mit Kalorien

Abbildung 37 zeigt die Erkennung für die Klasse Banane. Hier ist erkennbar, dass Bananen sowohl einzeln als auch in Gruppen detektiert werden. Die Bounding Boxes umschließen die Bananen, und die App blendet zusätzlich den Kalorienwert pro Stück ein. In den Bildern ist außerdem zu erkennen, dass die Erkennung auch bei unterschiedlichen Abständen und schrägen Perspektiven funktioniert.

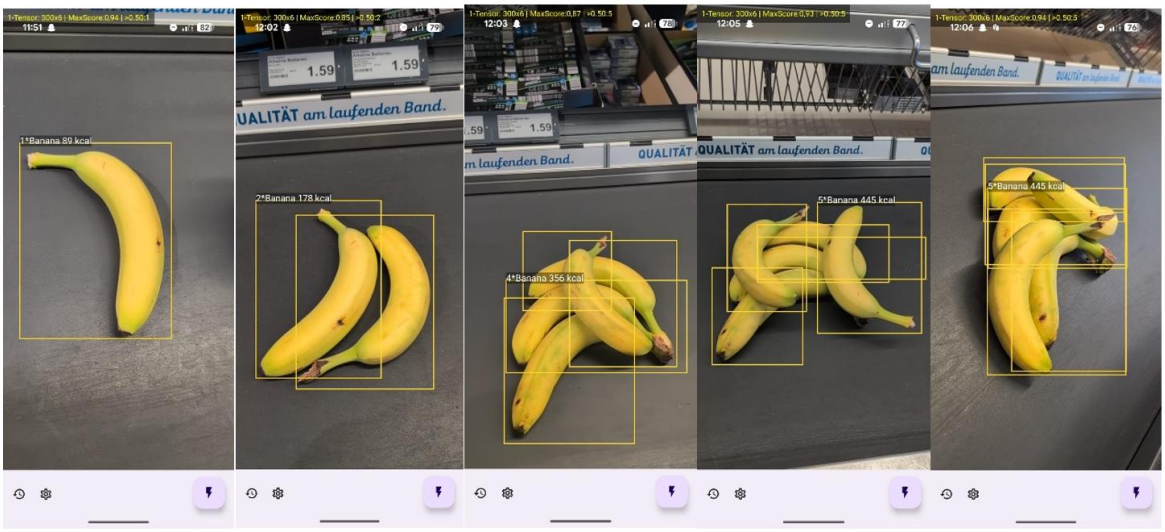

Abbildung 37:Erkennung Banana mit Kalorien

Die Abbildung 38 zeigt die Erkennung für die Klasse Mango. Die Mangos werden bei unterschiedlichen Stückzahlen zuverlässig erkannt und durch Bounding Boxes markiert. Auch hier wird der Kalorienwert pro Stück direkt in der Anzeige ausgegeben, sodass die Nutzerin bzw. der Nutzer die Information unmittelbar beim Scan erhält.

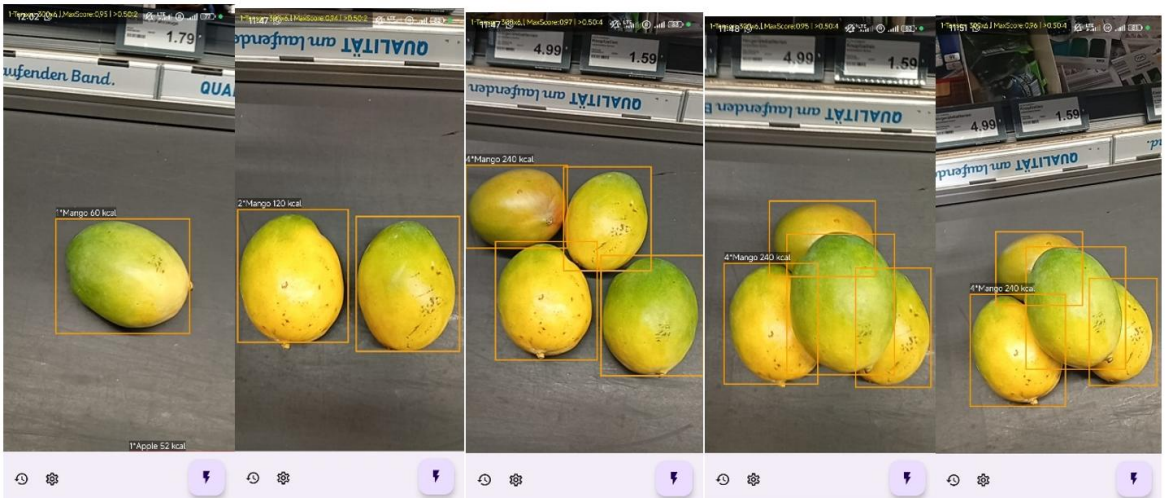

Abbildung 38:Erkennung Mango mit Kalorien

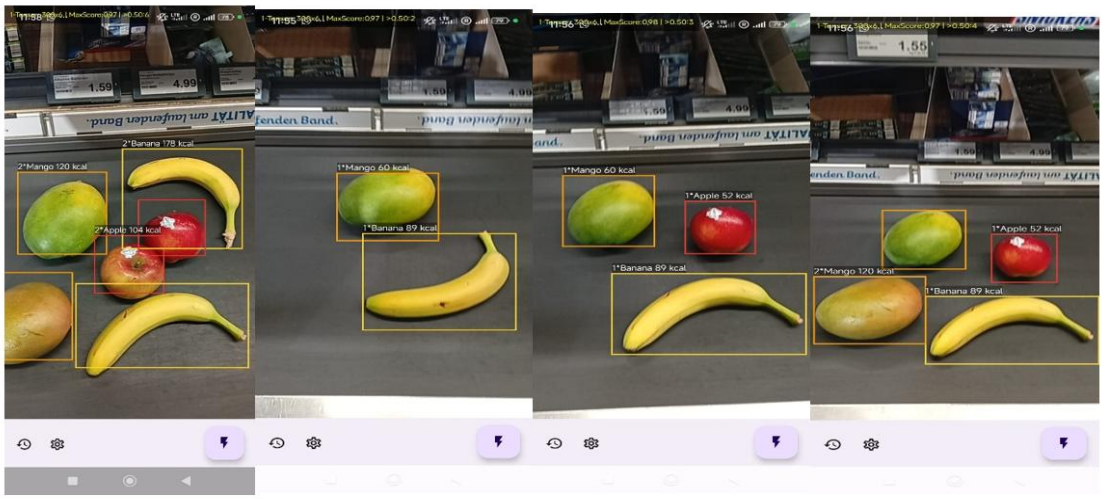

Abbildung 39: Mehrklassen-Erkennung

Die Abbildung 39 zeigt Mehrklassen-Erkennung, in denen Apple, Banane und Mango gleichzeitig erkannt werden. In den vier Beispielen ist zu sehen, dass die App die Früchte jeweils mit Bounding Boxes markiert und zusätzlich die Kalorienanzeige pro erkannte Stückzahl einblendet (z. B. *1× Apple 52 kcal*, *1× Banana 89 kcal*, *1× Mango 60 kcal* bzw. *2× Mango 120 kcal*). Außerdem wird deutlich, dass die Erkennung auch bei unterschiedlichen Blickwinkeln und verschiedenen Kombinationen von Früchten stabil funktioniert, da die Objekte in allen Bildern korrekt lokalisiert und klassifiziert werden.

Modell & Labels in app/src/main/assets/

4 Fazit und Ausblick

Das Projekt *SmartFruitVend* demonstriert erfolgreich, wie moderne Objekterkennung in Verbindung mit mobilen Endgeräten zur Erweiterung klassischer Verkaufsautomaten beitragen kann. Durch den Einsatz eines YOLOv8-basierten Detektionsmodells konnte ein System entwickelt werden, das Obst in Echtzeit erkennt, klassifiziert und gleichzeitig eine Kalorienabschätzung bereitstellt. Die Ergebnisse zeigen, dass das trainierte Modell eine hohe Präzision erreicht und insbesondere bei stabilen Aufnahmebedingungen zuverlässig arbeitet. Die Integration in eine Android-Applikation verdeutlicht zudem das Potenzial einer mobilen Nutzung, wodurch das System flexibel einsetzbar ist und über den Automatenkontext hinaus zusätzliche Anwendungsmöglichkeiten eröffnet.

Für den weiteren Projektverlauf ergeben sich mehrere sinnvolle Schritte. Erstens sollte der Datensatz insbesondere für schwierigere Szenarien erweitert werden, um die Robustheit des Modells weiter zu erhöhen. Zweitens könnte eine leistungsstärkere Modellvariante oder ein Feintuning der Hyperparameter die Genauigkeit der Objekterkennung weiter verbessern. Drittens bietet sich die Möglichkeit, die mobile Anwendung funktional zu erweitern, etwa durch eine Nährwertdatenbank, Nutzerprofile oder eine Integration in bestehende Automatensteuerungen.

5 Literaturverzeichnis

(kein Datum). Von https://www.youtube.com/watch?v=r0RspiLG260 abgerufen

CalculatorChamp. (2026). *CalculatorChamp*. Von https://calculatorchamp.com/fruit-calories-calculator abgerufen

developer. android. (2025). Von https://developer.android.com/media/camera/camerax/analyze abgerufen

Docs, U. Y. (2025). *Ultralytics YOLO Docs*. Von https://docs.ultralytics.com/modes/export/#tflite abgerufen

Edge, G. A. (2025). Von https://www.tensorflow.org/lite/android/quickstart abgerufen

Grundmann, M. (2006). *The Physics of Semiconductors. An Introduction Including Device and Nanophysics.* Berlin Heidelberg: Springer.

Label-Studio. (kein Datum). Von https://labelstud.io/learn/getting-started-with-label-studio/ abgerufen

Paul, R. (1972). *Feldeffekttransistoren – physikalische Grundlagen und Eigenschaften.* Stuttgart: Verlag Berliner Union.

Rössler, L. (13. 09 2018). Titel des Aufsatzes. *Titel der Zeitschrift*, S. 50-62.

Stierstadt, K. (2010). *Von der Mikrophysik zur Makrophysik.* Heidelberg: Springer-Verlag.

Studio, A. (2025). Von https://developer.android.com/develop/ui/compose/graphics/draw/overview abgerufen

surendramaran. (2025). *GitHub - surendramaran/YOLO: YOLOv8.* Von https://github.com/surendramaran/YOLO/tree/main/YOLOv8-Object-Detector-Android-Tflite abgerufen

wikipedia. (2026). *wikipedia*. Abgerufen am 02 2026 von https://en.wikipedia.org/wiki/Kotlin