Optimierung von Objekterkennung durch Bildaugmentation

Matthias Mildenberger[1]

Abstract Im Rahmen der vorliegenden Projektarbeit wurde ein MobileNet v1 zur Objekterkennung von blauen und roten Controllern der Nintendo Switch trainiert. Der Fokus lag darauf, die Erkennungsleistung des Modells zu verbessern, indem die Trainingsdaten durch gezielte Augmentation angereichert wurden. Das Modell wird im Weiteren auf einem Raspberry Pi 4 mit Webcam eingesetzt und die Inferenz wird mit einer Google Coral Tensor Processing Unit beschleunigt. Zusätzlich wurden je nach Controller, der erkannt wurde und wie sicher das Modell mit dabei war ein Piezo Speaker und LEDs angesteuert. Durch Augmentation in Form von Verschieben und Skalieren der Bilder konnte die Average Precision des Modells von $AP_{0.50:0.95}$ 0.314 auf $AP_{0.50:0.95}$ auf 0.366 verbessert werden. Außerdem war es dem Modell mit augmentierten Daten möglich, kleine Objekte zu erkennen, obwohl in den ursprünglichen Trainingsdaten gar keine kleinen Objekte vorhanden waren und diese erst durch Augmentation erzeugt wurden. Mit einer anderen, stärkeren Augmentationspipeline der Bilder wurde die Leistung des Modells hingegen schlechter. Am Ende der Arbeit wird grundsätzlich diskutiert, wie die Augmentation von Trainingsdaten verschiedener Bereiche gewählt werden sollte, um die Leistung eines Modells verbessern zu können.

Stichworte Objekterkennung, Augmentation, Machine Learning

1 Einleitung

Objekterkennung (object detection) ist der Teilbereich der Computer Vision, welcher sich damit beschäftigt, in Bildern verschiedene Entitäten zu lokalisieren und zu klassifizieren[3].

[1] matthias.mildenberger@haw-hamburg.de

In den letzten Jahren wuchs die Anzahl der Publikationen über object detection jährlich[16], was die zunehmende Relevanz des Themas verdeutlicht. In der Regel werden für die object detection neurale Netze benutzt, welche aus großen Trainingsdatensätzen lernen, wie Objekte aussehen, um dann über die Trainingsdaten hinaus die gelernten Objekte erkennen zu können. Ein häufiges Problem bei dem Training stellt der Mangel an qualitativ hochertigen Trainingsdaten dar. Hierbei hat es sich als hilfreich erwiesen, die vorhandenen Daten zu augmentieren[12], um damit aus den ursprünglichen Inputbildern neue, leicht unterschiedliche Bilder zu generieren. Verschiedene Transformationen der Originalbilder sind möglich. Verschieben, Verzerren, Farbcodes umwandeln, teilweises Verdecken, Drehen und noch weitere Möglichkeiten existieren, um aus den limitierten Trainingsdaten neue Daten zu erzeugen, die die Detektionsleistung eines Modelles erhöhen können. Zum Beispiel können durch Schrumpfen der Ursprungsbilder auch fehlende, kleine Objekte in den Trainingsdaten substituiert werden, sodass die Detektionsleistung auch für kleine Objekte erhöht werden kann[10].

Die vorliegende Hausarbeit stellt das Problem der oft unzureichenden Trainingsdaten in den Fokus und beschäftigt sich damit, aus einem limitierten Input Datensatz, durch Augmentation der Bilder, die Trainingsdaten an den richtigen Stellen zu ergänzen und so die Erkennungsleistung des Modells zu verbessern. Des weiteren wird das Modell im Nachhinein auf einer embedded Hardware mit beschränkter Leistung (RaspberryPi) zur Inferenz verwendet. Je nachdem, was das Modell erkennt, werden unterschiedliche Aktoren wie eine LED oder ein Piezo Speaker angetrieben.

2 Methode

Im folgenden Abschnitt wird detailreich beschrieben, welche Schritte bei der Durchführung des Projektes angewendet wurden, um die Ergebnisse genau nachvollziehen zu können und eine Reproduktion zu ermöglichen. Dabei wird zunächst der Datensatz, die

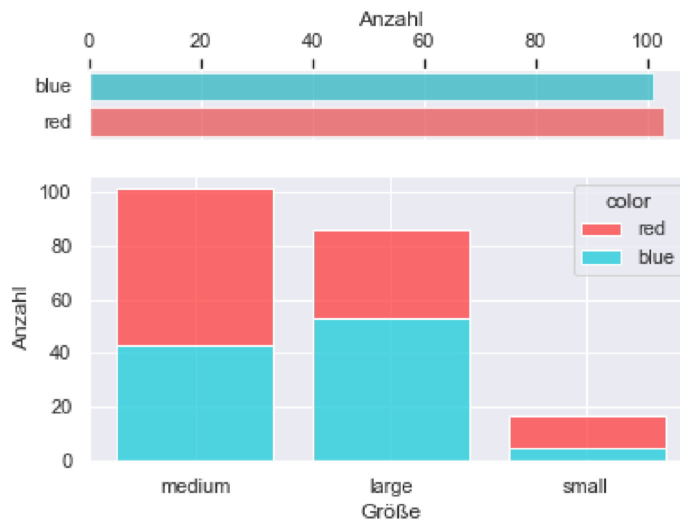
verwendete Netzarchitektur und die verwendete Hardware beschrieben. Außerdem erfolgt eine ausführliche Beschreibung der Evaluationsparameter.

2.1 Datensatz

Als Objekte, die in dieser Arbeit erkannt werden sollen, dienen Controller der Nintendo Switch. Es gibt zwei verschiedene Controller: einen roten und einen blauen. Um eine hohe Vergleichbarkeit der Trainingsdaten mit dem später im Einsatz benutzten Videostream der Kamera des RaspberryPi zu erlangen, wurden die Fotos auch mit der RaspberryPi Kamera erstellt. Dazu wurde ein Skript geschrieben, mit dem auf Knopfdruck auf einen Schalter, der am RaspberryPi angeschlossen war, ein Video mit der Kamera des RaspberryPi erstellt wurde. Zusätzlich hängt so die Erstellung der Daten nicht mehr von einem stationären Stromanschluss und Tastatur-/Mauseingaben ab. Auf diese Weise konnten frei vom Standort mit dem RaspberryPi und einer Powerbank die Videos erstellt werden. Aus diesen Videos wurden im Nachhinein einzelne Frames für den Trainingsdatensatz ausgewählt. Mit der Software LabelImg[14] wurden in den Bildern Boxen (bounding boxes) mit dem entsprechenden Label um die Controller eingezeichnet. Die Bilder mit ihren bounding boxes wurden dann in ein für Tensorflow passendes Format (tfrecords) umgewandelt, um damit ein object detection Modell trainieren zu können. Als Grundlage für die Erstellung der tfrecoords diente ein Skript, das von dem Autor der Dokumentation für die tensorflow object detection API zur Verfügung gestellt wurde[15]. In Abbildung 2.1 ist exemplarisch eines der gelabelten Fotos mit der zugehörigen bounding box zu sehen. Insgesamt sind in dem ursprünglichen Datensatz 589 Fotos mit insgesamt 608 Boxen (in manchen Bildern sind beide Controller zu sehen). Diese wurden aufgeteilt in einen Test- und einen Trainingsdatensatz. In dem Trainingsdatensatz befinden sich 404 Fotos (404 Boxen) und in dem Testdatensatz befinden sich 185 Fotos (204 Boxen). Da in der späteren Evaluation auch untersucht wird, inwiefern Objekte verschiedener Größen erkannt werden, sind diese in drei verschiedene Klassen bezügliche ihrer Größe unterteilt (small: $X < 32^2$ Pixel, medium: $32^2 <= X > 96^2$ Pixel, large: $X >= 96^2$ Pixel). Die genaue Aufschlüsselung der verschiedenen Größen mit den unterschiedlichen Objekten kann Abbildung 2.2 entnommen werden. In der GRafik wird für die verschiedenen Größen der Bilder die jeweilige Anzahl der blauen und roten Controller dargestellt, sowie in den Balken oben die Gesamtzahl der vorhandenen blauen und roten Controller.

Abbildung 2.1: Beispielfoto

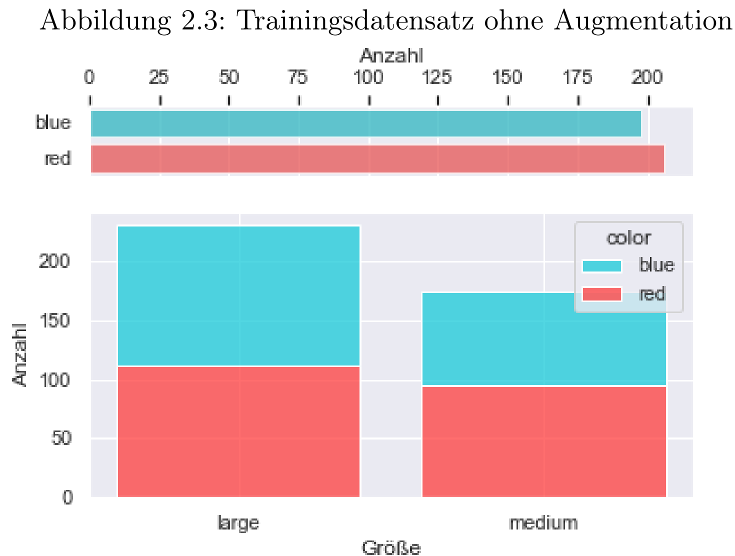
Abbildung 2.2: Testdatensatz



In dem Test- und dem Trainingsdatensatz sind die Bilder von blauen und roten Controllern in etwa gleich verteilt. Im Testdatensatz gibt es auch ähnlich viele mittelgroße und große Controller (∼100), aber nur vergleichsweise wenige kleine Controller (17). In dem Testdatensatz sind neben den, mit der Kamera des RaspberryPi aufgenommenen Fotos, auch noch Fotos enthalten, welche mit einer Smartphonekamera aufgenommen worden

sind. Das Ziel war es, einen möglichst heterogeneren Testdatensatz zu bekommen, welcher Unterschiede im Seitenformat, der Belichtung und verschiedener Hintergründe beinhaltet, um verschiedene Modelle anhand dieses allgemeinen Testdatensatzes zu validieren.

Abbildung 2.3: Trainingsdatensatz ohne Augmentation



Im Trainingsdatensatz, dessen genaue Bildaufteilung in Abbildung 2.3 visualisiert ist, gibt es gar keinen kleinen Controller. Dies ist kein Versehen, da das Thema der Hausarbeit ist, mit beschränkten Trainingsdaten umzugehen und diese durch etwaige Augmentation besser an die Testdaten anzupassen. Das Ziel ist es, auch Bilder zu erkennen, von denen ursprünglich keine oder nur sehr wenige Trainingsdaten vorhanden waren.

2.1.1 Bildaugmentation

Um die Qualität der Trainingsdaten zu erhöhen, wurden sie augmentiert und damit vervielfacht. Mithilfe der Python Library imgaug[9] wurden zwei verschiedene Augmentationspipelines erstellt, welche jeweils aus jedem Bild in den Trainingsdaten ca. zehn neue Bilder mit unterschiedlichen Transformationen generieren. In den zwei Pipelines gibt es insgesamt fünf verschiedene Transformationen, wobei je nach Pipeline unterschiedliche Transformationen ausgewählt werden.

Die verschiedenen möglichen Transformationen sind:

- ChangeColorTemperature: Die Temperatur der Farben wird verändert, durch Zufall werden die Farben etwas wärmer (stärkere Rot/Gelb Töne) oder kälter (stärkere Blau/Grün Töne).
- MultiplyBrightness: Die Helligkeit der Bilder wird verändert (entweder heller oder dunkler).
- CoarseSaltAndPepper: Es werden zufällige Artefakte im Bild erschaffen, die das Bild teilweise überdecken.
- AffineScale: Das Bild wird kleiner/größer skaliert.
- AffineTranslate: Das Bild wird in eine zufällige Richtung verschoben.

Pipeline starke Augmentation

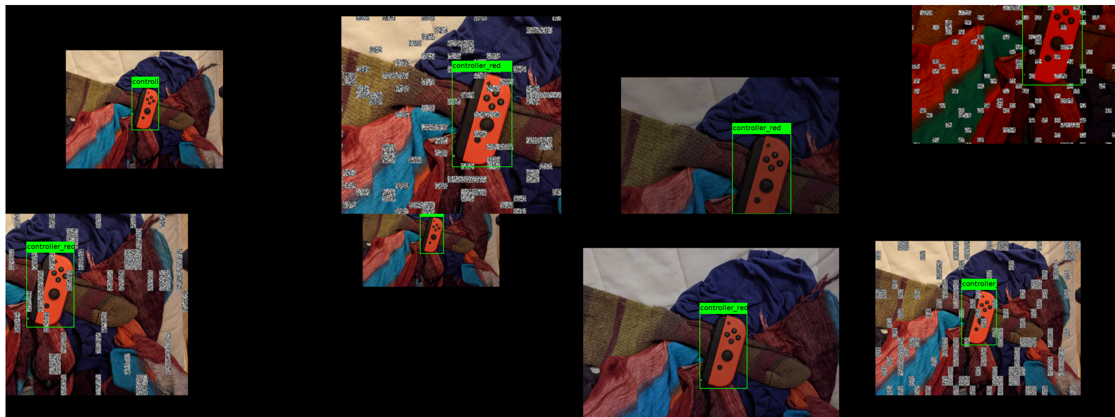
Zunächst wurde eine Augmentationspipeline erstellt, welche die Bilder stark verändert. Die Pipeline in Python ist wie folgt definiert:

Code Listing 2.1: Pipeline mit starker Augmentation

```
1 from imgaug import augmenters as iaa
2 pipeline_stark = iaa.SomeOf((2,None),[
3     iaa.ChangeColorTemperature((1100, 15000)),
4     iaa.MultiplyBrightness((0.5, 1.8)),
5     iaa.CoarseSaltAndPepper((0.05,0.2), size_percent=(0.002, 0.1)),
6     iaa.Affine(scale=(.3,1.3)),
7     iaa.Affine(translate_percent={"x": (-0.4, 0.4), "y": (-0.4, 0.4)}),
8 ])
```

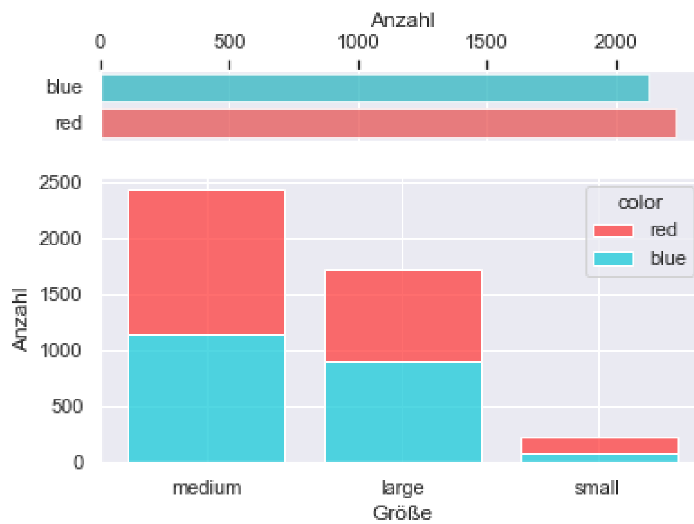
Nach dem Import der Library in Zeile eins wird eine Pipeline des Typen `SomeOf` erstellt, in denen alle fünf vorher beschriebenen Transformationen enthalten sind. Mit `SomeOf((2,None)` werden zufällig zwischen zwei und allen fünf Transformationen mit jeweils zufälliger Ausprägung (über die Argumente der einzelnen Transformationen angegeben) ausgewählt. Beispielsweise würde die Transformation `iaa.MultiplyBrightness((0.5,1.8))` die Helligkeit des Input Bilds mit einem zufälligen Wert zwischen 0.5 und 1.8 multiplizieren. Aus jedem Bild in den Trainingsdaten wurden mit dieser Pipeline nun zehn verschiedene Bilder erstellt. Beispielhaft werden in der Abbildung 2.4 acht erzeugte Bilder mit einem gemeinsamen Ursprungsbild gezeigt. Es ist zu sehen, dass nicht jeder Effekt in jedem Bild angewendet wurde.

Abbildung 2.4: Mit starker Pipeline augmentiertes Bild



Diese Tranformation wurde auf den gesamten Trainingsdatensatz angewendet. Wie an der Verteilung in Abbildung 2.5 zu sehen ist, gibt es nun in den erzeugten Trainingsdaten auch kleine Controller. Außerdem ist die Zahl der Trainingsbilder insgesamt gegenüber der ursprünglichen Trainingsbilder um den Faktor zehn gestiegen, da aus jedem ursprünglichen Bild zehn neue generiert wurden.

Abbildung 2.5: Trainingsdatensatz mit starker Augmentation



Pipeline schwache Augmentation

In einer anderen Pipeline wurden die ursprünglichen Trainingsbilder weniger stark augmentiert. Hier wurden die Bilder mit den oben beschriebenen Transformationen nur skaliert und verschoben.

Code Listing 2.2: Pipeline mit schwacher Augmentation

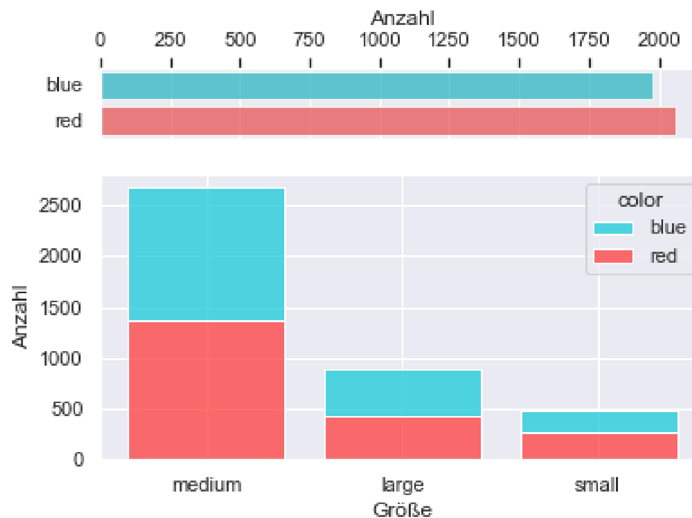
```
1 from imgaug import augmenters as iaa
2 pipeline_schwach = iaa.Sequential([
3     iaa.Affine(scale=(.25,1.1)),
4     iaa.Affine(translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)}),
5 ])
```

In dem Python Code wird nun die Pipeline des Typs `Sequential` erstellt. Mit diesem werden bei jeder Augmentation immer beide der definierten Transformationen (`Affine scale` und `Affine translate_percent`) angewendet. Jedes Bild wird also sowohl zufällig skaliert als auch in eine zufällige Richtung verschoben. Wieder wurden aus jedem Bild ca. zehn verschiedene augmentierte Bilder erstellt. In Abbildung 2.6 werden Beispielbilder, die aus der Augmentation hervorgingen, dargestellt. Wie zu erwarten war, bleibt das Bild durch die schwächere Augmentation dem Ursprung ähnlicher und unterscheidet sich lediglich in Position und Größe.

Abbildung 2.6: Mit schwacher Pipeline augmentiertes Bild

Auch mit der zweiten Augmentationspipeline gibt es nun in den Trainingsdaten kleine Controller wie in Abbildung 2.7 zu sehen ist. Wieder ist auch die Zahl der Trainingsbilder gegenüber der ursprünglichen Trainingsbilder etwa um den Faktor zehn gestiegen.

Abbildung 2.7: Trainingsdatensatz mit schwacher Augmentation



2.2 MobileNet

Im Zuge dieser Arbeit wurde ein Modell der Klasse MobileNet v1 trainiert. MobileNet beschreibt die Architektur eines Convolutional Neural Network (CNN), das für besonders kurze Inferenzzeiten und geringe Größe optimiert ist[7]. Dies sorgt dafür, dass ein MobileNet Modell insbesondre auch auf schwacher bzw. embedded hardware schnelle und präzise Objektdetektion ermöglicht. Die hohe Performance wird mit einer Methode namens ”depthwise separable convolution” erzielt. Im Vergleich zu einem herkömmlichen CNN, bei dem alle Kanäle gleichzeitig in einem convolutional Layer verarbeitet werden, werden bei der depthwise seperable convolution zunächst alle Kanäle seperat mit einem convolution layer gefiltert und dann mit einem 1x1 convolution layer zusammengefügt[7]. Dies spart einen erheblichen Teil der nötigen Rechenoperationen. Das Modell, das in dieser Arbeit benutzt wurde, ist quantisiert, statt Gleitkommawerten werden hier nur 8 Bit Integer miteinander verrechnet. Dies ist nötig, damit es mit der, im folgenden Abschnitt beschriebenen zur Inferenz verwendeten, Hardware kompatibel ist. Außerdem benötigt das Modell eine Input Größe von 300x300 Pixeln. Die Bilder, die als Input benutzt wurden, sind etwas größer und werden durch die Trainingspipeline der tensorflow object detection api[8] automatisch auf die richtige Größe skaliert. Dabei ist zu erwarten, dass durch die Skalierung die Bilder in ihrem Seitenformat verzerrt werden, weil die Kamera des RaspberryPi keine quadratischen Bilder aufnimmt. Da aber sowohl die Trainings-,

als auch der größte Teil der Testdaten sowie der tatsächliche Videostream in der live Anwendung mit der Kamera des RaspberryPi aufgenommen wurde, sind alle Daten auf die gleiche Art und Weise verzerrt. Deshalb ist das Seitenverhältnis auf dieser Hardware kein weiteres Problem.

2.3 Programmierumgebung

Das Modell wurde in der Informatik Computing Cloud (ICC) der HAW Hamburg mit einer Nvidia Tesla v100 Grafikkarte trainiert. Dazu wurde Tensorflow[1] v1.15 benutzt und mit der tensorflow object detection api[8] ein an verschiedenen Datensätzen vortrainiertes MobileNet v1 für die gegebene Problemstellung weitertrainiert (transfer learning).

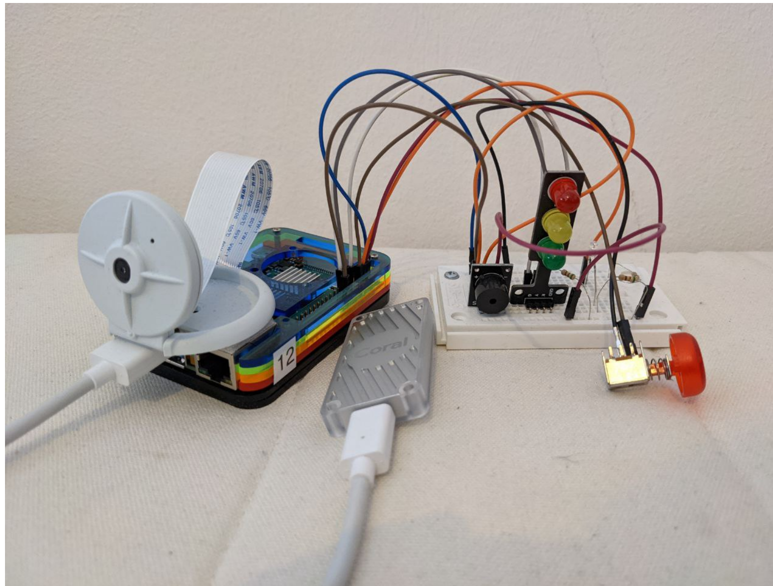
2.4 Hardware

Die Zielumgebung des Projektes ist ein RaspberryPi 4 mit 2GB RAM mit Raspberry OS. Um die Bilder aufzunehmen, ist daran eine Kamera angeschlossen. Ein Piezo Speaker wird über Pulswellenmodulation des Raspi angesprochen, um je nach Erkennungssicherheit des Modells einen Ton unterschiedlicher Höhe auszugeben. Außerdem ist eine LED Ampel an die GPIOs angeschlossen, um anzuzeigen, welche Farbe der Controller hat, der erkannt wird. Um die Inferenz auf dem RaspberryPi zu beschleunigen, ist über USB-3.0 eine Google Coral Tensor Processing Unit (TPU) mit dem RaspberryPi verbunden. Damit ein Modell von der TPU beschleunigt werden kann, muss es vollständig quantisiert sein und mit einem Skript von Google für die TPU kompiliert werden[6]. Zuletzt ist ein Schalter an den RaspberryPi angeschlossen, mit der in einem Skript die Kamera angesteuert werden kann, um auch ohne Maus und Tastatur Fotos für die Trainingsdaten schießen zu können. In Abbildung 2.8 ist das komplette technische Setup der RaspberryPi mit Peripherie auf einem Foto zu sehen.

2.5 Evaluierung der Modelle

Um ein Modell eines object detection Algorithmus zu evaluieren, benötigt es mehr als die Standardmaße Precision und Recall, da neben dem korrekten Klassifikationslabel auch noch vorhergesagt wird, wo in dem Bild sich ein Objekt befindet (in Form einer

Abbildung 2.8: RaspberryPi mit angeschlossener Peripherie



bounding box). Wenn nun also eine Vorhersage nur als korrekt klassifiziert würde, sofern neben dem Label auch die bounding Box an der exakt gleichen Stelle vorhergesagt wird, wie sie in den Trainingsdaten festgelegt ist, würde die tatsächliche Leistung des Modells stark unterschätzt werden. Denn wenn das Modell die Box im Vergleich zu der ground truth (der "wahren Position") nur um einen Pixel verschoben vorhergesagt hat, gälte diese als falsch, obwohl sie der Wahrheit schon sehr nahe kommt. Aus diesem Grunde gibt es für die object detection Maßstäbe zur Evaluierung, welche neben dem korrekten Label auch noch in Betracht ziehen, wie weit sich die vorhergesagte bounding box mit der tatsächlichen überschneidet. Mit einer solchen Evaluierungsmethode lässt sich leider keine herkömmliche confusion matrix zeichnen, da es kein eindeutig definiertes "korrekt" gibt, um die Zellen in der Matrix zu befüllen. Es bestünde zum Beispiel die Möglichkeit, alle Vorhersagen als "korrekt" zu bezeichnen, sofern das richtige Label vorhergesagt wurde und die vorhergesagte bounding box sich zu mehr als 50% mit der tatsächlichen überschneidet. Die 50% wären an dieser Stelle aber eine willkürliche Grenze und die Matrix würde den Informationsgehalt der Ergebnisdaten reduzieren, weil nicht mehr nachvollziehbar wäre, wie sehr sich die vorhergesagten und tatsächlichen Objekte überschnitten haben (und auch wie sicher das Modell sich mit der Detektion war). Aus diesem Grunde wird in dieser Arbeit eine andere Möglichkeit benutzt, die Modelle zu evaluieren und Precision und Recall zu berechnen.

2.5.1 Coco Evaluation Metrics

Um die Modelle zu evaluieren, werden die Metriken benutzt, welche auch zur Evaluierung für Modelle des coco Datensatzes benutzt werden[4]. Diese Metriken beziehen mit ein, wie groß die Überschneidung zwischen vorhergesagten und tatsächlichen bounding boxes ist und berechnet zusätzlich seperate Metriken für unterschiedliche Größen der vorhergesagten Objekte. Es wird zwischen drei verschiedenen Objektgrößen unterschieden:

- small: Objekt ist kleiner als 32^2 Pixel.
- medium: Objekt ist zwischen 32^2 und 96^2 Pixel groß.
- large: Objekt ist größer als 96^2 Pixel.

Zusätzlich gibt es noch die Option "all", bei der alle Objekte untersucht werden.

Die relevanten Metriken in den coco evaluation metrics sind Intersection over Union (IoU), Average Precision (AP) und Average Recall (AR)[11][2].

- IoU: Mit der IoU wird bestimmt, wie sehr sich zwei Flächen überlappen. Sie wird berechnet mit $\text{IoU} = \frac{\text{Fläche des Schnittes beider Flächen}}{\text{Fläche der Vereinigung beider Mengen}}$. Somit kann die IoU maximal 1 werden, wenn beide Flächen sich exakt überlappen und 0, wenn es keine Überlappung gibt.
- AP: Anders als ihr Name es vermuten lässt, werden in der Average Precision sowohl die Precision als auch der Recall (nicht Average Recall!) eines Modells in einem Wert zusammengefasst. Zur Berechnung werden für ein bestimmten festgelegten Schwellenwert einer IoU (z.B. 50%) gezählt, wieviele Objekte erkannt wurden (also sich zu mehr als 50% mit der ground truth überschneiden). Dazu werden alle Detektionen des Modells anhand ihrer Confidence (wie sicher die Vorhersage war) geordnet und gezählt, wieviele der Vorhersagen korrekt waren. Diese werden auf einer Kurve aufgetragen und gegen den Recall (den Anteil der Objekte, die erkannt wurden) geplotted. Es entsteht ein Precision/Recall Graph, aus diesem wird die Area under the Curve (AUC) berechnet, welche Average Precision genannt wird. Es hat sich nebenher etabliert, die AP nicht mit einem einzigen festen IoU zu berechnen, sondern im Intervall von IoU [0.5, 0.95] gleichmäßig verteilt unterschiedliche APs zu berechnen und darüber einen Mittelwert zu berechnen. Dieser Mittelwert kann auch mean Average Precision genannt werden. In dieser Hausarbeit wird im Subskript beschrieben, welche IoU für die Berechnung der AP verwendet wurde (z.B

$AP_{0.50:0.95}$ für die mean Average Precision im Intervall IoU [0.5, 0.95] und $AP_{0.50}$ für die AP mit IoU 0.5).

- AR: Um den Average Recall zu bestimmen, wird zu unterschiedlichen IoU Schwellenwerten berechnet, wie groß der Anteil der erkannten Objekte ist. Danach werden alle Anteile gemittelt. In der Regel werden die Anteile der erkannten Objekte in dem Intervall von IoU [0.5, 0.95] berechnet und darüber gemittelt.

In den coco evaluation metrics werden üblicherweise die unterschiedlichen Klassen, die erkannt werden sollen (im Falle dieser Arbeit der blaue und der rote Controller) zusammengenommen und nur ein gemeinsamer Wert für diese berechnet.

3 Ergebnisse

Für die drei beschriebenen Datensätze (ohne Augmentation, Starke Augmentation, Schwache Augmentation) wurde jeweils ein MobileNet v1 Modell mit 70.000 Steps und einer Batch Size von 12 Bildern trainiert. Im Folgenden wird die Güte der Modelle dargestellt, die mit den drei verschiedenen Trainingsdatensätzen trainiert wurden. Die berechneten coco detection metrics sind für jedes Modell mit den gleichen Testdaten berechnet worden, um eine Vergleichbarkeit zwischen den Modellen zu gewährleisten.

3.1 Ohne Augmentation

In Tabelle 3.1 sind die Ergebnisse des Modells abgetragen, welche ohne augmentierte Trainingsdaten trainiert wurden. Das Modell erkennt Objekte besser, wenn sie größer sind ($AP_{0.50:0.95}$ für mittelgroße Bilder liegt bei 0.206 und bei großen Bildern bei 0.509). Kleine Objekte kann es gar nicht erkennen.

3.2 Starke Augmentation

Die Metriken, die aus dem Modell mit stark augmentierten Daten stammen, sind in Tabelle 3.2 abgezeichnet. Diese sind durchweg schlechter als die, des Modells ohne Augmen-

Tabelle 3.1: Ergebnisse Datensatz ohne Augmentation

Metrik	Fläche	Wert
$AP_{0.50:0.95}$	all	0.314
$AP_{0.50}$	all	0.533
$AP_{0.75}$	all	0.362
$AP_{0.50:0.95}$	small	0.0
$AP_{0.50:0.95}$	medium	0.206
$AP_{0.50:0.95}$	large	0.509
$AR_{0.50:0.95}$	all	0.411
$AR_{0.50:0.95}$	small	0.0
$AR_{0.50:0.95}$	medium	0.285
$AR_{0.50:0.95}$	large	0.643

Tabelle 3.2: Ergebnisse Datensatz mit starker Augmentation

Metrik	Fläche	Wert
$AP_{0.50:0.95}$	all	0.109
$AP_{0.50}$	all	0.199
$AP_{0.75}$	all	0.121
$AP_{0.50:0.95}$	small	0.0
$AP_{0.50:0.95}$	medium	0.09
$AP_{0.50:0.95}$	large	0.162
$AR_{0.50:0.95}$	all	0.229
$AR_{0.50:0.95}$	small	0.0
$AR_{0.50:0.95}$	medium	0.19
$AR_{0.50:0.95}$	large	0.322

tation. Erkennbar ist dies zum Beispiel an der $AP_{0.50:0.95}$, welche ohne die augmentierten Daten bei 0.314 lag und mit augmentierten Daten auf 0.109 gefallen ist. Es werden wieder keine kleinen Objekte erkannt.

3.3 Schwache Augmentation

Das Modell, das mit den schwach augmentierten Daten trainiert worden ist (Ergebnisse in Tabelle 3.3), schneidet in fast allen Metriken besser ab als das Modell mit nicht augmentierten Daten. Insbesondre in dem allgemeinsten Kriterium der $AP_{0.50:0.95}$ konnte der Wert von 0.314 durch die Augmentation auf 0.366 erhöht werden. Auch hier kann man beobachten, dass Objekte besser erkannt werden, wenn sie größer sind ($AP_{0.50:0.95}$

Tabelle 3.3: Ergebnisse Datensatz mit schwacher Augmentation

Metrik	Fläche	Wert
$AP_{0.50:0.95}$	all	0.366
$AP_{0.50}$	all	0.604
$AP_{0.75}$	all	0.406
$AP_{0.50:0.95}$	small	0.02
$AP_{0.50:0.95}$	medium	0.313
$AP_{0.50:0.95}$	large	0.504
$AR_{0.50:0.95}$	all	0.449
$AR_{0.50:0.95}$	small	0.02
$AR_{0.50:0.95}$	medium	0.378
$AR_{0.50:0.95}$	large	0.615

bei mittelgroßen Objekten bei 0.313 vs. 0.504 bei großen Objekten). Interessant ist, dass nun selbst kleine Objekte erkannt werden können, auch wenn nur mit einer sehr geringen Wahrscheinlichkeit ($AR_{0.50:0.95}$ für kleine Objekte liegt bei 0.02). Dafür, dass kleine Objekte nun erkannt werden, hat aber die Erkennungsleistung für große Objekte etwas gelitten. Insbesondre der Recall ist etwas geringer bei großen Objekten als vor der Augmentation ($AR_{0.50:0.95}$ lag ohne Augmentation bei 0.643 und ist mit Augmentation auf 0.615 gesunken).

3.4 Auf dem RaspberryPi

Um live auf dem RaspberryPi in dem Videostream der Webcam Objekte zu detektieren, wurde ein Skript geschrieben, welches in einer Endlosschleife neue Bilder einliest, die Inferenz ausführt, das Bild mit den bounding boxen der erkannten Objekte anzeigt und die entsprechenden Aktoren antreibt. Das Skript ist orientiert an einem Skript von EdjeElectronics [5], welches wiederum orientiert ist an dem Skript der tensorflow Entwickler*innen[13]. Die live Erkennung auf dem RaspberryPi läuft auf der Coral TPU mit etwa 24 FPS. Es konnte im Python Skript gemessen werden, dass die Inferenz aber lediglich 10ms in Anspruch nimmt. Der größte Zeitanteil bei der Ausführung des Programmes wird also durch das Aufnehmen, Einladen, Annotieren und Anzeigen der Bilder beansprucht. Mit dem Mobilenet auf der Coral TPU könnten also pro Sekunde 100 Bilder erkannt werden, sofern diese alle bereits im Arbeitsspeicher vorhanden wären. Eine Live Demonstration des Projektes kann in einem kurzen Video auf Youtube unter folgendem Link gefunden werden: `https://youtu.be/eHBGWim-vzw`. Dort ist zu sehen, dass

je nach Genauigkeit ein unterschiedlich hoher Ton ausgegeben wird und eine LED in unterschiedlichen Farbe leuchtet, je nachdem welcher Controller erkannt wird.

4 Diskussion und Fazit

In dieser Hausarbeit konnte gezeigt werden, dass durch die tensorflow object detection API mit vergleichweise wenig Aufwand ein eigenes object detection Modell trainiert werden kann. Dieses Modell kann, sofern es quantisiert ist, mit einer externen TPU eine sehr schnelle Inferenz vorweisen und auf einer embedded hardware mit entsprechenden Aktoren auch ohne Display und stationärem Stromanschluss mit seiner Funktionaliät aufweisen. Außerdem wurde gezeigt, dass durch gezielte Bildaugmentation die Erkennungsleistung eines Modelles erhöht werden kann. Dies ist insbesondre relevant für Kontexte, in dem nur begrenzt viele Daten für das Training zur Verfügung stehen. Was sich aber auch gezeigt hat, ist, dass durch Augmentation (und damit Erhöhung der Anzahl der Trainingsdatensätze) nicht unbedingt die Leistung eines Modells verbessert wird, so wurde bei einer falsch gewählten Augmentationspipeline die Leistung sogar verschlechtert. Es funktionieren also nicht alle Transformationen eines Bildes gleich gut für die Bildaugmentation, als Beispiel wäre hier die Drehung eines Bildes zu nennen, durch dessen Einfluss die Bounding Box eines Objektes nach der Drehung zu groß wird. Dies zeigt sich im folgenden Beispielbild in Abbildung 4.1, bei dem nach der Drehung die Box nicht mehr genau um das Objekt passt.

Abbildung 4.1: Einfluss der Drehung auf die Bounding Box

Außerdem funktioniert eine Spiegelung der Input Bilder als Augmentation bei Daten nicht, welche nur in einer Spiegelrichtung richtig sind, wie beispielsweise Schriftzeichen oder den meisten Zahlen. Es lässt sich also schlussfolgern, dass für jeden spezifischen Anwendungskontext die richtige Augmentationspipeline gefunden werden muss, um die gewünschten Daten optimal zu ergänzen.

Um die Ergebnisse für das in der Hausarbeit erstellte Modell noch weiter zu verbessern, könnten noch weitere Datensätze mit unterschiedlicheren Augmentationspipelines erstellt werden. Dazu könnte anhand der Testbilder analysiert werden, an welchen Stellen das Modell Probleme hat bzw. nicht die richtigen Objekte erkennt, um gezielt für diese Fälle die Daten zu augmentieren.

Literaturverzeichnis

[1] Abadi, Martín ; Agarwal, Ashish ; Barham, Paul ; Brevdo, Eugene ; Chen, Zhifeng ; Citro, Craig ; Corrado, Greg S. ; Davis, Andy ; Dean, Jeffrey ; Devin, Matthieu ; Ghemawat, Sanjay ; Goodfellow, Ian ; Harp, Andrew ; Irving, Geoffrey ; Isard, Michael ; Jia, Yangqing ; Jozefowicz, Rafal ; Kaiser, Lukasz ; Kudlur, Manjunath ; Levenberg, Josh ; Mané, Dandelion ; Monga, Rajat ; Moore, Sherry ; Murray, Derek ; Olah, Chris ; Schuster, Mike ; Shlens, Jonathon ; Steiner, Benoit ; Sutskever, Ilya ; Talwar, Kunal ; Tucker, Paul ; Vanhoucke, Vincent ; Vasudevan, Vijay ; Viégas, Fernanda ; Vinyals, Oriol ; Warden, Pete ; Wattenberg, Martin ; Wicke, Martin ; Yu, Yuan ; Zheng, Xiaoqiang: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. – URL https://www.tensorflow.org/. – Software available from tensorflow.org

[2] Aidouni, Manal E.: *Evaluating Object Detection Models: Guide to Performance Metrics*. 2019. – URL https://manalelaidouni.github.io/manalelaidouni.github.io/Evaluating-Object-Detection-Models-Guide-to-Performance-Metrics.html#precision. – [Online; Stand 30. Januar 2021]

[3] Amit, Yali ; Felzenszwalb, Pedro ; Girshick, Ross: Object detection. In: *Computer Vision: A Reference Guide* (2020), S. 1–9

[4] Coco: *Detection Evaluation*. – URL https://cocodataset.org/#detection-eval. – [Online; Stand 30. Januar 2021]

[5] EdjeElectronics: *TFLite_detection_webcam.py*. – URL https://github.com/EdjeElectronics/TensorFlow-Lite-Object-Detection-on-Android-and-Raspberry-Pi/blob/master/TFLite_detection_webcam.py. – [Online; Stand 25. Februar 2021]

[6] GOOGLE: *Edge TPU Compiler.* – URL https://coral.ai/docs/edgetpu/compiler/. – [Online; Stand 25. Februar 2021]

[7] HOWARD, Andrew G. ; ZHU, Menglong ; CHEN, Bo ; KALENICHENKO, Dmitry ; WANG, Weijun ; WEYAND, Tobias ; ANDREETTO, Marco ; ADAM, Hartwig: Mobilenets: Efficient convolutional neural networks for mobile vision applications. In: *arXiv preprint arXiv:1704.04861* (2017)

[8] HUANG J, Sun C.: *Speed/accuracy trade-offs for modern convolutional object detectors.* 2018

[9] JUNG, Alexander B.: *imgaug.* 2018. – URL https://github.com/aleju/imgaug. – [Online; Stand 30. Januar 2021]

[10] KISANTAL, Mate ; WOJNA, Zbigniew ; MURAWSKI, Jakub ; NARUNIEC, Jacek ; CHO, Kyunghyun: Augmentation for small object detection. In: *arXiv preprint arXiv:1902.07296* (2019)

[11] KUMAR, Harshit: *Evaluation metrics for object detection and segmentation: mAP.* – URL https://kharshit.github.io/blog/2019/09/20/evaluation-metrics-for-object-detection-and-segmentation. – [Online; Stand 30. Januar 2021]

[12] MIKOŁAJCZYK, Agnieszka ; GROCHOWSKI, Michał: Data augmentation for improving deep learning in image classification problem. In: *2018 international interdisciplinary PhD workshop (IIPhDW)* IEEE (Veranst.), 2018, S. 117–122

[13] TENSORFLOW: *label_image.py.* – URL https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/examples/python/label_image.py. – [Online; Stand 25. Februar 2021]

[14] TZUTALIN: *LabelImg.* 2015. – URL https://github.com/tzutalin/labelImg. – [Online; Stand 30. Januar 2021]

[15] VLADIMIROV, Lyudmil: *Tensorflow object detection API Documentation.* 2018. – URL https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/tensorflow-1.14/index.html. – [Online; Stand 20. Februar 2021]

[16] ZOU, Zhengxia ; SHI, Zhenwei ; GUO, Yuhong ; YE, Jieping: Object detection in 20 years: A survey. In: *arXiv preprint arXiv:1905.05055* (2019)