

Dokumentation der Deep-Learning App

Mask - Detector



Gruppe 13

557110 Clemens Brauer

553610 Anton Schaarschmidt

Inhaltsverzeichnis

1	Einleitung	4
2	Vorbereitung	5
3	Datenbeschaffung	5
3.1	Quellen	6
3.2	Image Augmentation	7
3.3	Weitere potentielle Datenquellen	8
4	Plattform / Framework Selection	8
5	Modellauswahl	9
5.1	Überblick	9
5.2	Datensets	9
5.3	Modells	10
5.4	Weitere Modell Quellen	11
6	Trainingsvorbereitung	12
6.1	Split Data	12
6.2	Generierung der TFRecord Dateien	12
6.3	Erstellung der labelmap	12
6.4	Trainingskonfiguration der Modelle	13
7	Training	13
7.1	Trainingsprozess	14
7.2	Evaluation des Trainingsprozesses	14
7.3	Evaluationsauswertung	15
7.4	Evaluationsergebnisse	17
8	Modell Konvertierung und Optimierung	23
8.1	Konvertierung	23
8.2	Optimierung	24
8.3	Ergebnisse	25
9	App-Entwicklung	26
9.1	Wireframes	26
9.2	IOS Entwicklung	30
9.3	Installation und Modell-Updates	35
9.4	Test	35
10	Ergebnis & Verbesserung	36
10.1	Erkennung	36
10.2	Geschwindigkeit	38
10.3	Was würden wir das nächste Mal anders machen?	39

Abbildungsverzeichnis

Abbildung 1 Beispiel einer Object-Detection-Klassifizierung	5
Abbildung 2 Übersicht des Kaggle Datensatz	6
Abbildung 3 Mann ohne Maske?.....	7
Abbildung 4 Verteilung der Kategorien	7
Abbildung 5 Simulated masked faces.....	8
Abbildung 6 CSV-Ausschnitt eines Datensatz	12
Abbildung 7 Ausschnitt Trainingsprozess Konsole.....	14
Abbildung 8 Ausschnitt Evaluation Konsole.....	15
Abbildung 9 Beispieldarstellung ssd_mobilenet_v2 Erkennungsrate: with_mask.....	16
Abbildung 10 Beispieldarstellung ssd_mobilenet_v2 Evaluationsimage Step 7000.....	16
Abbildung 11 Trainingsresultat facessd_mobilenet_v2.....	17
Abbildung 12 Gegenüberstellung Training & Validation Loss facessd_mobilenet_v2	18
Abbildung 13 Trainingsresultat ssd_mobilenetv2_oidv4	19
Abbildung 14 Gegenüberstellung Training & Validation Loss ssd_mobilenetv2_oidv4.....	19
Abbildung 15 Trainingsresultat ssd_mobilenet_v1_fpn	20
Abbildung 16 Gegenüberstellung Training & Validation Loss ssd_mobilenet_v1_fpn	21
Abbildung 17 Fehlermeldung ssd_mobilenet_dsp.....	21
Abbildung 18 ssdlite_mobilenet_v2	22
Abbildung 19 Gegenüberstellung Training & Validation Loss ssdlite_mobilenet_v2.....	22
Abbildung 20 Durchschnittliche Ergebnisse der Modelle	23
Abbildung 21 Quantization Methoden im Vergleich.....	24
Abbildung 22 Start Screens	27
Abbildung 23 Time-Laps-Ansicht	28
Abbildung 24 Aufnahmen-Ansicht.....	28
Abbildung 25 Aufnahmeoptionen-Ansicht.....	29
Abbildung 26 Reports-Ansicht	29
Abbildung 27 Time-Laps-Ansicht Umsetzung	30
Abbildung 28 Aufnahmen-Ansicht Umsetzung.....	31
Abbildung 29 Full-Image-Ansicht Umsetzung.....	31
Abbildung 30 Aufnahmeoptionen-Ansicht Umsetzung.....	32
Abbildung 31 Time-Laps-Aufnahme-Ansicht Umsetzung.....	32
Abbildung 32 Report-Ansicht Umsetzung.....	33
Abbildung 33 Kommunikationskonzept	34
Abbildung 34 spezielles Masken-Designs	37

Tabellenverzeichnis

Tabelle 1 Verteilungsergebnis der Kategorien	8
Tabelle 2 Gegenüberstellung der Frameworks	9
Tabelle 3 Übersicht Modelle.....	11
Tabelle 4 Konfigurationsparameter	13
Tabelle 5 facessd_mobilenet_V2 Konfiguration	17
Tabelle 6 ssd_mobilenetv2_oidv4 Konfiguration.....	18
Tabelle 7 ssd_mobilenet_v1_fpn Konfiguration	20
Tabelle 8 ssdlite_mobilenet_v2 Konfiguration.....	21
Tabelle 9 Optimierungsresultat.....	25
Tabelle 10 Optimierungsresultat Benchmark	26
Tabelle 11 Testergebnisse.....	36

1 Einleitung

Seit Anfang des Jahres stehen wir als Menschheit vor der Herausforderung die globale Verbreitung des COVID-19 Virus einzudämmen. Die bisher vorliegenden Informationen zur Epidemiologie des Virus zeigen, dass Übertragungen insbesondere bei engem ungeschütztem Kontakt zwischen Menschen vorkommen. Nach derzeitigem Kenntnisstand des Robert-Koch-Instituts (RKI)¹ erfolgt die Übertragung vor allem über respiratorische Sekrete, in erster Linie Tröpfchen, die z.B. beim Husten, Niesen, oder lautem Sprechen freigesetzt werden. Aus diesem Grund empfiehlt das RKI das Tragen eines mehrlagigen medizinischen Mund-Nasen-Schutzes, um zum einen die Freisetzungerregerhaltiger Tröpfchen aus dem eigenen Nasen-Rachen-Raum zu behindern und zum anderen die Aufnahme von Tröpfchen oder Spritzern aus dem Nasen-Rachen-Raum des Gegenübers zu verhindern.

Aus diesem Grund besteht unter anderem in Berlin die Pflicht zum Tragen des Mund-Nasen-Schutzes in allen öffentlichen Einrichtungen sowie im öffentlichen Nahverkehr. Um zu überprüfen, ob die angeordneten Bestimmungen von der Bevölkerung angenommen werden, ist unsere Idee eine automatisierte statistische Erhebung durchzuführen, die auf Grundlage von Deep-Learning erfasst, ob der Mund-Nasen-Schutz regelkonform getragen wird.

Ein Ansatz dafür könnte perspektivisch die Installation von IOT Geräten an öffentlichen Plätzen sein, welche mithilfe eines vorab trainierten neuronalen Netzes live das Tragen von Masken durch Kameraaufnahmen ermitteln. Die dabei erfassten Daten, können im Anschluss für eine statistische Auswertung genutzt werden.

Um dieser Idee einen Schritt näher zu kommen, möchten wir in dieser Ausarbeitung ein Object-Detection-Modell trainieren, welches auf Bildern Gesichter erkennt, die den Mund-Nasen-Schutz tragen, nicht tragen oder falsch tragen. Dieses Modell soll im Anschluss prototypisch in eine IOS App eingebunden werden, um es im live Einsatz testen zu können. Dabei sollen in der App bereits eine Zählung sowie statistische Auswertung der erkannten Kategorien erfolgen.

Grundlegend ist das Modell und die dazugehörige Applikation nicht für den kommerziellen Gebrauch vorgesehen. Die Zielgruppe könnten Institute wie das RKI sein, welche eine Veränderung der Reproduktionszahl mit dieser statistischen Auswertung in Zusammenhang setzen könnte. Auch Anwendungsmöglichkeiten im Bereich der Soziologie sind denkbar, wobei ermittelt werden kann, wie eine Bevölkerung mit dem Umstand der Pandemie umgeht. Weiterhin könnten staatliche Einrichtungen die Anwendung nutzen, um zu erkennen, wie gut die angeordneten Bestimmungen von der Bevölkerung befolgt werden.

Um eine qualitativ hochwertige statistische Aussage treffen zu können, ist eine hohe Genauigkeitsrate des Modells wichtig. Zudem ist es entscheidend, dass das Modell schnell reagiert, da es in dem perspektivischen IOT Projekt Live-Zählungen durchführen soll. Daher streben wir eine Modellgenauigkeit von mindestens 95% sowie eine für Live-Erkennung geeignete hohe Geschwindigkeit an.

Um die Dokumentation übersichtlicher zu gestalten haben wir ein Git-Repository² erstellt, welches Skripte, Bilder und Videos enthält. Die Ordnerstruktur in dem Repository entspricht dabei der Inhaltsstruktur der Dokumentation. Verweise auf dieses Repository werden mit *Git Repository*: angeführt.

¹ https://www.rki.de/DE/Home/homepage_node.html

² https://github.com/SwiftJimmy/Hausarbeit_Wissensmanagement_Deep_Learning

2 Vorbereitung

Im ersten Schritt haben wir recherchiert, mit welchem Verfahren wir einzelne Objekte auf Bildern erkennen und klassifizieren können. Dabei sind wir schnell auf den Begriff *Object-Detection* gestoßen. Anders als bei einer einfachen Bild-Klassifizierung, werden bei einem Object-Detection-Verfahren einzelne Objekte in einem Bild lokalisiert, mit Bounding-Boxes umrahmt und im Anschluss kategorisiert (siehe Abbildung 1).

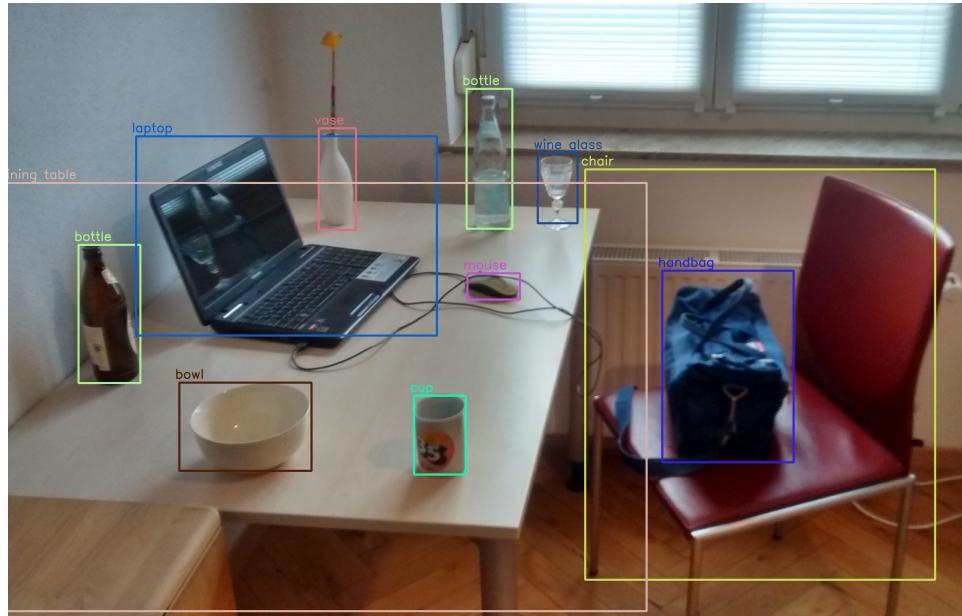


Abbildung 1 Beispiel einer Object-Detection-Klassifizierung

Unsere Idee war es, ein Object-Detection-Modell zu trainieren, welches Gesichter in die Kategorien „Mit Maske“, „Ohne Maske“ und „Maske falsch getragen“ klassifiziert. Im Anschluss soll die Anzahl der erkannten Gesichter in Abhängigkeit ihrer zugewiesenen Kategorie ermittelt und ausgegeben werden.

Als problematisch ist uns bereits während der Recherche aufgefallen, dass bei der Zählung während einer Live-Aufnahme berücksichtigt werden muss, ob ein Gesicht bereits erkannt und gezählt wurde, da ansonsten Mehrfachzählungen auftreten, die die Statistik verfälschen. Aufgrund dieser Schwierigkeit haben wir uns entschieden die Funktionalität der App auf die Aufnahme von Bildern zu beschränken. Da unser trainiertes Modell dennoch zukünftig für Live-Erkennung und -Zählung genutzt werden soll, haben wir uns dazu entschieden, die in der Einleitung angestrebten Performance-Metriken als Ziel weiter beizubehalten.

3 Datenbeschaffung

Bei der Datenbeschaffung haben wir uns darauf fokussiert, Bilddaten zu finden, welche bereits anhand der Kategorien „Mit Maske“, „Ohne Maske“ und „Maske falsch getragen“ annotiert wurden. Dafür haben wir Google Dataset Search³ genutzt. Die Datenquellen, die gefundenen Ergebnisse, sowie unsere Herangehensweise bei der Vermehrung der Daten werden in diesem Kapitel beschrieben.

³ <https://datasetsearch.research.google.com/>

3.1 Quellen

Der Datensatz *face-mask-detection*⁴, welcher über die Plattform Kaggle zur Verfügung gestellt wird, dient mit bereits 848 annotierten JPG Bildern als Grundlage für diese Arbeit. Die Annotation liegen dabei im PASCAL VOC Format als XML Dateien vor und beinhaltet die Bildkoordinaten und Labels der drei Kategorien *with_mask*, *without_mask* und *mask_weared_incorrect* (Der grammatische Fehler „weared“ ist leider in dem Datensatz vorgegeben). Abbildung 2 gibt einen detaillierten Überblick über den kompletten Datensatz.

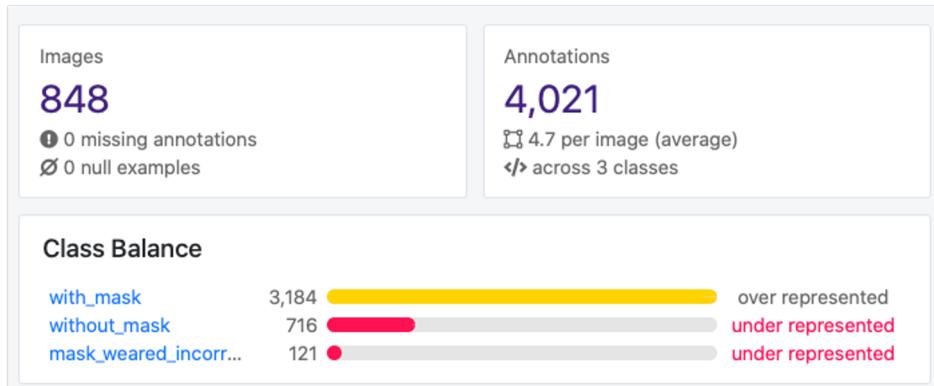


Abbildung 2 Übersicht des Kaggle Datensatz

Wie in der Abbildung zu erkennen, ist das Aufkommen der einzelnen Kategorien schlecht verteilt. Der Anteil der gelabelten Personen mit Maske ist mit einer Anzahl von 3184 viel höher als der Anteil der gelabelten Personen mit falsch getragener oder ohne Masken.

Um dieses Missverhältnis auszugleichen, haben wir 400 weitere Bilder über die Google-Image-Search, Instagram und aus einem weiteren Kaggle Datensatz (*medical-mask-dataset*)⁵ gesammelt und diese mithilfe des Open-Source-Tools LabelImg⁶ per Hand annotiert. Das medical-mask-dataset war dabei bereits in die Kategorien *good* (äquivalent zu *with_mask*) und *bad* (äquivalent zu *without_mask* oder *mask_weared_incorrect*) annotiert. Um uns die Arbeit zu erleichtern, haben wir ein Python Script⁷ geschrieben, welches bereits die Labels entsprechend unserer Beschriftung umformatiert. Anschließend mussten wir noch die vorab als *bad* gelabelten Kategorien in unsere Kategorien *without_mask* und *mask_weared_incorrect* per Hand sortieren.

Dabei war es besonders herausfordernd festzulegen, ab wann eine Abbildung als *mask_weared_incorrect* oder *without_mask* gelabelt wird. In Abbildung 3 wird dieser Konflikt verdeutlicht.

⁴ <https://www.kaggle.com/andrewmvd/face-mask-detection>

⁵ <https://www.kaggle.com/vtech6/medical-masks-dataset>

⁶ <https://github.com/tzutalin/labelImg>

⁷ Git Repository: 3. Datenbeschaffung/3.1 xml_From_good_to_with_mask.py



Abbildung 3 Mann ohne Maske?

Das Gesicht des Jungen auf der rechten Seite kann ohne Probleme als *mask_weared_incorrect* bestimmt werden. Im Falle des Herrn auf der linken Seite viel uns diese Entscheidung schon etwas schwerer. Eine *mask_weared_incorrect* Annotierung könnte den Algorithmus dahingehend verwirren, dass sein Gesicht, wie bei als *without_mask* markierten Personen, frei von Abdeckungen ist. Dennoch ist eindeutig erkennbar, dass die Maske „falsch“ getragen wird. Die bereits annotierten Kaggle Bilder konnten uns diese Entscheidung nicht abnehmen, da diese immer recht eindeutig waren. Um wiederholtes Annotieren zu vermeiden, haben wir uns darauf geeinigt alle Gesichter, bei denen sich die Maske unterhalb der Nase befindet, als *mask_weared_incorrect* zu annotieren.

Insgesamt konnten wir dadurch die Anzahl von 337 Bildern, welche die Labels *without_mask* und *mask_weared_incorrect* beinhalten, auf 737 erweitern.

3.2 Image Augmentation

Da es sich bei den vorab beschriebenen Vorgehen der Bildersuche und Annotierung um sehr zeitintensive Verfahren handelt, haben wir weiterhin Image Augmentation angewandt, um die Verteilung der Kategorien anzugeleichen. Zur Umsetzung haben wir ein Python Script geschrieben⁸, welches die 737 Bilder sowie deren bestehenden Annotationen wie folgt verarbeitet:

- Zufällige Rotation um -20 bis -7 Grad
- Zufällige Rotation um 7 bis 20 Grad
- Vertikale Spiegelung
- Zufälliger Verkleinerung auf 50-80 % der originalen Größe

Als Ergebnis konnten wir unseren gesamten Datensatz von ursprünglich 848 Bildern auf insgesamt 4257 erweitern und dabei eine in Abbildung 4 dargestellte Kategorienverteilung erzielen.

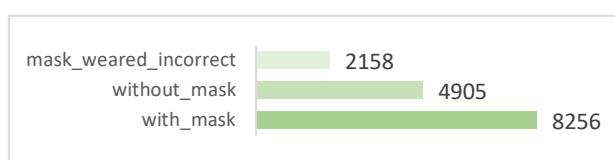


Abbildung 4 Verteilung der Kategorien

⁸ Git Repository: 3. Datenbeschaffung/3.2 image_Augmentation.py

Auch wenn die Verteilung noch nicht perfekt ausgewogen ist, konnten wir das Verhältnis der einzelnen Kategorien, gegenüber dem in Abbildung 2 dargestellten Grunddatensatz, deutlich verbessern. Das Ergebnis der Verteilung ist in Tabelle 1 gegenübergestellt.

Kategorie	Verteilung-Grunddatensatz	Verteilung-Ergebnis	Veränderung
Mask weared incorret	3%	14,1%	+11,1%
Withou mask	17,8%	32%	+14,2%
With mask	79,2%	53,9%	-25,3%

Tabelle 1 Verteilungsergebnis der Kategorien

3.3 Weitere potentielle Datenquellen

Auf der Suche nach geeigneten Daten sind wir auf weitere interessante Datensätze für Masken tragende Personen gestoßen. Beispielsweise wird über den Github Account *UniversalDataTool*⁹ eine CSV Datei zur Verfügung gestellt, die auf Instagram-Bilder von Personen verweist, welche eine Maske oder keine Maske tragen. Dabei wurden die Masken genauer in die Kategorien *medical_mask* und *not_medical_mask* unterschieden.

Ein weiteres Verfahren zum Generieren von Bildern wird in dem Journal-Artikel *Masked Face Recognition Dataset and Application*¹⁰ von Wang et al. beschrieben. Dabei werden *simulated masked face images* erstellt, indem Bilder von Masken über Gesicherter gelegt werden (siehe Abbildung 5).



Abbildung 5 Simulated masked faces

Das Dataset wird über den Github Account *X-zhangyang*¹¹ zur Verfügung gestellt. Durch dieses Verfahren können die Masken beliebig farblich verändert werden, um diese Unterschiede im Training zu erlernen.

4 Plattform / Framework Selection

Wie bereits in der Einleitung beschreiben, könnte unsere Lösung perspektivisch an öffentlichen Plätzen genutzt werden, um mithilfe von IOT Geräten das Tragen von Masken statistisch auszuwerten. In unserer Ausarbeitung erstellen wir dieses System prototypisch auf einem iPhone, um unser trainiertes Modell vorab im live Einsatz testen zu können. Aus diesem Grund benötigen wir ein Open-Source-Framework, welches sowohl auf IOT-, als auch mit IOS Geräten genutzt werden kann. Bei der Auswahl haben wir die in Tabelle 2 aufgelisteten Frameworks betrachtet und jeweils die beiden Punkte *Unterstützte Plattformen* und *Open-Source* bewertet.

⁹ <https://github.com/UniversalDataTool/coronavirus-mask-image-dataset>

¹⁰ <https://arxiv.org/pdf/2003.09093.pdf>

¹¹ <https://github.com/X-zhangyang/Real-World-Masked-Face-Dataset>

Framework	Unterstützte Plattformen	Open-Source
PyTorch ¹²	Auf der Webseite ist das Deployment auf IOS und Android beschrieben	✓
TensorFlow ¹³	Auf der Webseite ist das Deployment auf IOS, Android und IOT Geräten beschrieben	✓
Fritz.ai ¹⁴	Auf der Webseite ist das Deployment auf IOS und Android beschrieben	✗

Tabelle 2 Gegenüberstellung der Frameworks

PyTorch ist ein Open-Source-Framework und bietet auf seiner Webseite Deployment-Möglichkeiten für IOS und Android Geräte. Über das Deployment auf IOT Geräten konnten wir keine Informationen auf der Webseite finden. Fritz.ai ist nicht Open-Source und ist daher für uns nicht interessant. Da TensorFlow definitiv alle Anforderungen erfüllt, haben wir uns bei der Er- und Bereitstellung des Computer-Vision-Modells für dieses Framework entschieden. Unterstützend bei der Auswahl hat uns TensorFlow mit einer übersichtlichen Dokumentation, sowie der einfachen Möglichkeit das Modell effizient und speichersparend in einer optimierten Form auf den Endgeräten zu deployen, überzeugt.

5 Modellauswahl

Bei der Erstellung des Modells wird in dieser Arbeit auf die Methode des Transfer-Learning zurückgegriffen. Neben dem Vorteil einer geringeren Trainingszeit, kann mit diesem Verfahren auch mit einem relativ kleinen Datensatz gute Ergebnisse erzielt werden. In diesem Kapitel beschreiben wir unsere Herangehensweise bei der Auswahl der zu betrachtenden Pre-Trained Modelle.

5.1 Überblick

TensorFlow bietet über ein Git-Repository¹⁵ den so genannten *TensorFlow detection model zoo* an. Dabei handelt es sich um eine Sammlung von Pre-Trained Modellen für Object-Detection, welche beispielsweise für Transfer-Learning genutzt werden können. Die Modelle wurden anhand von unterschiedlichen Datensets erstellt und bezüglich ihrer Geschwindigkeit (*Speed ms*) und Genauigkeit (*mAP*) bewertet. Zusätzlich erhalten jene die Bezeichnung *mobilenet*, welche in das von uns später gewünschte TensorFlow-lite Format konvertiert werden können und für mobile Geräte angepasst wurden. In diesem Abschnitt beschreiben wir unser Vorgehen für das Auswahlverfahren der für das Transfer-Learning verwendeten Modelle, auf Grundlage des TesorFlow detection model zoo.

5.2 Datensets

Im ersten Schritt haben wir die Datensets betrachtet, auf welchen die verschiedenen Modelle trainiert wurden. Dabei war unser Ziel herauszubekommen, auf Grundlage welcher Kategorien die Datensets erstellt wurden, um möglichst unserem Anwendungsfall ähnelnde Modelle zu verwenden. Wie bereits einleitend beschrieben, haben wir den Fokus dabei ausschließlich auf Modelle gelegt, welche die Bezeichnung *mobilenet* beinhalten. Durch diese Vorsortierung erhielten wir die Auswahl von Modellen aus den folgend beschriebenen Datensets:

¹² <https://pytorch.org>

¹³ <https://www.tensorflow.org>

¹⁴ <https://www.fritz.ai>

¹⁵ https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md

COCO¹⁶

Dieses Datenset besteht aus ca. 200000 Bildern, welche basierend auf 91 Kategorien annotiert wurden. Die Kategorien decken eine Auswahl von allgemeinen Objekten wie Personen, Verkehrszeichen oder Haushaltsgegenständen ab. Insgesamt beinhaltet das Datenset rund 1,5 Millionen Bounding-Boxes.

Open Images¹⁷

Open Images bietet Datensets in verschiedenen Versionen an. Die im *Tensorflow model zoo* angebotenen Modelle wurden auf Grundlage der Open Image Version 2 und 4 erstellt.

- Version 2 besteht aus rund 800000 Bildern, welche basierend auf 600 verschiedenen Kategorien annotiert wurden. Insgesamt beinhaltet das Datenset 2 Millionen Bounding-Boxes.
- Version 4 besteht aus rund 1,74 Millionen Bildern, welche basierend auf 600 verschiedenen Kategorien annotiert wurden. Insgesamt beinhaltet das Datenset 14,6 Millionen Bounding-Boxes.

Die Kategorien umfassen dabei Objekte aus dem täglichen Leben wie Bäume, Personen, Gesichter oder Möbel.

Beide Datensätze beinhalten Kategorien von Objekten aus dem alltäglichen Leben, wobei der Open Images Datensatz diese detaillierter beschreibt. Dies ist gut an dem Beispiel Person erkennbar. In dem COCO Datensatz wird auf kleinster Detailebene die Person als Person erkannt. In dem Open Images Datensatz wurden hingegen neben der Person auch das Gesicht, der Kopf, die Hände, die Haare, die Nase und der Körper annotiert. Aufgrund des höheren Detailierungsgrad und der Erkennung von Gesichtern, was unserem Anwendungsfall am nächsten kommt, sind wir davon ausgegangen, dass die Modelle des Open Images bessere Ergebnisse durch Transfer Learning erzielen werden. Da jedoch insgesamt nur zwei *mobilenet* Modelle, welche auf Grundlage des Open Images Datenset trainiert wurden, in dem *Tensorflow model zoo* angeboten werden, haben wir bei der Modell Auswahl auch COCO Modelle betrachtet.

5.3 Modelle

Insgesamt bietet der *Tensorflow model zoo* 14 mobilenet Modelle zur Auswahl an. Wie bereits in der Einleitung beschreiben, benötigen wir für unseren Anwendungsfall ein genaues sowie performantes Modell, um später eine aussagekräftige und statistisch relevante Live-Erkennung zu ermöglichen. Das bedeutet, dass wir uns bei der Auswahl eines geeigneten Modells auf geringe Speed-Werte und zugleich hohe mAP (*mean Average Precision*) fokussieren. Aus diesem Grund haben wir uns die in Tabelle 3 aufgelisteten Modelle, sortiert nach ihrer Genauigkeit, als Kandidaten für das Training herausgesucht.

Pos	Name	Datenset	Speed (ms)	mAP
1	facessd_mobilenet_v2	Open-Images	20	73 (faces)
2	ssd_mobilenetv2_oidv4	Open-Images	89	36
3	ssd_mobilenet_v1_fpn	COCO	56	32
4	ssd_mobilenet dsp	COCO	12.3	28.9

¹⁶ <https://cocodataset.org/>

¹⁷ <https://storage.googleapis.com/openimages/web/index.html>

5	ssdlite_mobilenet v2	COCO	27	22
---	----------------------	------	----	----

Tabelle 3 Übersicht Modelle

Unsere Auswahl begründen wir weiterhin wie folgt:

1. facessd_mobilenet_v2

Das facessd_mobilenet_v2 Modell hat eine sehr hohe Genauigkeitsrate von 73 mAP bei der Erkennung von Gesichtern und ist zudem mit einer Geschwindigkeit von 20 ms besonders performant. Es wurde anhand des Open-Images Dataset in Version 4 trainiert. Die Annotation erfolgte nur anhand der Kategorie *face*. Da wir in gewisser Maßen auch Gesichter erkennen möchten, haben wir dieses Modell ausgewählt.

2. ssd_mobilenetv2_oidv4

Auch, wenn das ssd_mobilenetv2_oidv4 Modell mit einer Geschwindigkeit von 89 ms im Vergleich zu den anderen ausgewählten Modellen langsam scheint, kann es mit einer guten Erkennungsrate von 36 mAP überzeugen. Auch, wenn die Geschwindigkeit sehr langsam zu sein scheint, ist es im Vergleich mit den anderen Modellen des *Tensorflow model zoo* aufgrund der *mobilenet* Optimierung eines der schnelleren.

3. ssd_mobilenet_v1_fpn

Mit einem Geschwindigkeitswert von 56 ms ist es zwar im Vergleich zum ssdlite_mobilenet_v2 Modell mehr als doppelt so langsam, dafür kann es aber mit einer besseren Erkennungsrate von 32 mAP überzeugen.

4. ssd_mobilenet_dsp

Das ssd_mobilenet_dsp Modell weist mit 12.3 ms die höchste Geschwindigkeit der angebotenen *mobilenet* Modells auf. Mit einer Erkennungsrate von 28.8 mAP kann es zudem mit einer guten Geschwindigkeit/Genauigkeit Ratio überzeugen.

5. ssdlite_mobilenet_v2

Aufgrund der vergleichsweisen schnellen Geschwindigkeit von 27 ms und einem moderaten Genauigkeitswert von 22 haben wir dieses Modell ausgewählt.

Alle Modelle müssen über den jeweiligen Link aus dem *TensorFlow model zoo* heruntergeladen werden. Wir haben jedes Modell in dem gleichnamigen Ordner in unserem Git-Repository¹⁸ abgelegt. Darin sind die für das Transfer-Learning benötigten *model.ckpt* Dateien enthalten.

5.4 Weitere Modell Quellen

Neben den von uns verwendeten *TensorFlow model zoo* bietet TensorFlow eine weitere Modellauswahl in dem GitRepo¹⁹ *TensorFlow 2 detection model zoo* an. Darin werden vier weitere *mobilenet* Modelle angeboten, welche auf Grundlage des COCO Datasets trainiert wurden. Im Gegensatz zu den von uns verwendeten Modellen wurden diese mit TensorFlow 2 trainiert. Leider hatten wir technische Herausforderungen beim Transfer-Learning unter TensorFlow 2, sodass wir diese Modelle nicht betrachtet haben. Die Geschwindigkeits- und Genauigkeitswerte übertreffen jedoch die von uns in Abschnitt 5.3 beschriebenen Modellen nicht. Es wurden Maximalwerte von 48 ms bei 29 mAP erreicht.

¹⁸ Git Repository: 5. Modellauswahl/Modelle

¹⁹https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

6 Trainingsvorbereitung

Nachdem wir die Modelle ausgewählt haben sind wir für das Training entsprechend der TensorFlow Dokumentation²⁰ vorgegangen. Die Schritte werden in diesem Abschnitt genauer beschrieben.

6.1 Split Data

Damit die Bilder für das Training sowie eine Evaluation genutzt werden können, wurde eine zufällige Aufteilung in 75% Trainingsdaten (3157 Bilder) und 25% Testdaten (1100 Bilder) mithilfe eines Python Script²¹ durchgeführt. Zusätzlich wurde für das Trainings- und Testset jeweils eine CSV Datei erstellt²², welche für jedes Bild die entsprechenden Notationen samt Koordinaten und Bezeichnung auflistet. Ein Ausschnitt einer dieser Dateien ist in Abbildung 6 dargestellt.

filename	width	height	class	xmin	ymin	xmax	ymax
Bild_18_rotated_-20_to_-7.jpg	300	168	mask_weared_incorrect	83	1	299	168
Bild_3_zoom.jpg	227	222	mask_weared_incorrect	63	87	116	154
makssksksss618_rotated_7_to_20.jpg	301	400	without_mask	101	166	216	306
makssksksss528.jpg	301	400	without_mask	43	169	149	308
makssksksss272.jpg	275	400	mask_weared_incorrect	48	107	218	304
makssksksss108_flip.jpg	400	225	with_mask	220	29	245	51
makssksksss108_flip.jpg	400	225	with_mask	172	35	194	58
makssksksss108_flip.jpg	400	225	with_mask	54	67	68	81
makssksksss108_flip.jpg	400	225	with_mask	24	68	38	82
makssksksss108_flip.jpg	400	225	without_mask	122	72	133	83

Abbildung 6 CSV-Ausschnitt eines Datensatz

Die CSV wird benötigt, um im Anschluss eine TFRecord Datei für das Training zu erstellen. Das Ergebnis der vorab durchgeführten Schritte ist in dem Git-Repository²³ abgelegt.

6.2 Generierung der TFRecord Dateien

Beim Training mit großen Datenmengen empfiehlt TensorFlow²⁴ die Trainings- und Testdaten jeweils in TFRecord Dateien abzulegen. Dabei handelt es sich um ein Datenformat, welches die Bilder und Annotationen in Binär-Form enthält. Binäre Daten nehmen weniger Platz auf der Festplatte ein, benötigen weniger Zeit zum Kopieren und können viel effizienter von der Festplatte gelesen werden. Dadurch erhöht man die Geschwindigkeit beim Trainieren des Modells. Um die Erstellung dieser TFRecord Files für unsere Test- und Trainingsdaten zu realisieren, haben wir das Script `generate_tfrecord.py` des Git-Repositories `raccoon_dataset`²⁵ genutzt und es an unsere Kategorien `with_mask`, `without_mask` und `mask_weared_incorrect` angepasst (siehe 4.2.1 `generate_tfrecord.py`²⁶). Da die einzelnen TFRecord Files die Größe von 100MB überschreiten, konnten wir diese leider nicht in GitHub hochladen.

6.3 Erstellung der labelmap

Das zukünftige neuronale Netz gibt im Outputlayer die erkannten Kategorien immer nur als ganzzahligen Wert aus. Damit dieser später einfacher interpretiert werden kann, haben wir eine labelmap Datei²⁷ erstellt, welche jeder ausgegebenen Zahl die jeweilige ausgeschriebene

²⁰ https://www.tensorflow.org/lite/models/object_detection/overview

²¹ Git Repository: 6. Trainingsvorbereitung/6.1 split_Dataset.py

²² Git Repository: 6. Trainingsvorbereitung/6.1 generate_tfrecord_csv.py

²³ Git Repository: 3. Datenbeschaffung/images

²⁴ https://www.tensorflow.org/tutorials/load_data/tfrecord

²⁵ https://github.com/datiran/raccoon_dataset

²⁶ Git Repository: 6. Trainingsvorbereitung/6.2 generate_tfrecord.py

²⁷ Git Repository: 6. Trainingsvorbereitung/labelMap/labelmap.pbtxt

Kategorie zuweist. Die ganzzahligen Werte müssen dabei denen der in Abschnitt 6.2 angepassten generate_tfrecord.py Datei entsprechen. Die labelmap Datei wird dabei in folgender Struktur pro Kategorie aufgebaut:

```
item {
    id: 1
    name: 'with_mask'
}
```

6.4 Trainingskonfiguration der Modelle

TensorFlow stellt für alle im *TensorFlow model zoo* aufgelisteten Modelle eine Konfigurationsdatei über das *models* Git-Repository²⁸ zur Verfügung. In diesen Konfigurationsdateien sind bereits Parametereinstellungen, wie learning rate, box_coder und image-resizer für das Transfer-Learning hinterlegt. Dennoch müssen weitere Anpassungen, in Abhängigkeit des neu zu trainierenden Anwendungsfalls, hinterlegt werden. Die in Tabelle 4 aufgelisteten Anpassungen haben wir für alle Konfigurationsdateien gleich getroffen, um diese nach dem Training besser vergleichen zu können.

Parameter	Value	Beschreibung
num_classes	3	Drückt die Anzahl der zu klassifizierenden Kategorien aus (with_mask, without_mask, mask_weared_incorrect).
data_augmentation_options	NONE	Da wir bereits Image-Augmentation durchgeführt haben (siehe Abschnitt 3.2) wird dieser Parameter in jeder Konfiguration leer gelassen.
batch_size	24	Bestimmt die Anzahl an Bildern, welche bei jedem Trainings-Step betrachtet werden.
fine_tune_checkpoint	Pfad zu model.ckpt	Der absolute Pfad zu der model.ckpt Datei des jeweiligen Modells.
Train input_path	Pfad zu train.record	Der absolute Pfad zu der zuvor erstellten train.record Datei.
Train label_map_path	Pfad zu labelmap.pbtxt	Der absolute Pfad zu der zuvor erstellten labelmap.pbtxt Datei.
Eval input_path	Pfad zu test.record	Der absolute Pfad zu der zuvor erstellten test.record Datei.
Eval label_map_path	Pfad zu labelmap.pbtxt	Der absolute Pfad zu der zuvor erstellten labelmap.pbtxt Datei.
metrics_set	NONE	Damit alle Modelle gleich evaluiert werden, wird keine spezielle Metrik hinterlegt.
num_examples	1100	Bestimmt die Anzahl der Testdaten.

Tabelle 4 Konfigurationsparameter

Alle Konfigurationsdateien sind beispielhaft in unserem Git-Repository hinterlegt²⁹.

7 Training

Nachdem alle Trainingsvorbereitungen abgeschlossen waren, wurde der Trainingsprozess gestartet. In diesem Kapitel wird der Trainingsablauf sowie die Evaluationsergebnisse dokumentiert.

²⁸ https://github.com/tensorflow/models/tree/master/research/object_detection/samples/configs

²⁹ Git Repository: 6. Trainingsvorbereitung/Config

7.1 Trainingsprozess

Während des ersten Trainingsdurchlaufs mussten wir feststellen, dass der Prozess auf dem lokalen Gerät sehr zeitintensiv ist. Aus diesem Grund haben wir den Trainingsprozess auf Google Colaboratory³⁰ durchgeführt. Google bietet über seine Plattform die Möglichkeit Trainings mithilfe einer freien Tesla K80 GPU durchzuführen. Dadurch konnten wir die Trainingszeit um bis zu 66% reduzieren.

Für das Training haben wir die Jupyter Notebook Datei `training.ipynb`³¹ erstellt, welche in Google Drive hochgeladen und mit Google Colaboratory geöffnet werden muss. In dem Skript wird eine Verbindung zu dem eigenen Google Drive aufgebaut und das TensorFlow *models* Repository heruntergeladen und installiert. Sobald die Installation abgeschlossen ist, muss in den Pfad `content/drive/My Drive/models/research/object_detection` folgender Inhalt eingefügt werden:

- ein Ordner mit dem Namen *training*, welcher die zuvor erstellte `labelmap.pbtxt`³² und entsprechende Konfigurationsdatei³³ enthält
- ein Pre-Trained Modell³⁴ mit entsprechenden `model.ckpt` Dateien
- die in Abschnitt 6.2 erstellten TFRecord Dateien für das Trainings- und Testset

Im Anschluss wird das Training mithilfe der `train.py` Datei³⁵ aus dem *models* Git-Repository gestartet. Wie in Abbildung 7 dargestellt, wird über die Konsole während des Trainingsprozesses dauerhaft der aktuelle Trainingsstand angezeigt.

```
I0712 11:04:56.370645 4521987520 learning.py:512] global step 4547: loss = 1.6233 (19.122 sec/step)
INFO:tensorflow:global step 4548: loss = 1.8049 (17.134 sec/step)
I0712 11:05:13.505404 4521987520 learning.py:512] global step 4548: loss = 1.8049 (17.134 sec/step)
INFO:tensorflow:global step 4549: loss = 2.0158 (14.873 sec/step)
I0712 11:05:28.379426 4521987520 learning.py:512] global step 4549: loss = 2.0158 (14.873 sec/step)
INFO:tensorflow:global step 4550: loss = 1.6992 (21.003 sec/step)
I0712 11:05:49.387104 4521987520 learning.py:512] global step 4550: loss = 1.6992 (21.003 sec/step)
INFO:tensorflow:global step 4551: loss = 1.6511 (14.235 sec/step)
I0712 11:06:03.625114 4521987520 learning.py:512] global step 4551: loss = 1.6511 (14.235 sec/step)
INFO:tensorflow:global step 4552: loss = 1.8805 (16.969 sec/step)
I0712 11:06:20.594596 4521987520 learning.py:512] global step 4552: loss = 1.8805 (16.969 sec/step)
INFO:tensorflow:global step 4553: loss = 2.0955 (11.306 sec/step)
I0712 11:06:31.901638 4521987520 learning.py:512] global step 4553: loss = 2.0955 (11.306 sec/step)
INFO:tensorflow:global step 4554: loss = 1.6866 (10.324 sec/step)
I0712 11:06:42.226336 4521987520 learning.py:512] global step 4554: loss = 1.6866 (10.324 sec/step)
INFO:tensorflow:Recording summary at step 4554.
I0712 11:06:45.671527 123145421189120 supervisor.py:1050] Recording summary at step 4554.
```

Abbildung 7 Ausschnitt Trainingsprozess Konsole

Zu sehen ist der aktuelle *Step*, welcher jeweils, wie in Abschnitt 6.4 beschrieben, aus einer *batch size* von 24 Bildern, der *Training Loss* Wert sowie die benötigte Berechnungszeit pro *Step* in Sekunden besteht. Außerdem wird regelmäßig ein *Record summary* erstellt, welches den aktuellen Trainingsstand als `model.ckpt` Datei abspeichert.

7.2 Evaluation des Trainingsprozesses

Um während des laufenden Trainingsprozesses die Performance des aktuellen Trainingsstandes zu prüfen, haben wir parallel zu jedem Training eine Evaluation durchgeführt. Dafür haben wir

³⁰ <https://colab.research.google.com/notebooks/intro.ipynb#>

³¹ Git Repository: 7. Training/7.1 training.ipynb

³² Git Repository: 6. Trainingsvorbereitung/labelMap/labelmap.pbtxt

³³ Git Repository: 6. Trainingsvorbereitung/Config

³⁴ Git Repository: 5. Modellauswahl/Modelle

³⁵ https://github.com/tensorflow/models/blob/master/research/object_detection/train.py

die Jupyter Notebook Datei elavModell.ipynb³⁶ erstellt, welche in Google Drive hochgeladen und mit Google Colaboratory geöffnet werden muss. In dem Skript wird eine Verbindung zu dem eigenen Google Drive aufgebaut und das Setup für die laufende Maschine durchgeführt.

Im Anschluss wird die Evaluation mithilfe der *eval.py* Datei³⁷ aus dem models Git-Repository gestartet. Wie in Abbildung 8 dargestellt, wird über die Konsole während des Evaluationsprozesses dauerhaft der aktuelle Evaluationsstand angezeigt.

```

INFO:tensorflow:Running eval ops batch 100/1100
I0712 11:43:28.662868 4551499200 eval_util.py:339] Running eval ops batch 100/1100
INFO:tensorflow:Running eval ops batch 200/1100
I0712 11:43:46.556197 4551499200 eval_util.py:339] Running eval ops batch 200/1100
INFO:tensorflow:Running eval ops batch 300/1100
I0712 11:44:02.364687 4551499200 eval_util.py:339] Running eval ops batch 300/1100
INFO:tensorflow:Running eval ops batch 400/1100
I0712 11:44:19.399969 4551499200 eval_util.py:339] Running eval ops batch 400/1100
I0712 11:44:38.084619 4551499200 evaluator.py:237] Skipping image
I0712 11:44:31.366426 4551499200 evaluator.py:237] Skipping image
I0712 11:44:31.532521 4551499200 evaluator.py:237] Skipping image
INFO:tensorflow:Running eval ops batch 500/1100
I0712 11:44:33.699983 4551499200 eval_util.py:339] Running eval ops batch 500/1100
I0712 11:44:35.218318 4551499200 evaluator.py:237] Skipping image
I0712 11:44:36.100279 4551499200 evaluator.py:237] Skipping image
I0712 11:44:38.837019 4551499200 evaluator.py:237] Skipping image
INFO:tensorflow:Running eval ops batch 600/1100
I0712 11:44:58.936336 4551499200 eval_util.py:339] Running eval ops batch 600/1100
INFO:tensorflow:Running eval ops batch 700/1100
I0712 11:45:07.613157 4551499200 eval_util.py:339] Running eval ops batch 700/1100
I0712 11:45:18.789617 4551499200 evaluator.py:237] Skipping image
I0712 11:45:16.146353 4551499200 evaluator.py:237] Skipping image
I0712 11:45:18.132858 4551499200 evaluator.py:237] Skipping image
INFO:tensorflow:Running eval ops batch 800/1100
I0712 11:45:23.263968 4551499200 eval_util.py:339] Running eval ops batch 800/1100
I0712 11:45:27.676577 4551499200 evaluator.py:237] Skipping image
INFO:tensorflow:Running eval ops batch 900/1100
I0712 11:45:39.480857 4551499200 eval_util.py:339] Running eval ops batch 900/1100
I0712 11:45:54.505345 4551499200 evaluator.py:237] Skipping image
INFO:tensorflow:Running eval ops batch 1000/1100
I0712 11:45:56.279483 4551499200 eval_util.py:339] Running eval ops batch 1000/1100
I0712 11:45:58.299623 4551499200 object_detection_evaluation.py:335] image b'makssssksss828.jpg' does not have groundtruth difficult flag specified
INFO:tensorflow:Running eval ops batch 1100/1100
I0712 11:46:12.271390 4551499200 eval_util.py:339] Running eval ops batch 1100/1100
INFO:tensorflow:Running eval batches done.
I0712 11:46:12.459120 4551499200 eval_util.py:373] Running eval batches done.

INFO:tensorflow:# success: 1089
I0712 11:46:12.459295 4551499200 eval_util.py:378] # success: 1089
INFO:tensorflow:# skipped: 11
I0712 11:46:12.460684 4551499200 eval_util.py:379] # skipped: 11
I0712 11:46:13.083056 4551499200 object_detection_evaluation.py:1318] average_precision: 0.126418
I0712 11:46:13.394686 4551499200 object_detection_evaluation.py:1318] average_precision: 0.096869
I0712 11:46:13.740676 4551499200 object_detection_evaluation.py:1318] average_precision: 0.231610
INFO:tensorflow:Writing metrics to tf summary.
I0712 11:46:15.710659 4551499200 eval_util.py:87] Writing metrics to tf summary.
INFO:tensorflow:Losses/Loss/classification_loss: 7.560480
I0712 11:46:15.716292 4551499200 eval_util.py:94] Losses/Loss/classification_loss: 7.560480
INFO:tensorflow:Losses/Loss/localization_loss: 3.094835
I0712 11:46:15.719660 4551499200 eval_util.py:94] Losses/Loss/localization_loss: 3.094835
INFO:tensorflow:PascalBoxes_PerformanceByCategory/AP@0.5IOU/mask_weared_incorrect: 0.231610
I0712 11:46:15.720271 4551499200 eval_util.py:94] PascalBoxes_PerformanceByCategory/AP@0.5IOU/mask_weared_incorrect: 0.231610
INFO:tensorflow:PascalBoxes_PerformanceByCategory/AP@0.5IOU/with_mask: 0.126418
I0712 11:46:15.720999 4551499200 eval_util.py:94] PascalBoxes_PerformanceByCategory/AP@0.5IOU/with_mask: 0.126418
INFO:tensorflow:PascalBoxes_PerformanceByCategory/AP@0.5IOU/without_mask: 0.096869
I0712 11:46:15.721582 4551499200 eval_util.py:94] PascalBoxes_PerformanceByCategory/AP@0.5IOU/without_mask: 0.096869
INFO:tensorflow:PascalBoxes_Precision/mAP@0.5IOU: 0.151366
I0712 11:46:15.724970 4551499200 eval_util.py:94] PascalBoxes_Precision/mAP@0.5IOU: 0.151366
INFO:tensorflow:Metrics written to tf summary.
I0712 11:46:15.725505 4551499200 eval_util.py:95] Metrics written to tf summary.
INFO:tensorflow:Starting evaluation at 2020-07-12-09:47:44

```

Abbildung 8 Ausschnitt Evaluation Konsole

Alle 300 Sekunden wird überprüft, ob der Trainingsprozess eine aktuelle *Record summary* abgespeichert hat. Wenn dies der Fall ist, wird eine Evaluation anhand des evaluations *batch* gestartet, welches, wie in Abschnitt 6.4 durch den Parameter *num_examples* bestimmt, alle 1100 Testbilder beinhaltet und durchläuft. Zusätzlich wird die Genauigkeit der Kategorien *mask_weared_incorrect*, *with_mask* und *without_mask* sowie deren Durchschnitt nach jedem Evaluationsdurchgang ausgegeben. Dafür wird die Anzahl der richtig gefundenen durch die Anzahl der zu findenden Bounding Boxes pro Kategorie berechnet. Die berechneten Metriken werden im Anschluss abgespeichert, um später einen Verlauf der Trainingsevaluation auszugeben.

7.3 Evaluationsauswertung

Um die in Abschnitt 7.2 gespeicherten Metriken auszulesen, bietet TensorFlow ein Dashboard names TensorBoard³⁸ an. Um dieses mit unseren Evaluationsmetriken zu füttern, haben wir die

³⁶ Git Repository: 7. Training/7.2 elavModell.ipynb

³⁷ https://github.com/tensorflow/models/blob/master/research/object_detection/legacy/eval.py

³⁸ <https://www.tensorflow.org/tensorboard>

Jupyter Notebook Datei dashboard.ipynb³⁹ erstellt, welche in Google Drive hochgeladen und mit Google Colaboratory geöffnet werden muss. In dem Skript wird eine Verbindung zu dem eigenen Google Drive aufgebaut und das TensorBoard gestartet. Daraufhin wird eine Weboberfläche geöffnet, auf welcher, wie in Abbildung 9 beispielhaft dargestellt, ein Diagramm für jede Metrik dargestellt wird.

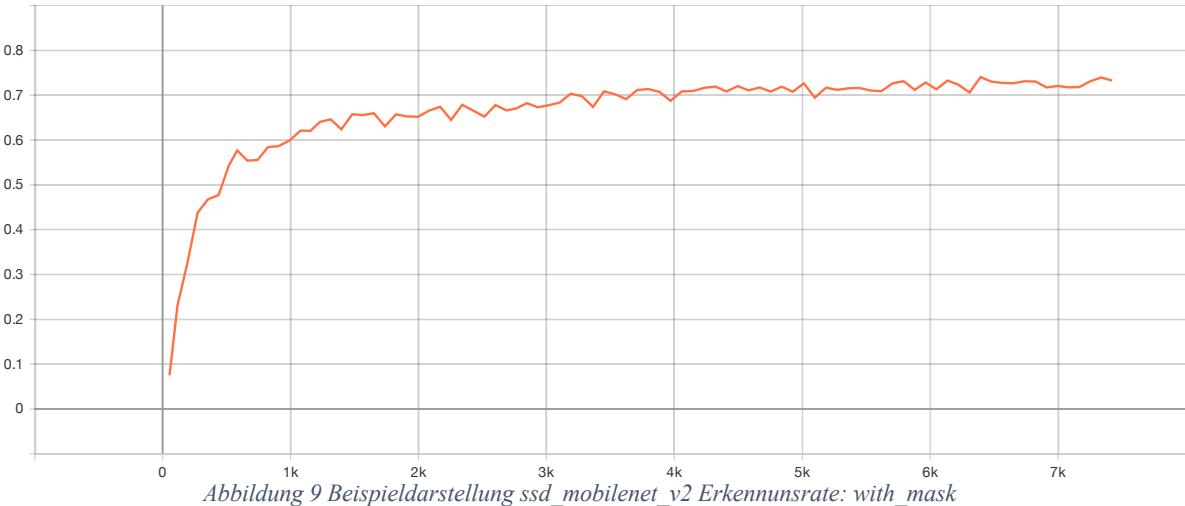


Abbildung 9 Beispieldarstellung `ssd_mobilenet_v2` Erkennungsrate: `with_mask`

Auf der X-Achse werden die Steps und auf der Y-Achse die Erkennungsgenauigkeit in Prozent dargestellt. Zusätzlich werden auf der Weboberfläche 10 Bilder aus dem Trainingsdataset abgebildet. Diese werden, wie in Abbildung 10 erkennbar, mit jedem Evaluationsschritt durch den aktuellen Trainingsstand bewertet.

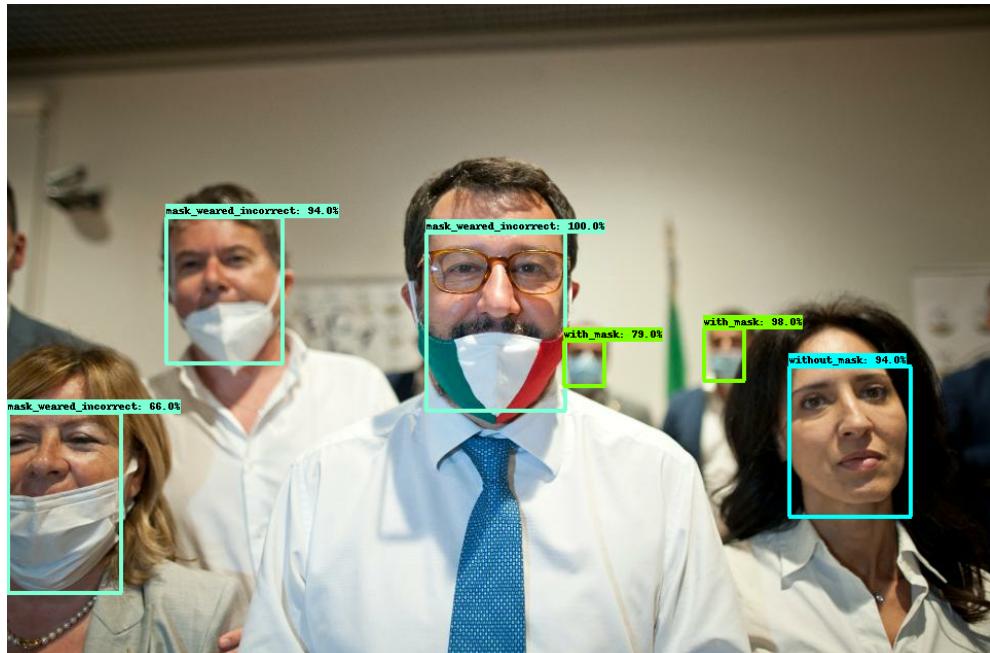


Abbildung 10 Beispieldarstellung `ssd_mobilenet_v2` Evaluationsimage Step 7000

Die erkannten Kategorien sind dabei mit farblich unterschiedlich markierten Bounding Boxes dargestellt und beschreiben zudem den erkannten Wahrscheinlichkeitsgrad.

³⁹ Git Repository: 7. Training/7.3 dashboard.ipynb

7.4 Evaluationsergebnisse

Um herauszubekommen, welches der in Abschnitt 5.3 ausgewählten Modelle sich in der Praxis am besten für unseren Anwendungsfall eignet, haben wir mit jedem ein Transfer-Learning Training, wie vorab beschrieben, durchgeführt. Die Anzahl der Trainings-Steps wurde dafür auf 10000 festgelegt. Dies hat den Hintergrund, dass trotz der Nutzung der Tesla GPU ein Training im Schnitt 10 Stunden gedauert hat. Weiterhin wurde die kostenfreie Nutzung der GPU regelmäßig nach einer längeren Nutzungszeit für mehrere Stunden deaktiviert.

Folgend stellen wir die Ergebnisse für jedes Modell vor. Diese beinhalten jeweils die Gegenüberstellung des Trainings und Validation Loss sowie der einzelnen Kategorie-Erkennungsraten pro Step. Zusätzlich beschreiben wir die individuellen Konfigurationsparameter, welche für jedes Training genutzt wurden. Dabei haben wir uns an die default-Werte der in Abschnitt 6.4 beschriebenen Konfigurationsdateien gehalten, da uns bei diesen ersten Transfer-Learning Versuchen die Erfahrungswerte für spezifische Einstellungen fehlen.

facessd_mobilenet_v2

Das erste Transfer-Learning haben wir mithilfe des facessd_mobilenet_v2 Modells erstellt, wobei die default Konfigurationseinstellungen wie folgt genutzt wurden.

Parameter	Werte	Beschreibung
fixed_shape_resizer	320x320	Die Größe (width x height) in die die eingelesenen Bilder umgewandelt werden
initial_learning_rate	0.004	Die Größe des Gewichtungs-Anpassungsfaktors während des gesamten Trainings

Tabelle 5 facessd_mobilenet_V2 Konfiguration

Das Resultat nach 10000 Trainings-Steps ist in Abbildung 11 dargestellt. Man kann eine stetige Erkennungsgenauigkeit der einzelnen Kategorien im Laufe des Trainings erkennen, wobei das Endresultat bei weitem nicht unserem in der Einleitung beschreibenen Ziel von 95% entspricht.

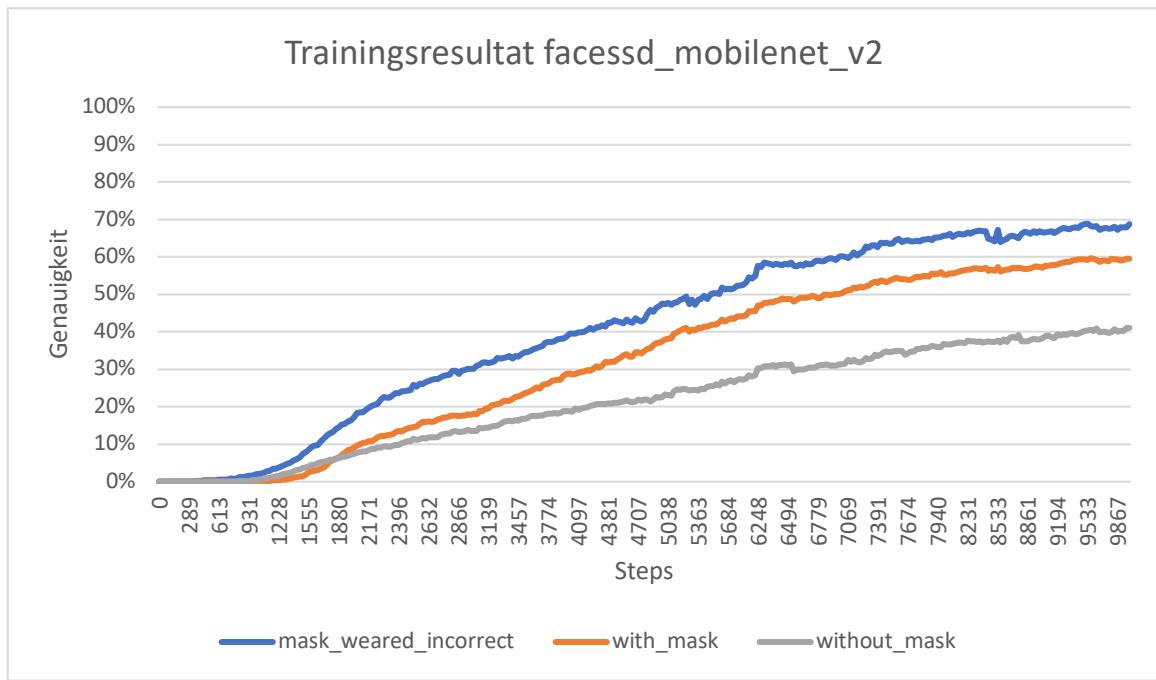


Abbildung 11 Trainingsresultat facessd_mobilenet_v2

Die Genauigkeiten der Kategorien mask_weared_incorrect und with_mask haben sich dabei gegenüber der Kategorie without_mask etwas ab.

Auf der in Abbildung 12 dargestellten Gegenüberstellung des Trainings und Validation Loss ist zu erkennen, dass ab Step 4700 das Validation Loss steigt und sich dadurch vom Training Loss abhebt. Gemäß TensorFlow⁴⁰ deutet dies auf Overfitting hin.

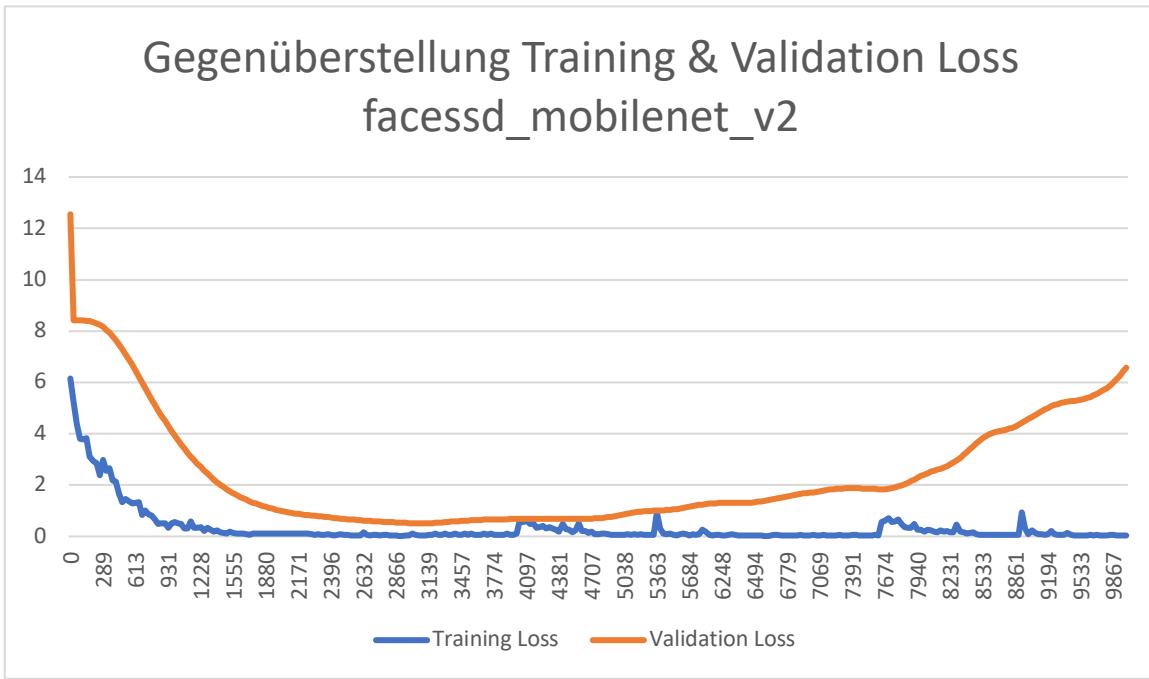


Abbildung 12 Gegenüberstellung Training & Validation Loss facessd_mobilenet_v2

Dementsprechend hätten wir das Training bereits ab Step 4700 abbrechen können, wobei dadurch die Erkennungsrate der einzelnen Kategorien noch niedriger ausgefallen wäre.

ssd_mobilenetv2_oidv4

Das mithilfe des ssd_mobilenetv2_oidv4 Modells durchgeführte Training, haben wir mit den folgenden default Konfigurationseinstellungen durchgeführt.

Parameter	Werte	Beschreibung
fixed_shape_resizer	300x300	Die Größe (width x height) in die die eingelesenen Bilder umgewandelt werden
initial_learning_rate	0.0008	Die Größe des Gewichtungs-Anpassungsfaktors während des gesamten Trainings

Tabelle 6 ssd_mobilenetv2_oidv4 Konfiguration

Auffällig hierbei ist, dass die *initial_learning_rate* mit 0.0008, im Gegensatz zu dem facessd_mobilenet_v2 Training, viel geringer gewählt wurde.

Das Resultat nach 10000 Trainings-Steps ist in Abbildung 13 dargestellt. Man kann eine stetige Verbesserung der Erkennungsgenauigkeit der einzelnen Kategorien im Laufe des Trainings erkennen, wobei das Endresultat bereits etwas besser als das Trainingsresultat des facessd_mobilenet_v2 ausgefallen ist.

⁴⁰ <https://www.youtube.com/watch?v=GMrTBtzJkCg>

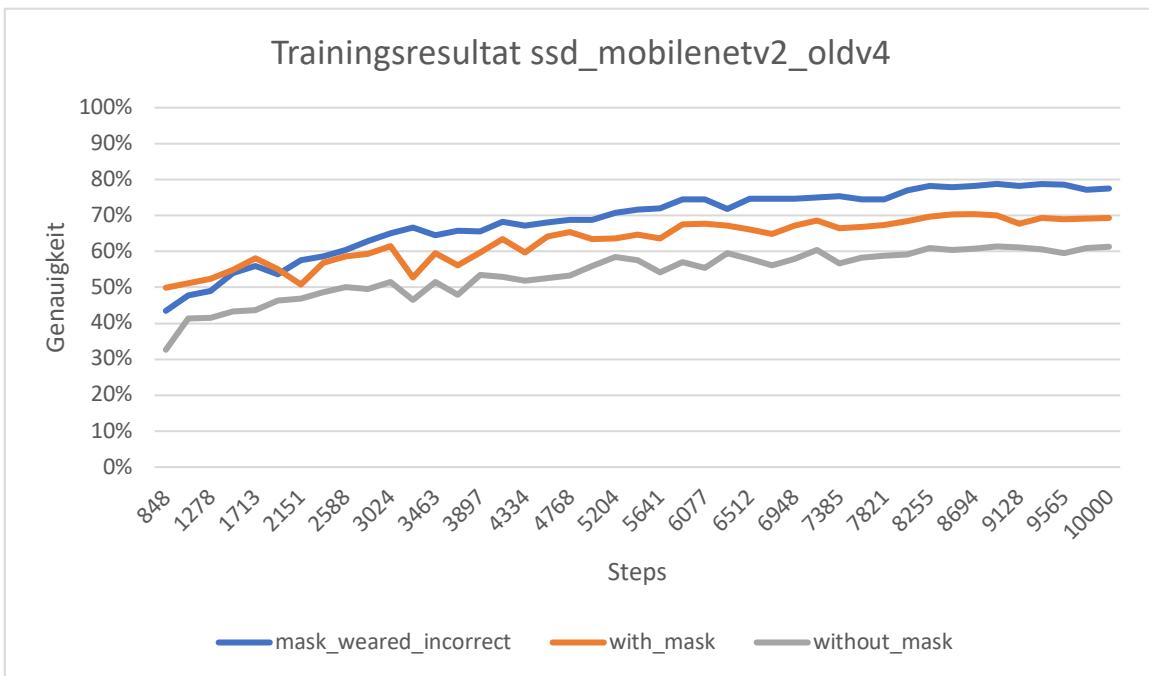


Abbildung 13 Trainingsresultat ssd_mobilenetv2_oldv4

Aufgrund eines Konfigurationsfehlers bei der Evaluation hat die Speicherung der Werte erst ab Step 848 begonnen.

Auf der in Abbildung 14 dargestellten Gegenüberstellung des Trainings- und Validation-Loss ist ein dauerhafter Abfall der beiden Werte zu erkennen. Im Gegensatz zu dem vorab durchgeföhrten Training ist hierbei kein Anzeichen von Overfitting sichtbar.

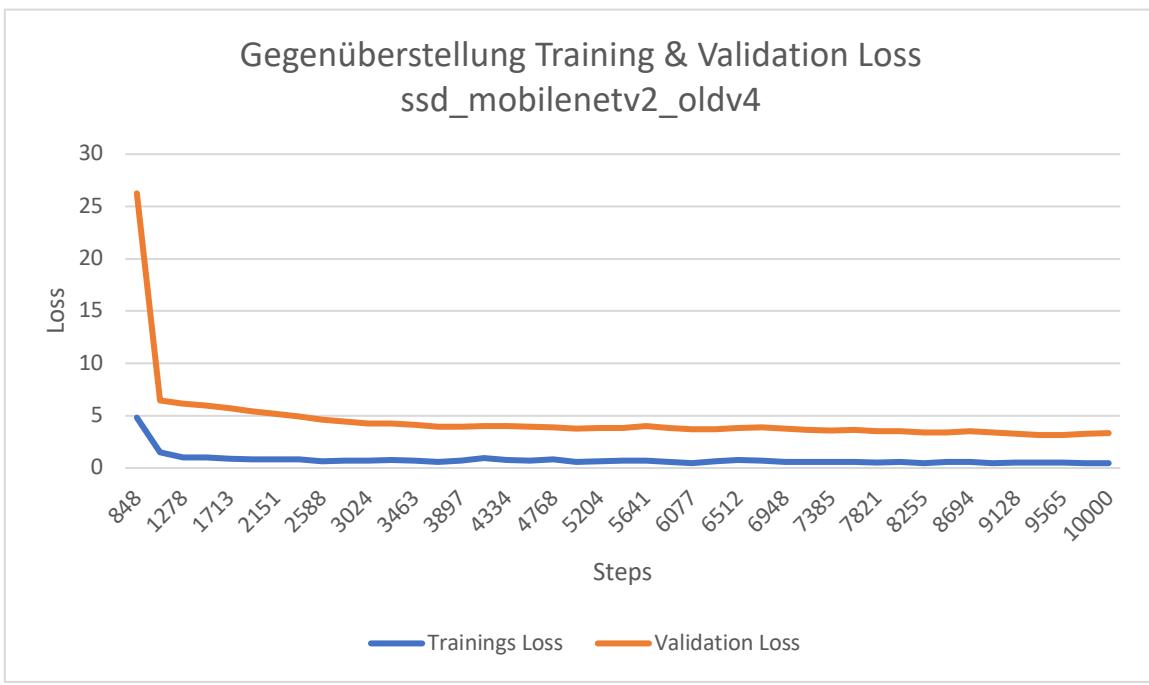


Abbildung 14 Gegenüberstellung Training & Validation Loss ssd_mobilenetv2_oldv4

ssd_mobilenet_v1_fpn

Das mithilfe des ssd_mobilenet_v1_fpn Modells durchgeführte Training, haben wir mit den folgenden default Konfigurationseinstellungen durchgeführt.

Parameter	Werte	Beschreibung
fixed_shape_resizer	640x640	Die Größe (width x height) in die die eingelesenen Bilder umgewandelt werden
warmup_learning_rate	0,013333	Die Größe des Gewichtungs-Anpassungsfaktors während der ersten 2000 Steps
learning_rate_base	0.04	Die Größe des Gewichtungs-Anpassungsfaktors während der letzten 8000 Steps

Tabelle 7 ssd_mobilenet_v1_fpn Konfiguration

Eine Besonderheit dieser Konfiguration ist die Veränderung der Learning-Rate während des Trainings.

Das Resultat nach 10000 Trainings-Steps ist in Abbildung 15 dargestellt. Die Kategorien with_mask und mask_weared_incorrect erreichen dabei eine Erkennungsgenauigkeit von ca. 81%. Auch die Kategorie without_mask erreicht mit ca. 70% den bisher höchsten Wert.

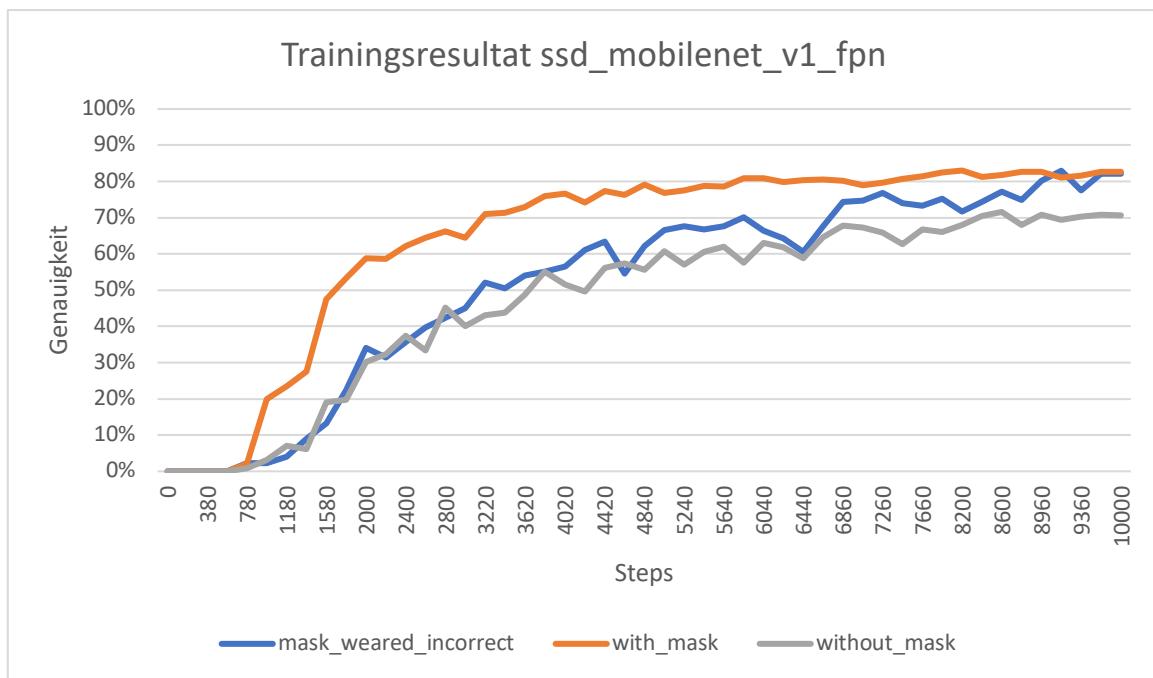


Abbildung 15 Trainingsresultat ssd_mobilenet_v1_fpn

Auf der in Abbildung 16 dargestellten Gegenüberstellung des Trainings- und Validation-Loss ist ein dauerhafter Abfall der beiden Werte zu erkennen. Overfitting ist dabei nicht erkennbar.

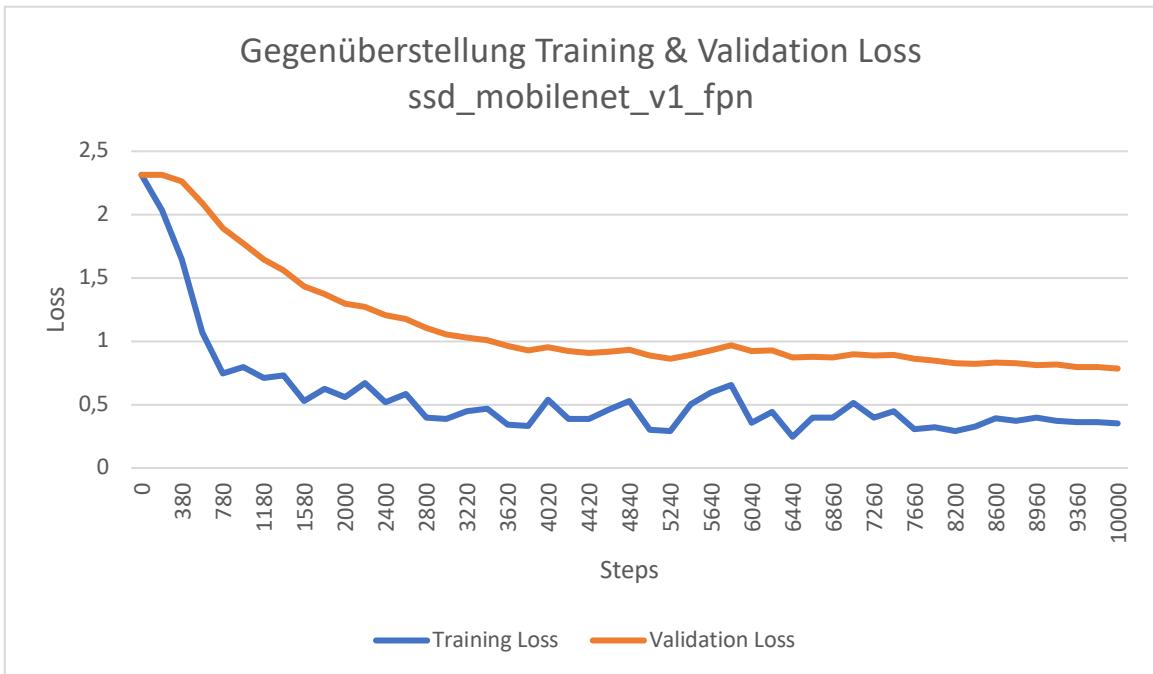


Abbildung 16 Gegenüberstellung Training & Validation Loss ssd_mobilenet_v1_fpn

ssd_mobilenet_dsp

Ein Transfer-Learning Training mithilfe des ssd_mobilenet_dsp Modells war leider nicht möglich. Nach ca. 300 Training-Steps haben wir immer wieder die in Abbildung 17 dargestellte Fehlermeldung erhalten.

```

File "/tensorflow-1.15.2/python3.6/tensorflow_core/python/training/_monitored_session.py", line 1345, in run
    return self._sess.run(*args, **kwargs)
File "/tensorflow-1.15.2/python3.6/tensorflow_core/python/training/_monitored_session.py", line 1418, in run
    run_metadata=run_metadata)
File "/tensorflow-1.15.2/python3.6/tensorflow_core/python/training/_monitored_session.py", line 1176, in run
    return self._sess.run(*args, **kwargs)
File "/tensorflow-1.15.2/python3.6/tensorflow_core/python/client/session.py", line 956, in run
    run_metadata_ptr)
File "/tensorflow-1.15.2/python3.6/tensorflow_core/python/client/session.py", line 1180, in _run
    feed_dict_tensor, options, run_metadata)
File "/tensorflow-1.15.2/python3.6/tensorflow_core/python/client/session.py", line 1359, in _do_run
    run_metadata)
File "/tensorflow-1.15.2/python3.6/tensorflow_core/python/client/session.py", line 1384, in _do_call
    raise type(e)(node_def, op, message)
tensorflow.python.framework.errors_impl.InvalidArgumentError: assertion failed: [[0.422619045][0.148809522][0.119047619]...] [[0.571428597][0.25][0.226190478]...]
[[{{(node Assert/AssertGuard/Assert)}}]]
[[IteratorGetNext]]
```

Abbildung 17 Fehlermeldung ssd_mobilenet_dsp

Laut der TensorFlow Community könnte es sich bei dem Fehler um ein Problem mit dem Format der Trainingsdaten handeln. Dies konnten wir jedoch nicht identifizieren. Zudem ist der *fixed_shape_resizer* Parameter, wie auch bei dem erfolgreichen Training mithilfe des facessd_mobilenet_v2 Modells, mit 300x300 angegeben.

ssdlite_mobilenet_v2

Das letzte Transfer-Learning haben wir mithilfe des ssdlite_mobilenet_v2 Modells erstellt, wobei die default Konfigurationseinstellungen wie folgt genutzt wurden.

Parameter	Werte	Beschreibung
fixed_shape_resizer	300x300	Die Größe (width x height) in die die eingelesenen Bilder umgewandelt werden
initial_learning_rate	0.004	Die Größe des Gewichtungs-Anpassungsfaktors während des gesamten Trainings

Tabelle 8 ssdlite_mobilenet_v2 Konfiguration

Das Resultat nach 10000 Trainings-Steps ist in Abbildung 18 dargestellt. Die Kategorien mask_weared_incorrect erzieht mit 82% den bisher höchsten erreichten Wert.

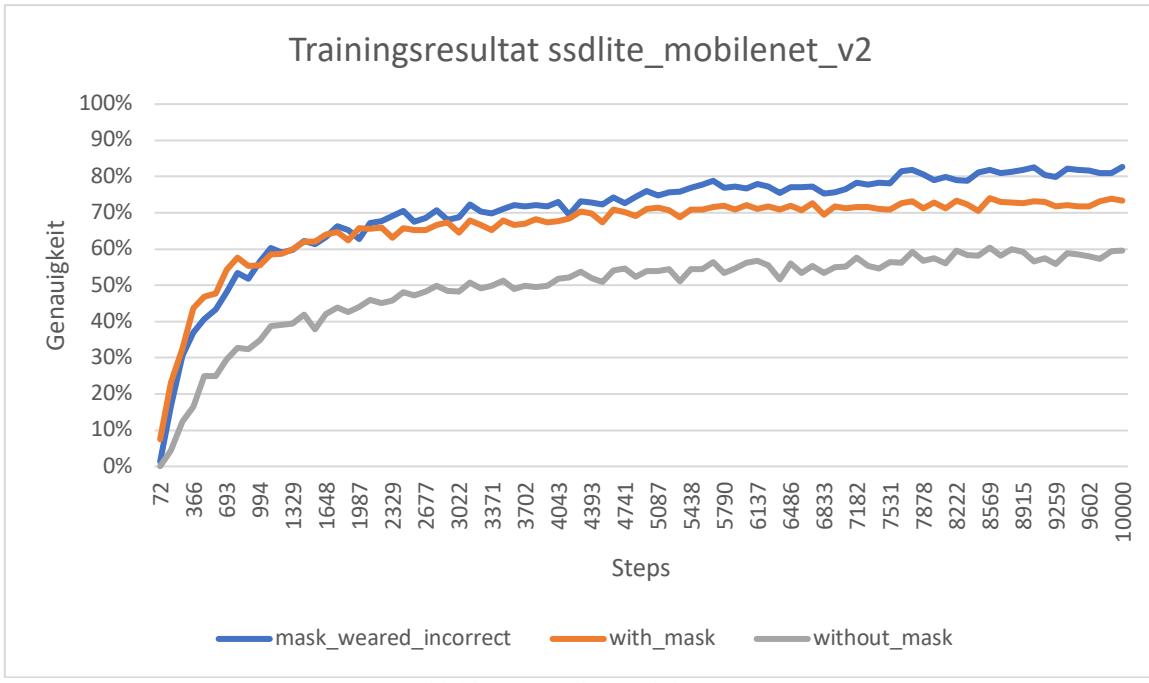


Abbildung 18 ssdlite_mobilenet_v2

Auf der in Abbildung 19 dargestellten Gegenüberstellung des Trainings- und Validation-Loss ist ein dauerhafter Abfall der beiden Werte zu erkennen. Overfitting ist dabei nicht erkennbar.

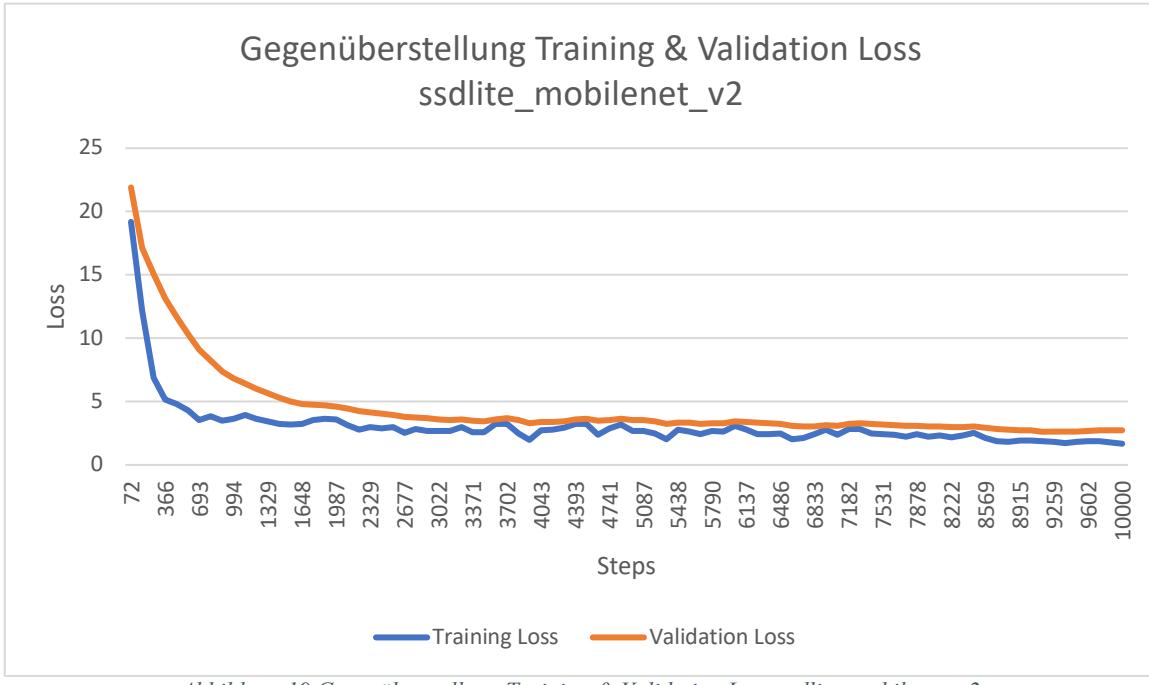


Abbildung 19 Gegenüberstellung Training & Validation Loss ssdlite_mobilenet_v2

Gesamtvergleich

Um das beste Ergebnis auszuwählen, haben wir die Durchschnittsgenauigkeitswerte der drei Kategorien je Modell gegenübergestellt. Das Resultat ist in Abbildung 20 dargestellt.

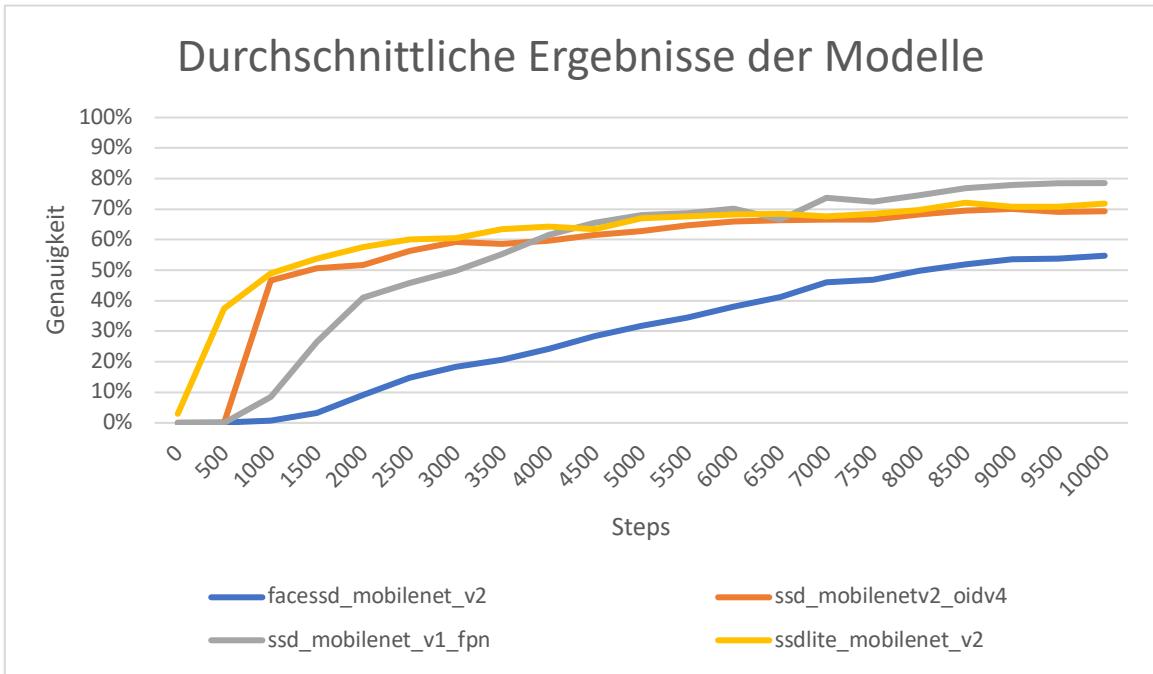


Abbildung 20 Durchschnittliche Ergebnisse der Modelle

Das beste Ergebnis von 78% Erkennungsrate nach 10000 Trainings-Steps konnte mithilfe des *ssd_mobilenet_v1_fpn* Modell erreicht werden. Das in Abschnitt 5.3 als Favorit ausgewählte Modell *facessd_mobilenet_v2* hat hingegen das ungenauste Resultat hervorgebracht. Das könnte damit zusammenhängen, dass dieses auf eine binäre Erkennung (Gesicht oder kein Gesicht) trainiert wurde und daher bei der durchgeführten Multi-Klassifikation schlechter performt.

Auch, wenn wir das angestrebte Ziel von 95% Genauigkeitsrate während des Trainings nicht erzielen konnten, haben wir für die weiteren Schritte das genannte Modell mit der höchsten Erkennungsrate von 78% genutzt.

8 Modell Konvertierung und Optimierung

In diesem Kapitel wird beschrieben, wie wir das trainierte Modell in ein TensorFlow-Lite Modell konvertieren und weiterhin optimiert haben. Dieser Vorgang ist nötig, damit das in Abschnitt 7.4 ausgewählte Modell in unserer IOS App genutzt werden kann. Dabei gehen wir auf das Vorgehen bei der Konvertierung, Optimierung und die Evaluation der Ergebnisse ein.

8.1 Konvertierung

Um das gewählte Modell zu konvertieren muss vorab ein Frozen-Graph erstellt werden. Für die Erstellung des Frozen-Graph haben wir die Jupyter Notebook Datei `create frozen graph.ipynb`⁴¹ erstellt, welche in Google Drive hochgeladen und mit Google Colaboratory geöffnet werden muss. In dem Skript wird eine Verbindung zu dem eigenen Google Drive aufgebaut und das Setup für die laufende Maschine durchgeführt.

Im Anschluss wird die Erstellung des Graphen mithilfe der `export_tflite_ssdlite_graph.py` Datei⁴² aus dem `models` Git-Repository gestartet. Als Übergabeparameter müssen die Pfade der im

⁴¹ Git Repository: 8. Modell Konvertierung und Optimierung/8.1 create frozen graph.ipynb

⁴² https://github.com/tensorflow/models/blob/master/research/object_detection/export_tflite_ssdlite_graph.py

Training verwendeten Konfigurationsdatei sowie des gewünschten trainierten Modellstand übergeben werden. Als Ergebnis erhalten wir den Frozen-Graph unseres Modells⁴³.

Mithilfe des Frozen-Graphen kann im Anschluss das Modell in ein TensorFlow-Lite (tf-lite) umgewandelt werden. Dafür haben wir die Jupyter Notebook Datei *convert_to_tflite.ipynb*⁴⁴ erstellt, welche in Google Drive hochgeladen und mit Google Colaboratory geöffnet werden muss. In dem Skript wird eine Verbindung zu dem eigenen Google Drive aufgebaut und das Setup für die laufende Maschine durchgeführt.

Anschließend erfolgt die Konvertierung für die wir die in der TensorFlow Dokumentation vorgegebenen Command line Befehle⁴⁵ befolgt haben. Dabei wird unter anderem der Pfad des Frozen-Graphen, der Input-shape aus der Konfigurationsdatei sowie die Output-Directory der zu erstellenden *tflite* Datei angegeben. Als Ergebnis erhalten wir ein Float32 Modell als tf-lite Datei⁴⁶. Dieses vergleichen wir später in Abschnitt 8.3 mit dem optimierten tf-lite Modell.

8.2 Optimierung

In der TensorFlow Dokumentation⁴⁷ werden die zwei Optimierungsoptionen Quantization und Pruning beschrieben. Folgend gehen wir jeweils auf die beiden Verfahren ein.

Quantization

Die Quantisierung nach dem Training ist eine Konvertierungstechnik, mit der die Modellgröße reduziert und gleichzeitig die Latenz der CPU und des Hardware-Beschleunigers verbessert werden kann. Die Modellgenauigkeit nimmt dabei nur geringfügig ab. In der TensorFlow Dokumentation werden drei verschiedene Quantization Methoden beschrieben⁴⁸, wobei deren Stärken in Abbildung 21 dargestellt werden.

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

Abbildung 21 Quantization Methoden im Vergleich

Die Full Integer quatization trifft unsere Anforderungen am besten, da, wie in der Einleitung beschrieben, das trainierte Modell zukünftig in IOT Geräten eingesetzt werden soll. Durch die Microcontroller und CPU Hardwareunterstützung sowie einer vierfachen Verkleinerung und dreifachen Beschleunigung unseres Modells ist die Optimierung mithilfe der Full integer quatization Technik am sinnvollsten.

Für die Erstellung des optimierten Modells haben wir die Jupyter Notebook Datei *convert_to_full_integer_tflite.ipynb*⁴⁹ erstellt, welche in Google Drive hochgeladen und mit Google Colaboratory geöffnet werden muss. In dem Skript wird eine Verbindung zu dem eigenen Google Drive aufgebaut und das Setup für die laufende Maschine durchgeführt.

⁴³ Git Repository: 8. Modell Konvertierung und Optimierung/Frozen-Graph

⁴⁴ Git Repository: 8. Modell Konvertierung und Optimierung/8.1 convert_to_tflite.ipynb

⁴⁵ <https://www.tensorflow.org/lite/convert/cmdline>

⁴⁶ Git Repository: 8. Modell Konvertierung und Optimierung/tf-lite/model.tflite

⁴⁷ https://www.tensorflow.org/lite/performance/model_optimization

⁴⁸ https://www.tensorflow.org/lite/performance/post_training_quantization

⁴⁹ Git Repository: 8. Modell Konvertierung und Optimierung/8.2 convert_to_full_integer_tflite.ipynb

Im Anschluss erfolgt die Konvertierung, wobei wir uns an die in der TensorFlow Dokumentation vorgegebenen Command line Befehle⁵⁰ gehalten haben. Dabei wird, neben denen in Abschnitt 8.1 beschriebenen Parametern, der Befehl QUANTIZED_UINT8 bei der Erstellung des optimierten Modells angegeben. Als Ergebnis erhalten wir ein Full-Integer Modell als tf-lite Datei⁵¹. Dieses vergleichen wir später in Abschnitt 8.3 mit dem vorab erstellten Float32 tf-lite Modell.

Pruning

Ein weiteres Verfahren zur Optimierung eines Modells ist das Pruning. In der TensorFlow Dokumentation werden die folgenden Effekte beschrieben⁵²:

- Nachteil:
Einzelne Parameter werden innerhalb des Modells entfernt, was eine Auswirkung auf die Erkennungsgenauigkeit hat. Die Latenz (Geschwindigkeit) sowie die tatsächliche Modellgröße ändern sich dabei nicht.
- Vorteil:
Reduzierung der Modell-Download-Größe

Wie in der Einleitung beschrieben ist unser Genauigkeitsanspruch an das Modell sehr hoch, wodurch wir neben der Quantisierung ungern an Erkennungsgenauigkeit verlieren möchten. Weiterhin ist das Modell und die dazugehörige Applikation nicht für den kommerziellen Gebrauch vorgesehen. Öffentliche Einrichtungen wie Museen oder Bahnhöfe sind mittlerweile mit W-Lan ausgestattet, wodurch eine Reduzierung der Modell-Download-Größe für uns nicht weiter interessant ist.

8.3 Ergebnisse

In diesem Abschnitt vergleichen wir die beiden erstellten Modelle. Dafür haben wir die Geschwindigkeit, Größe und Genauigkeit der Modelle gemessen und gegenübergestellt. Das Ergebnis ist in Tabelle 9 dargestellt.

Modell	Größe	Geschwindigkeit	Ø Genauigkeit
Float32	18,6 MB	53 ms	78%
Full-Integer	4,7 MB	40 ms	76,3%

Tabelle 9 Optimierungsresultat

Die Latency (Geschwindigkeit) haben wir mithilfe des TensorFlow Benchmark Tool⁵³ für IOS mit einem iPhone 8 gemessen. Dabei haben wir jeweils die besten Ergebnisse bei der Nutzung von 2 Threads erzielt. Die Genauigkeit wurde auf Grundlage der in Abschnitt 6.1 beschriebenen 1100 Testbildern und eines bestehenden Evaluations-Repository⁵⁴ erstellt.

Durch die Quantisierung konnte die Modell-Größe stark reduziert werden. Weiterhin können wir eine Steigerung der Geschwindigkeit feststellen. Dies macht sich bereits in einem Live-Versuch mit der TensorFlow IOS Object-Detection Demo-App⁵⁵ bemerkbar, da das Modell bei

⁵⁰ <https://www.tensorflow.org/lite/convert/cmdline>

⁵¹ Git Repository: 8. Modell Konvertierung und Optimierung/tf-lite/ model-full-Integer.tflite

⁵² https://www.tensorflow.org/lite/performance/model_optimization#pruning

⁵³ <https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark/ios>

⁵⁴ <https://github.com/cloud-annotations/object-detection-python>

⁵⁵ https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/ios

sich ständig ändernden Umgebungen spürbar schneller reagiert und dadurch genauere Ergebnisse anzeigt. Die durchschnittliche Erkennungsgenauigkeit ist um 1,7% gesunken.

Weiterhin haben wir mit der IOS Entwicklungsumgebung Xcode den Batterieverbrauch und die Nutzung des Arbeitsspeichers anhand eines iPhone 8 verglichen. Die Ergebnisse sind in Tabelle 10 dargestellt.

Modell	Ø Batterieverbrauch	Arbeitsspeichernutzung
Float32	High Energy Impact	196 MB 10%
Full-Integer	Low Energy Impact	111 MB 5,6%

Tabelle 10 Optimierungsresultat Benchmark

Durch die Quantisierung konnte der durchschnittliche Batterieverbrauch und die Nutzung des Arbeitsspeichers entscheidend verbessert werden.

9 App-Entwicklung

In diesem Kapitel beschreiben wir unser Vorgehen bei der IOS App-Entwicklung. Wie in der Einleitung bereits erwähnt, ist unser Ziel, mit der App, über die Kamera, Bilder zu erfassen, um darauf alle Gesichter zu erkennen die den Mund-Nasen-Schutz tragen, nicht tragen oder falsch tragen. Die erkannten Kategorien sollen im Anschluss gezählt und im Verhältnis gegenübergestellt werden. Darüber hinaus soll es die Möglichkeit geben, die Aufnahmen global zu speichern, um eine große Datengrundlage für die Verbesserung des Modells zu schaffen. Weiterhin sollen neu trainierte Modelle einfach auf dem Gerät *over the air* deployed werden können.

Folgend beschreiben wir unsere Wireframe-Ideen und gehen im Anschluss auf die technische Umsetzung ein.

9.1 Wireframes

Mithilfe der Webapplikation Moqups⁵⁶ haben wir die Wireframes erstellt. Dabei konnten wir remote über unsere Ideen für die App diskutieren und diese direkt für den anderen sichtbar umsetzen. Um einen Gesamtüberblick über unser Ergebnis (inkl. Storyboard) zu erhalten, haben wir die Datei *Überblick Wireframes.png*⁵⁷ in unserem Git-Repository abgelegt und zusätzlich via Moqups einen Online-View⁵⁸ erstellt. Folgend werden unsere Ideen zu den einzelnen Screens beschrieben.

Start-Screen

Der Start-Screen ist die Ansicht, die ein User sieht, sobald er die App öffnet. Nach unserer Vorstellung sollten die meisten Funktionen, die die Applikation bietet, bereits vom Start-Screen aus erreicht werden können.

⁵⁶ <https://app.moqups.com>

⁵⁷ Git Repository: 9. App-Entwicklung /9.1 Überblick Wireframes.png

⁵⁸ <https://app.moqups.com/xD9w5XacH6/view/page/ae8fe8eb0>

Für den Start-Screen hatten wir dabei zwei unterschiedliche Ideen. Diese werden in Abbildung 22 dargestellt.

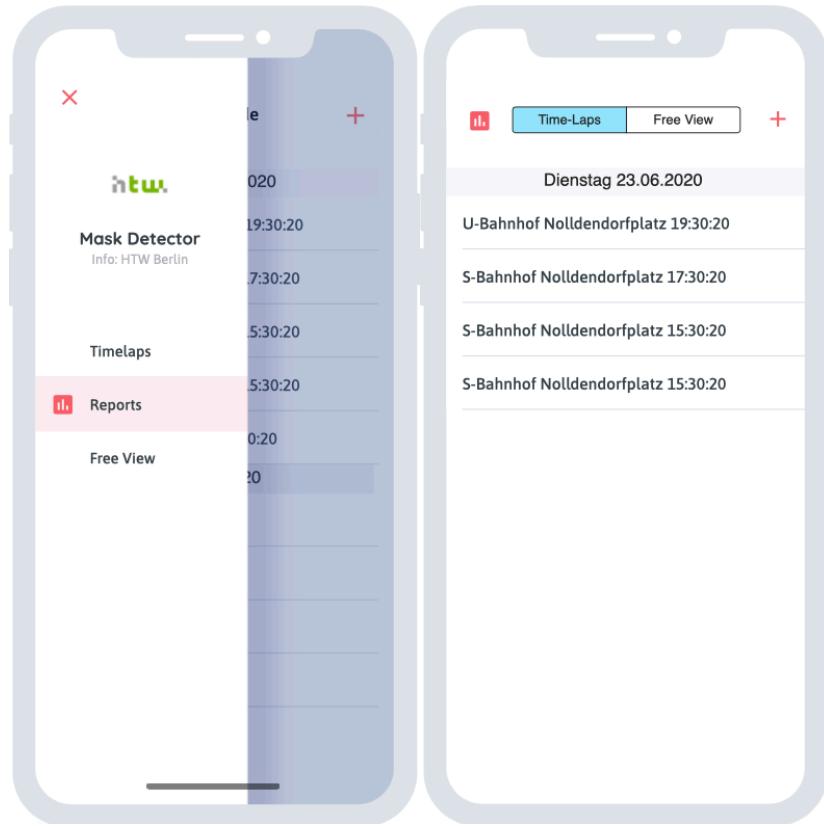


Abbildung 22 Start Screens

Die auf der linken Seite abgebildete Idee sollte ein eingeschobenes Navigations-Menü beinhalten, von wo aus man zu den drei Ansichten *Timelaps*, *Reports* oder *Free View* wechseln kann. Alternativ dazu haben wir in der Ansicht auf der rechten Seite die Auswahlmöglichkeiten in den Header-Bereich geschoben. Durch einen Klick auf das Symbol hat der User die Möglichkeit auf die *Reports* Ansicht zu wechseln. Der Radiobutton ermöglicht einen Wechsel zwischen *Time-Laps* und *Free View* und über das Symbol soll die Aufnahme Funktion im *Free View*, beziehungsweise die Aufnahmeoptionen im *Time Laps* Modus geöffnet werden. Aufgrund der einfacheren prototypischen Umsetzung der Variante 2 haben wir uns im Entwicklungsprozess an die rechte Idee orientiert.

Time-Laps-Ansicht

In der Time-Laps-Ansicht werden alle Aufnahme-Sessions nach Datum sortiert aufgelistet (siehe Abbildung 23). Mit einer Aufnahme-Session ist dabei eine Sammlung von einem oder mehreren Bildern gemeint, welche an einem Standort aufgenommen und mithilfe des Modells ausgewertet wurden. Hinter jeder Session-Bezeichnung wird das Datum der letzten Aufnahme angegeben und die bereits erwähnte Sortierung ist übersichtlich nach Tagen getrennt dargestellt. Falls nicht mehr alle Aufnahme-Sessions auf einem Bildschirm angezeigt werden können, soll die Navigation über eine Scroll-Funktion möglich sein, wobei der Header dauerhaft sichtbar bleibt. Hier bietet das Symbol dem User die Möglichkeit auf die *Reports*-Ansicht zu wechseln. Hingegen soll das Symbol dem User eine Ansicht für die Aufnahmeoptionen im *Time Laps* Modus öffnen. Weiterhin soll dieser die Möglichkeit haben auf eine einzelne Aufnahme-Session zu klicken, um die einzelnen Bilder der Session in einer *Aufnahme-Ansicht* einzusehen. Die Löschung einer Aufnahme-Session soll durch einen links Swipe möglich sein.

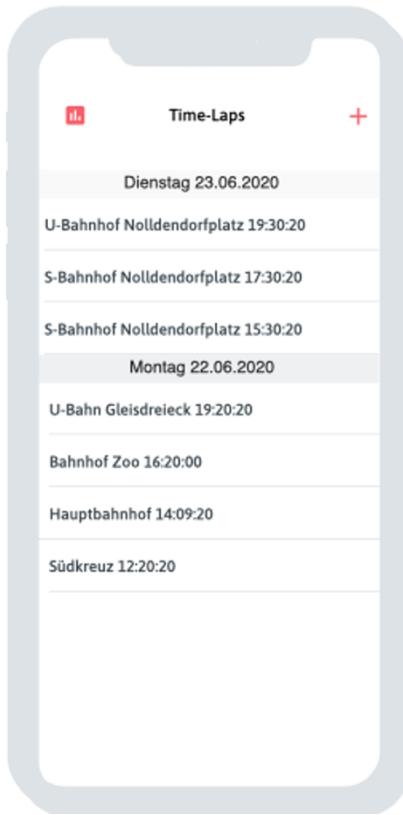


Abbildung 23 Time-Laps-Ansicht

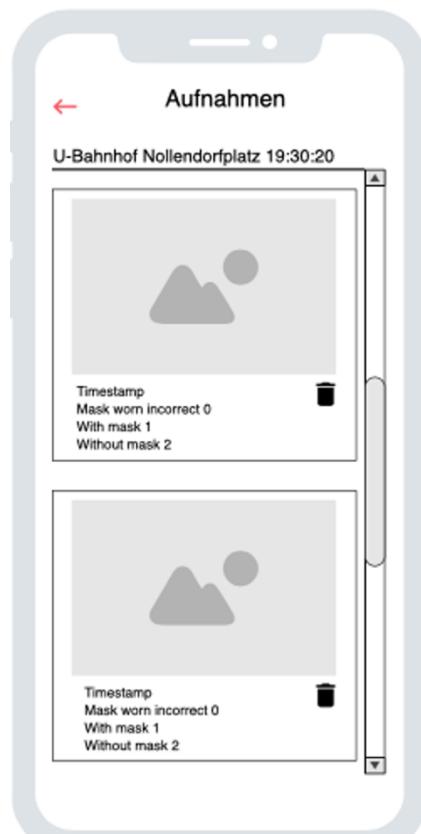


Abbildung 24 Aufnahmen-Ansicht

Aufnahmen-Ansicht

Dieser View wird aufgerufen, sobald der User in der Time-Laps-Ansicht auf eine bereits erstellte Aufnahme-Session tippt. In der Aufnahme-Ansicht werden alle in einer Aufnahme-Session aufgenommenen Bilder nach Datum sortiert aufgelistet (siehe Abbildung 24). Die Bilder sollen dabei in Kacheln in einer Liste dargestellt werden. Innerhalb einer Kachel sollten zusätzlich Informationen zu der Anzahl an erkannten Kategorien für das dargestellte Bild zur Verfügung stehen. Falls nicht mehr alle Kacheln auf einem Bildschirm angezeigt werden können, soll die Navigation über eine Scroll-Funktion möglich sein, wobei der Header dauerhaft sichtbar bleibt. Durch einen Klick auf das Symbol kann der User einzelne Bilder wieder löschen. Weiterhin hat der User die Möglichkeit zurück auf die Time-Laps-Ansicht zu navigieren, indem er den Button auswählt.

Aufnahmeoptionen-Ansicht

Dieser View wird aufgerufen sobald der User in der Time-Laps-Ansicht das Symbol anklickt. Die Aufnahmeoptionen-Ansicht dient dazu, Einstellungen für die Time-Laps Aufnahme zu treffen. Mit einer Time-Laps Aufnahme meinen wir eine Abfolge von Fotos, die in einem vorab definierten Intervall aufgenommen werden. Um das in Kapitel 2 beschriebene Problem der Live-Zählung zu umgehen, schien uns diese Idee der Aufnahmemöglichkeit eine gute Alternative zu sein. Dadurch hat der User die Möglichkeit das iPhone an einem Standort zu positionieren, um beispielsweise im 30 Sekundentakt Bilder aufzunehmen. In stark frequentierten Bereichen, wie U-Bahn-Aufgängen, hat man somit die Möglichkeit Aufnahmen von immer wieder unterschiedlichen Personen aufzunehmen. In das Input-Feld *Name* gibt der User dafür den Namen des Standortes an. In dem Feld *Interval* wird die Taktung als Integer-Wert vom User definiert, wobei er zusätzlich mit einem Dropdown-Button zwischen Sekunden und Minuten wählen kann. Sobald alle Parameter definiert wurden, kann der Start-Button geklickt werden, um die Kamera zu aktivieren und mit der Aufnahme zu beginnen. Durch den Abbrechen Button gelangt der User Zurück zu der Time-Laps-Ansicht



Abbildung 25 Aufnahmeoptionen-Ansicht

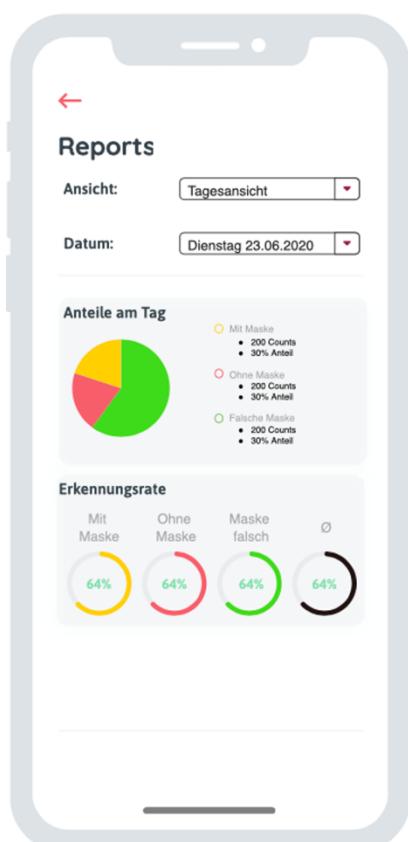


Abbildung 26 Reports-Ansicht

Reports-Ansicht

Dieser View wird aufgerufen, sobald man in der Time-Laps-Ansicht auf das Symbol tippt. Diese Ansicht erfüllt die in der Einleitung beschriebenen Anforderungen der statistischen Auswertung der erkannten Kategorien. Der User soll sich den Report in verschiedenen Ansichten anzeigen lassen können. Denkbar ist dabei eine Auswahl zwischen *Tagesansicht*, *Wochenansicht* und *Monatsansicht*. Zusätzlich soll eine genauere Definition des Zeitraumes erfolgen können. Das in Abbildung 26 dargestellte Beispiel zeigt die Auswahl *Tagesansicht* und das ausgewählte Datum 26.06.2020 an. Bei der Auswahl *Wochenansicht* könnte dann entsprechend eine Kalenderwoche und bei der *Monatsansicht* ein Monat ausgewählt werden. In einer Dashboard-Ansicht werden dann die Ergebnisse der Kategorie-Verteilung sowie die jeweilige Erkennungsrate angezeigt. Bei der Erkennungsrate handelt es sich um die „Sicherheit“, mit der das Modell die einzelnen Kategorien erkannt hat. Vorausgesetzt man nutzt ein sehr gutes Modell, könnte diese Erkennungsrate in eine spätere Auswertung berücksichtigt werden. Wenn beispielsweise 80% der Personen keine Maske getragen haben, sich das Modell bei der Bestimmung aber nur zu durchschnittlich 60% sicher war, kann das Ergebnis angezweifelt werden. Durch einen Klick auf den Button gelangt der User zurück zur Time-Laps-Ansicht.

9.2 IOS Entwicklung

Die IOS App haben wir auf Grundlage der vorgestellten Wireframe-Ideen entwickelt. Dabei mussten wir uns bereits am Anfang gegen eine Core ML Entwicklung mithilfe von *Tensorflow Lite Core ML delegate* entscheiden. Grund dafür ist gemäß der Dokumentation⁵⁹ die fehlende Unterstützung von Full-Integer Modellen. Wie in Abschnitt 8.2 beschrieben, ist ein solches Modell jedoch die beste Lösung für unseren Anwendungsfall. Die Nutzung von Core ML hätte eine geringere Latency für alle User, welche unsere App mit einem iPhone XS oder neuer verwenden.

Um bei der Implementierung des TensorFlow Lite Swift Interpreter Zeit zu sparen, haben wir die TensorFlow Objekt-Detektion Example Applikation⁶⁰ als Grundlage für unsere Entwicklung genutzt. Dadurch konnten wir die bereits erstellten Klassen InferenceViewController, PreviewView, CameraFeedManager, ModelDataHandler, InfoCell, curvedView und OverlayView ohne Änderungen nutzen. Der Grundaufbau der Example Applikation ermöglicht es das bestehende Beispiel TF-lite-Modell durch das eigene zu ersetzen und dieses mit der Rückkamera zu nutzen. Zu dieser bestehenden Applikation haben wir weitere Table-Views hinzugefügt und dadurch unsere Wireframe-Vorlagen bestmöglich umgesetzt. Dabei haben wir darauf geachtet, dass die Funktionen und Ansichten von allen Geräten (iPad und iPhone) ab IOS 9.0 unterstützt werden. Damit decken wir nahezu 100% der aktiven IOS Geräte ab⁶¹. Das Ergebnis kann kommentiert in unserem Git-Repository⁶² eingesehen und auch gern auf das eigene Gerät mit Xcode deployed werden.

Folgend beschreiben wir die erstellten Views und ihre Funktionen, sowie das Kommunikationskonzept mit Google Firebase.



Abbildung 27 Time-Laps-Ansicht
Umsetzung

Time-Laps-Ansicht

Die Time-Laps-Ansicht wurde, wie in Abbildung 27 dargestellt, umgesetzt. Diese dient zugleich als Start-Screen. Wenn der User die App öffnet, kann er sofort alle Aufnahme-Sessions einsehen. Unter jeder Session Bezeichnung (Ort:) wird der Zeitpunkt der letzten Aufnahme angeführt. Die Aufnahme-Sessions werden, auf Grundlage ihrer letzten enthaltenen Aufnahme, übersichtlich nach Tagen gruppiert. Falls nicht mehr alle Aufnahme-Sessions auf einem Bildschirm angezeigt werden können, erfolgt die Navigation über eine Scroll-Funktion, wobei der Header dauerhaft sichtbar bleibt. Durch einen, auf dem jeweiligen Tabellenelement ausgeführten Swipe nach links, kann das jeweilige Element gelöscht werden und über den Button *Report* hat der User die Möglichkeit zu der Report-Ansicht zu wechseln. Das **+** Symbol dient dazu die Ansicht der Aufnahmeeoptionen zu öffnen. Weiterhin hat der User die Möglichkeit auf eine einzelne Aufnahme-Session zu klicken, um die einzelnen Bilder der Session in einer *Aufnahme-Ansicht* einzusehen. Die Live-View Option aus unseren Start-Screen Wireframes haben wir verworfen, da während der

⁵⁹ https://www.tensorflow.org/lite/performance/coreml_delegate

⁶⁰ https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/ios

⁶¹ <https://david-smith.org/iosversionstats/>

⁶² Git Repository: 9. App-Entwicklung/Mask-Detector IOS App

Bildaufnahme bereits eine Live-Erkennung der Objekte erfolgt. Die Implementierung des Views wurde in der TimeLapsTableViewController.swift⁶³ Datei erstellt.

Aufnahmen-Ansicht

Dieser View wird aufgerufen, sobald man in der TimeLaps-Ansicht auf eine bereits erstellte Aufnahme-Session tippt. In der Aufnahme-Ansicht werden alle in einer Aufnahme-Session aufgenommenen Bilder nach Datum sortiert aufgelistet (siehe Abbildung 28). Jede Aufnahme wird in zugeschnittener Form mit entsprechender Datumsangabe und Informationen zu der Anzahl an erkannten Kategorien in einer Listendarstellung angezeigt. Falls nicht mehr alle Aufnahmen auf einem Bildschirm angezeigt werden können, erfolgt die Navigation über eine Scroll-Funktion, wobei der Header dauerhaft sichtbar bleibt. Durch einen, auf dem jeweiligen Tabellenelement ausgeführten Swipe nach links, kann dieses gelöscht werden. Anders als in den Wireframes angedacht hat der User zudem die Möglichkeit jedes Bild durch einen Klick in Großformat anzuzeigen. Über den Button *Time-Laps Button* erfolgt der Wechsel zurück in die gleichnamige Ansicht. Die Implementierung des Views wurde in der ImagesTableViewController.swift⁶⁴ Datei erstellt.

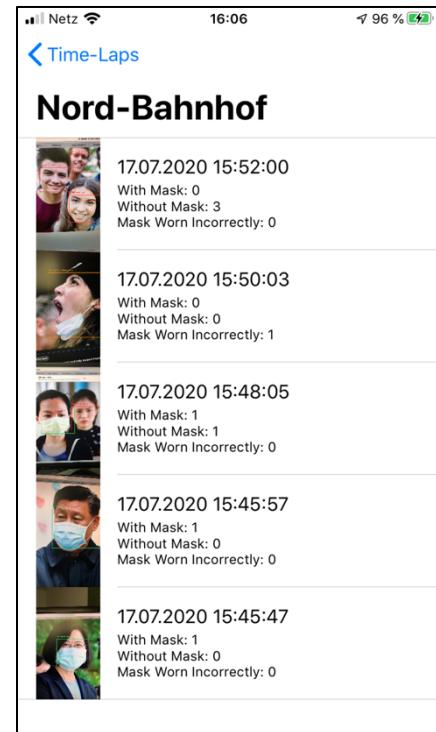


Abbildung 28 Aufnahmen-Ansicht Umsetzung

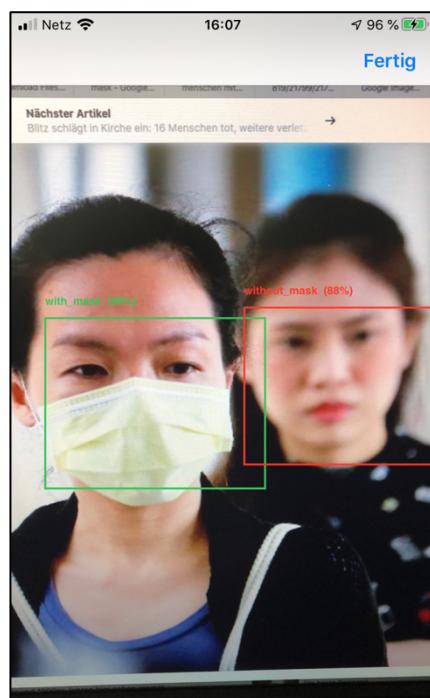


Abbildung 29 Full-Image-Ansicht Umsetzung

Full-Image-Ansicht

Dieser View wird aufgerufen, sobald man in der Aufnahme-Ansicht auf eine Aufnahme tippt. Dem User wird das entsprechende Bild auf dem kompletten Display angezeigt (siehe Abbildung 29). Die erkannten Kategorien sind darauf durch ein farbiges Rechteck umrahmt dargestellt. Zudem wird über dem Rechteck den Namen der Kategorie sowie die Erkennungsgenauigkeit angezeigt. Über den Button *Fertig* gelangt der User zurück in die Aufnahmen-Ansicht. Die Implementierung des Views wurde in der BigImageViewController.swift⁶⁵ Datei erstellt.

⁶³ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/TimeLapsTableViewController.swift

⁶⁴ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/ImagesTableViewController.swift

⁶⁵ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/BigImageViewController.swift

Aufnahmeoptionen-Ansicht

Dieser View wird aufgerufen sobald der User in der Time-Laps-Ansicht das Symbol anklickt. Die Aufnahmeoptionen-Ansicht dient dazu, Einstellungen für die Time-Laps Aufnahme zu treffen (siehe Abbildung 30). Mit einer Time-Laps Aufnahme meinen wir eine Abfolge von Fotos, die in einem vorab definierten Intervall aufgenommen werden. In das Input-Feld *Ort* gibt der User dafür den Namen des Standortes an. in dem Feld *Intervall* wird die Taktung als Integer-Wert vom User definiert, wobei er zusätzlich mit einem Dropdown-Button zwischen Sekunden und Minuten wählen kann. Sobald alle Parameter definiert wurden, färbt sich das Symbol blau und kann anschließend angetippt werden. Durch einen Klick auf die Kamera öffnet sich die Time-Laps-Aufnahme-Ansicht. Über den Button *Abbrechen* gelangt der User zurück in die Time-Laps-Ansicht. Die Implementierung des Views wurde in der `ConfigureMaskDetectionTableViewController.swift`⁶⁶ Datei erstellt. Die Implementierung des Drop-Downbutton wurde in der `ChooseItemTableViewController.swift`⁶⁷ Datei erstellt.

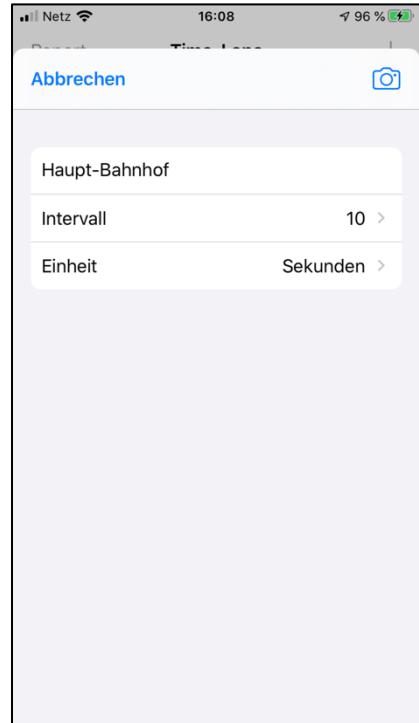


Abbildung 30 Aufnahmeoptionen-Ansicht Umsetzung

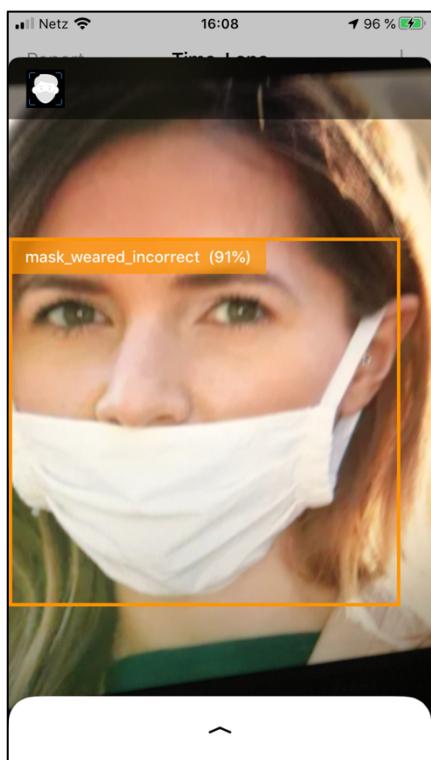


Abbildung 31 Time-Laps-Aufnahme-Ansicht Umsetzung

Time-Laps-Aufnahme-Ansicht

Dieser View wird aufgerufen, sobald man in der Aufnahmeoptionen-Ansicht auf das Symbol tippt. Dem User wird die Live-Aufnahme der Rückkamera angezeigt. In dem Live-Bild werden die erkannten Objekte direkt mit einem Rahmen markiert. Zusätzlich erfolgt eine Benennung des erkannten Objektes. Dieser tolle Effekt wird mithilfe des TensorFlow-Interpreters ausgeführt. Dieser rechnet jedes erfasste Bild in die Größe des vom Modell erwarteten Inputwertes um und führt die Objekt-Erkennung durch. An dieser Stelle macht sich die Optimierung unseres Modells bemerkbar. Diese Funktion dient in unserem Prototypen dazu, bereits ein Gefühl für die in der Einleitung beschriebene und später angestrebte Live-Zählung mithilfe von IOT Geräten zu bekommen. In unserer Lösung wird, in Abhängigkeit der zuvor ausgewählten Intervall-Optionen, regelmäßig ein Bild der aktuellen Ansicht aufgenommen und abgespeichert. Falls eine WLAN Verbindung besteht, werden die Bilder automatisch in annotierter und nicht annotierter Form an einen Google-Firebase Storage⁶⁸ gesandt. Das Beenden der Aufnahme erfolgt durch einen

⁶⁶ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/
ConfigureMaskDetectionTableViewController.swift

⁶⁷ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/
ChooseItemTableViewController.swift

⁶⁸ <https://firebase.google.com>

Swipe nach unten. Die Implementierung des Views wurde in der ViewController.swift⁶⁹ Datei erstellt.

Reports-Ansicht

Dieser View wird aufgerufen, sobald man in der Time-Laps-Ansicht auf *Report* tippt (siehe Abbildung 31). Diese Ansicht zeigt die Verteilung der Kategorien an einzelnen Tagen an.

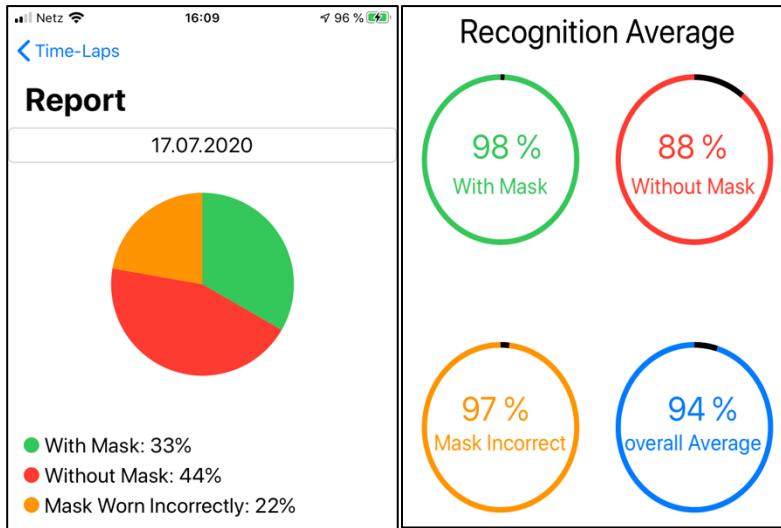


Abbildung 32 Report-Ansicht Umsetzung

Durch einen Klick auf das aktuelle Datum (in diesem Beispiel 17.07.2020) öffnet sich eine Drop-Down-Auswahl, aus welcher ein verfügbarer Tag ausgewählt werden kann. Das Kreisdiagramm repräsentiert die darunter angegebenen Verteilungen der Kategorien. Zusätzlich zeigt die App die durchschnittlichen Erkennungsgenauigkeiten der einzelnen Kategorien, sowie aller Kategorien. Die animierten Kreise wurden mithilfe der Libary *SRCountdownTimer*⁷⁰ eingebunden. Die in den Wireframes dargestellte Auswahl der Monats- und Wochenansicht wurde in dieser Version des Prototyps noch nicht berücksichtigt. Die Implementierung des Views wurde in der ReportsViewController.swift⁷¹ Datei erstellt. Die Implementierung des Drop-Downbutton wurde in der ChooseItemTableViewController.swift⁷² Datei erstellt.

Um die einzelnen Views und Funktionen im Live-Einsatz zu betrachten, haben wir eine Video-Aufnahme⁷³ erstellt, welche in dem Git-Repository eingesehen werden kann.

Kommunikationskonzept mit Google Firebase

Während der Entwicklung haben wir uns Gedanken über Konzepte zur Nutzung der aufgenommenen Bilder und über das weitere Training sowie die Bereitstellung des bestehenden Modells gemacht. Eine Idee war dabei einen On-Device Trainingsalgorithmus zu nutzen. Der Nachteil an dieser Lösung ist jedoch, dass die Geräte nur von ihren eigenen Bildern profitieren und schlussendlich unterschiedliche Erkennungsgenauigkeiten aufweisen. Ein besserer Ansatz ist die regelmäßige Verteilung eines, mithilfe von allen neuen Aufnahmen trainiertes, Modells.

⁶⁹ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/ViewController.swift

⁷⁰ <https://github.com/maikdrop/SRCountdownTimer.git>

⁷¹ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/ReportsViewController.swift

⁷² Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/ChooseItemTableViewController.swift

⁷³ Git Repository: 9. App-Entwicklung/9.2 Präsentation.MP4

Zur Umsetzung dieser Idee haben wir, wie bereits in der Beschreibung der Time-Laps-Aufnahme-Ansicht erwähnt, eine Schnittstelle zu Google Firebase in unsere App implementiert. Die Kommunikation erfolgt dabei wie in Abbildung 33 dargestellt.

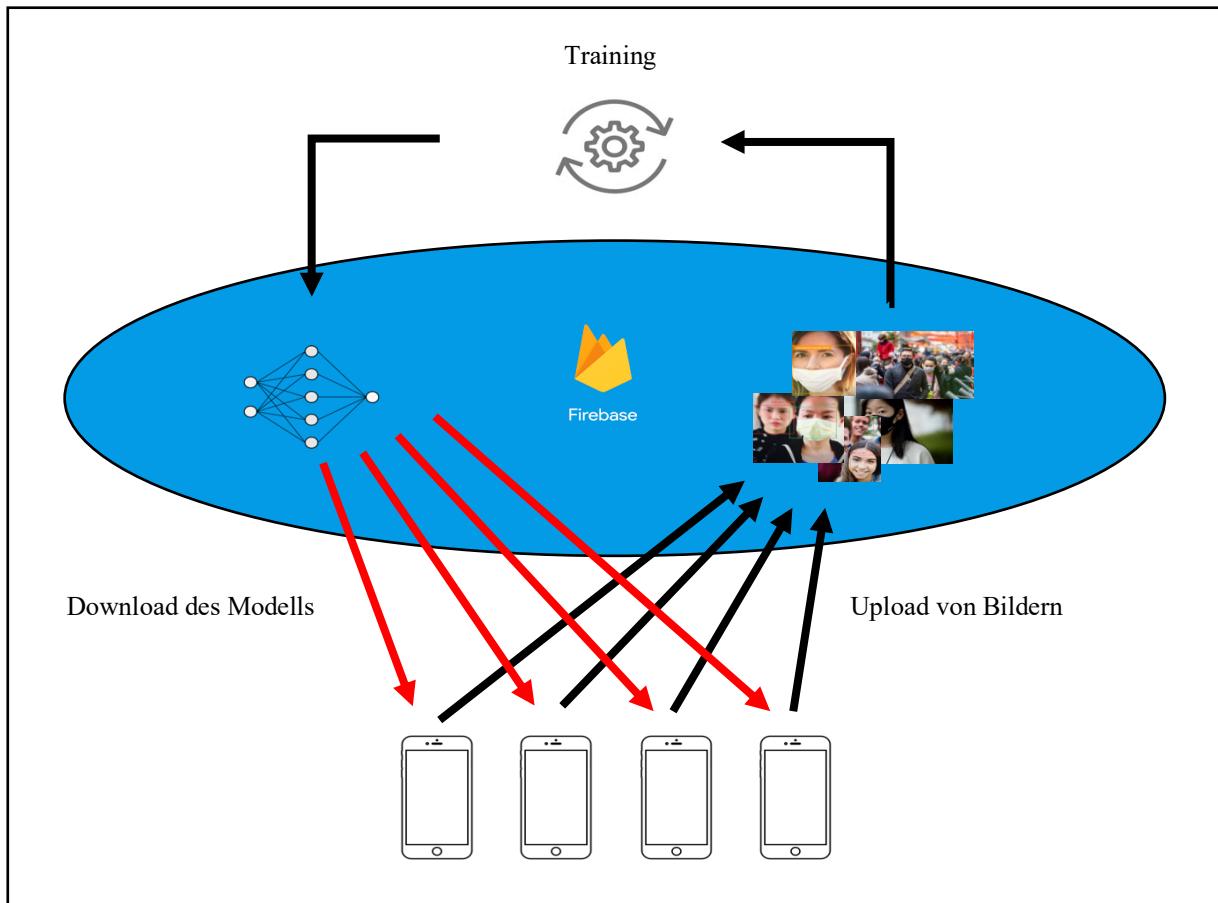


Abbildung 33 Kommunikationskonzept

Bei jedem App-Start wird das aktuelle Modell von Firebase heruntergeladen und von dem TensorFlow Interpreter genutzt. Weiterhin werden alle Bilder, in annotierter und nicht annotierter Form, nach der Aufnahme, in den Firebase Storage geladen. Diese können im Anschluss anhand ihrer Annotationen ausgewertet bzw. zum Optimieren des aktuellen Modells genutzt werden. Das verbesserte Modell wird daraufhin via Firebase den Geräten zur Verfügung gestellt, welche dieses beim nächsten App-Start wieder herunterladen. Um die Kosten für mobile Daten nicht in die Höhe zu treiben, erfolgt der Up- und Download nur bei einer bestehenden WLAN Verbindung. Die Implementierung des Bilder-Upserts wurde in der Datei ViewController.swift⁷⁴ erstellt. Der Modell-Download beim App-Start wird in der Datei AppDelegate.swift⁷⁵ ausgeführt. Die Überprüfung der WLAN Verbindung erfolgt mithilfe der Reachability⁷⁶ Klasse.

Um den Austausch eines Modells zu demonstrieren, haben wir das Video 9.2 Modellwechsel.mov⁷⁷ aufgenommen. Bei der Aufnahme ist zu erkennen, dass wir beim Start der App ein Coco Modell nutzen, welches Objekte wie Keyboard und Laptop erkennt. Im Anschluss

⁷⁴ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/ViewController.swift

⁷⁵ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/ViewControllers/AppDelegate.swift

⁷⁶ Git Repository: 9. App-Entwicklung/Mask-DetectorIOS App/ObjectDetection/Model/ Reachability.swift

⁷⁷

laden wir ein trainiertes Mask-Detection Modell in Firebase hoch, starten unsere App erneut und demonstrieren die Erkennung der Kategorie without_mask.

9.3 Installation und Modell-Updates

Wie in der Einleitung bereits beschrieben, ist diese Applikation nicht für den kommerziellen Gebrauch vorgesehen. Aus diesem Grund ist eine Bereitstellung über den Apple App-Store nicht notwendig. Vielmehr sollten in Instituten oder andren Einrichtungen MDM Lösungen⁷⁸ genutzt werden, um die Applikation auf den eigenen Geräten zu installieren.

9.4 Test

Wir haben die App von 2 Personen an verschiedenen Plätzen testen lassen. Für die Tests haben wir vorab 12 Fälle definiert. Die Ergebnisse werden in Tabelle 11 dargestellt.

ID	Testfälle	Vorbereitung	Ergebnis
1	Erstellen einer neuen Aufnahme.		bestanden
2	Überprüfung des korrekten Zeitintervalls bei der Aufnahme	Eine Aufnahme muss gestartet sein.	bestanden
3	Überprüfung, ob die Anzahl an erkannten Objekten einer Aufnahme der Anzahl an dargestellten Boxen pro Kategorie entspricht	Eine Aufnahme muss beendet sein. Auf der Aufnahme müssen Objekte erkannt worden sein.	bestanden
4	Änderung des Report-Datums	Es müssen Aufnahme-Sessions von mindestens 2 Tagen bestehen	bestanden
5	Überprüfung der korrekten Sortierung der Aufnahme-Sessions	Es müssen mehrere Aufnahme-Sessions von mindestens 2 Tagen bestehen	bestanden
6	Überprüfung der korrekten Reportdarstellung	Es müssen mehrere Aufnahme-Sessions bestehen. Es muss die Anzahl an erkannten Objekten ausgezählt werden.	bestanden
7	Download eines neuen Modells bei einer bestehenden WLAN Verbindung	Ein neues Modell muss in Firebase hochgeladen werden. Das Gerät muss mit einem WLAN verbunden sein. Die App muss neu gestartet werden.	bestanden
8	Nutzung des bestehenden Modells, falls keine WLAN Verbindung besteht	Ein neues Modell muss in Firebase hochgeladen werden. Das Gerät darf nicht mit einem WLAN verbunden sein. Die App muss neu gestartet werden.	bestanden
9	Upload von originalen Aufnahmen bei einer bestehenden WLAN Verbindung	Aufnahme muss gestartet werden. Das Gerät muss mit einem WLAN verbunden sein.	bestanden
10	Upload von annotierten Aufnahmen bei einer bestehenden WLAN Verbindung		bestanden

⁷⁸ <https://developer.apple.com/documentation/devicemanagement>

11	Upload von originalen Aufnahmen ist ohne WLAN Verbindung nicht möglich	Aufnahme muss gestartet werden. Das Gerät darf nicht mit dem WLAN verbunden sein	bestanden
12	Upload von annotierten Aufnahmen ist ohne WLAN Verbindung nicht möglich		bestanden

Tabelle 11 Testergebnisse

Wie in den Ergebnissen zu erkennen ist, konnten die grundlegenden Funktionen problemlos ausgeführt werden. Jedoch wurde von den Testern die Reaktionszeit bei der Erkennung von Gesichtern, sowie die Erkennungsgenauigkeit im Allgemeinen kritisiert. Teilweise hat sich der Over-Layer (die farbigen Boxen) für mehrere Sekunden nicht verändert. Zusätzlich wurden viele Gesichter nicht oder falsch identifiziert. Dementsprechend war eine sinnvolle Auswertung der Statistik nicht möglich. In unserm Git-Repository haben wir Beispielaufnahmen⁷⁹ hochgeladen, welche die Defizite bei der Erkennung aufzeigen.

Weiterhin haben wir Feedback für Verbesserungen der App erhalten.

- Aufnahmen in Querformat sollten möglich sein
- Der Aufnahme-View sollte das komplette Display einnehmen
- Das Beenden der Aufnahmen durch einen Swipe-Down ist irritierend. Es sollte einen Beenden Button geben
- Die Statistik sollte mehr Filteroptionen anbieten (Monat/ Woche/ Ort)
- Die Bilder sollten exportierbar sein bzw. in der Foto-App abgespeichert werden können
- Es sollte die aktuelle Version des Modells angezeigt werden
- Orte sollten nach einer Aufnahme noch umbenannt werden können
- Der Ort sollte über den Standort automatisch identifiziert werden

10 Ergebnis & Verbesserung

Leider konnten wir die in der Einleitung beschriebenen Anforderungen an unsere Applikation nur teilweise erfüllen. Die größten Schwierigkeiten ergeben sich aus einer zu ungenauen Erkennungsrate sowie einer langsamen Performance des Modells. Um diese Metriken zukünftig zu verbessern, haben wir im letzten Schritt dieses Projektes das Ergebnis des Modells und der App nochmals untersucht. In diesem Kapitel gehen wir auf unsere Erkenntnisse genauer ein.

10.1 Erkennung

Um die einzelnen Schwachstellen in der Erkennung deutlich zu machen, haben wir, zusätzlich zu den in Abschnitt 9.4 erwähnten Beispielaufnahmen, Videos heruntergeladen und mithilfe des Scripts *Video_to_jpg.py*⁸⁰ in einzelne JPG-Sequenzen umgewandelt. Für jedes Bild wurde daraufhin eine Annotierung anhand der, durch unser Modell, erkannten Gesichter durchgeführt (Script *detect_img_sequenz.py*⁸¹). Die daraus entstandene Sequenz aus Bildern haben wir im Anschluss erneut zu einem Video zusammengeführt. Bei der Betrachtung der Aufnahmen haben wir folgendes festgestellt:

⁷⁹ Git Repository: 9. App-Entwicklung/9.4 Beispielaufnahmen

⁸⁰ Git Repository: 10. Ergebnis & Verbesserung /10.1 Video_to_jpg.py

⁸¹ Git Repository: 10. Ergebnis & Verbesserung /10.1 detect_img_sequenz.py

- **Erkennung in der Nähe**

Die Erkennung von nahen Aufnahmen funktioniert mit unserem Modell am besten. In dem Video *Einzelaufnahme.mov*⁸² ist dies gut zu erkennen. Der Grund dafür ist eine verhältnismäßig hohe Anzahl an Nahaufnahmen im Trainingsset. Speziell die Kategorie *mask_waered_incorrect* wird ausschließlich auf Aufnahmen erkannt, welche mit maximal 1 – 2 m Abstand erfolgen.

- **Erkennung in der Ferne**

Je kleiner die Gesichter werden, desto schlechter ist die Erkennung. Dies ist sehr gut in dem Video *Mask-Finder 3*⁸³ erkennbar. Die Gesichter werden im Regelfall erst ab einem gewissen Punkt erkannt.

Lösung: Das Trainingsset sollte aus mehreren Bildern mit Gesichtern in weiterer Entfernung bestehen.

- **Erkennungsgenauigkeiten der Kategorien**

Konträr zu dem Ergebnis aus Abschnitt 7.4 wurde die Kategorie *mask_weared_incorrect* am schlechtesten erkannt. Speziell in den Videos Mask-Finder 2⁸⁴ und Mask-Finder 3⁸³ ist dies zu erkennen. Gesichter mit falsch getragener Maske werden häufig als *with_mask* oder gar nicht erkannt. Weiterhin werden die Kategorien *with_mask* und *without_mask* sporadisch vertauscht (siehe Video Mask-Finder 1⁸⁵).

Lösung: Wie in Abbildung 4 gezeigt wurde, ist die Kategorie *mask_weared_incorrect* im Gegensatz zu den anderen weitaus weniger repräsentiert. Weiterhin sind nahezu alle Aufnahmen der Kategorie aus der Nähe aufgenommen. In Zukunft sollten die Kategorien mit einer hohen Anzahl nahezu gleich repräsentiert sein. Dies gewährleistet eine bessere Unterscheidung aller Kategorien, sowie ein genaueres Evaluationsergebnis.

- **Erkennung von unterschiedlichen Maskenfarben**

Die Erkennung von hellen Masken (am besten weiß und blau) funktioniert am besten, was mit einer hohen Repräsentation an weißen Masken im Datensatz zusammenhängt. Weiterhin lässt sich das Modell durch die Nutzung von speziellen Masken-Designs, wie in Abbildung 31 dargestellt, täuschen.



Abbildung 34 spezielles Masken-Designs

Weiterhin werden Vollbärte hin und wieder als Maske missinterpretiert.

⁸² Git Repository: 10. Ergebnis & Verbesserung / Einzelaufnahme.mov

⁸³ Git Repository: 10. Ergebnis & Verbesserung / Mask-Finder 3.mov

⁸⁴ Git Repository: 10. Ergebnis & Verbesserung / Mask-Finder 2.mp4

⁸⁵ Git Repository: 10. Ergebnis & Verbesserung / Mask-Finder 1.mp4

Lösung: Die Trainingsdaten müssen mit vielen unterschiedlichen Masken-Typen, -Farben und -Designs angereichert werden.

- **Problem bei Drehung des Gesichtes**

Die besten Erkennungs-Ergebnisse werden bei Frontalaufnahmen erzielt. Sobald sich eine Person leicht dreht, kann dem Gesicht durch unser Modell keine Kategorie zugewiesen werden. Dies ist sehr gut in dem Video Mask-Finder⁸⁶ zu sehen.

Lösung: Die Trainingsdaten müssen mit Bildern von Gesichtern aus verschiedenen Perspektiven erweitert werden. Speziell die Seitenprofile sind im Moment unterrepräsentiert.

- **Problem bei Bewegungen**

Bei zu schnellen Kamera- oder Personenbewegungen verschwimmt das Bild leicht. Dadurch werden die Kategorien nicht ordnungsgemäß erkannt. Dies ist im Video Mask-Finder⁸⁷ gut zu erkennen.

Lösung: Die Trainingsdaten müssen mit Bildern von aus der Bewegung heraus aufgenommenen Gesichtern erweitert werden.

- **Hauttypen**

Der größte Anteil unserer Trainingsdaten bildet Personen aus dem asiatischen Raum ab. Dies führt teilweise zu Erkennungsschwierigkeiten von Personen mit anderen Hauttypen.

Lösung: Auch aus dieser Perspektive betrachtet sollten die Trainingsdaten optimiert werden.

Weiterhin kann die Erkennung durch einen optimierten Trainingsprozess verbessert werden. Die in Abschnitt 6.4 beschriebene Konfigurationsdatei bietet eine große Auswahl an Parametern, welche einen direkten Einfluss auf das Trainingsergebnis haben. Beispielsweise kann mit einer Veränderung der *Learning-Rate* und *Batch-Size* experimentiert werden. Zusätzlich spielt die Dauer des Trainings eine entscheidende Rolle. Die von uns gewählten 10000 Steps wurden von uns hauptsächlich aufgrund der langen Trainingszeit gewählt. Die in der Konfigurationsdatei vorgeschriebenen Trainings-Steps beliefen sich auf bis zu 200000.

10.2 Geschwindigkeit

Wie in den Videos unserer Test-Live-Aufnahmen⁸⁸ zu sehen, ist unser Modell für Live-Erkennungen noch viel zu langsam. Wie in dem TensorFlow Model Zoo⁸⁹ dargelegt, erreichen die Pixel4 Edge TPU models eine Latency von 6.9 Millisekunden. Dies ist ca 6x schneller als unser Modell. Dementsprechend hat unsere Lösung noch viel Optimierungspotenzial.

Als Lösungsansatz könnten in Zukunft eine genauere Konfiguration des Quantization-Prozesses durchgeführt werden. Ähnlich wie beim Training beschreibt auch hier TensorFlow in der Dokumentation⁹⁰ eine Auswahl an Parametern. Weiterhin könnten andere, von

⁸⁶ Git Repository: 10. Ergebnis & Verbesserung / Mask-Finder 1.mp4

⁸⁷ Git Repository: 10. Ergebnis & Verbesserung / Mask-Finder 2.mp4

⁸⁸ Git Repository: 9. App-Entwicklung/9.4 Beispieldaten

⁸⁹ https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md

⁹⁰ https://www.tensorflow.org/lite/performance/quantization_spec

TensorFlow vorgestellte Optimierungs-Optionen⁹¹ betrachtet werden. Beispielsweise das *Quantization-aware-training*⁹² oder *Post-trained dynamic range quantization*⁹³.

Zusätzlich könnte die App mit CoreML implementiert werden, um zu untersuchen, ob der Geschwindigkeitsverlust durch die 32/16 Bit-Quantifizierung, durch Apples Neural Engine ausgeglichen werden kann.

10.3 Was würden wir das nächste Mal anders machen?

Neben den vorab vorgeschlagenen Lösungsansätzen hätten wir uns bei den Kategorien nur auf die Auswahl *with_mask* und *without_mask* fokussieren sollen. Dafür hätten uns grundlegend mehr Trainingsdatensets online zur Verfügung gestanden. Zudem hätten wir dadurch das Missverhältnis des Kategoriendaufkommens besser angeleichen können. Weiterhin würden wir mehr Zeit in die Variationsvielfalt der im Abschnitt 3.2 beschriebenen Image-Augmentation stecken. Neben den von uns durchgeführten Verfahren wäre dabei beispielsweise eine Verdunklung und Erhellung der Bilder denkbar. Ein weiterer Punkt, den wir beim nächsten Mal besser machen würden, ist die Fokussierung auf 1-2 Modells beim Transfer-Learning. Dadurch, dass wir bei diesem ersten Versuch noch keine Erfahrungswerte mit den verschiedenen Modellen hatten, blieb uns zum Schluss keine Zeit mehr einzelne Optimierungsanpassungen auszuprobieren.

Auch die Implementierung der IOS App war für uns, als absolute Neulinge in der Swift-Programmierung, sehr herausfordernd. Im Nachhinein betrachtet, wäre es sinnvoller gewesen einzelne Funktionen in zusätzliche Klassen auszulagern, um diese immer wieder verwenden zu können. Zusätzlich würden wir zukünftig die Nutzung von parallelen Threads geschickter einsetzen, um die App bei der Erkennung von Objekten flüssiger zu machen. Außerdem würden wir die Funktion der Live-Erkennung, wie in den Wireframes (Abschnitt 9.1) eigentlich geplant, strikt von der Bilder-Aufnahme trennen.

⁹¹ https://www.tensorflow.org/lite/performance/model_optimization#quantization

⁹² https://www.tensorflow.org/model_optimization/guide/quantization/training

⁹³ https://www.tensorflow.org/lite/performance/post_training_quant