VERGLEICH VERSCHIEDENER ARCHITEKTURFAMILIEN BEI FACE MASK DETECTION

DOMINIK SMREKAR, SANIYE OGUL, JOHANNES HORST

Inhaltsverzeichnis

Einleitung

Die rasante Weiterentwicklung von Objekterkennungssystemen hat in den letzten Jahren zu vielfältigen Anwendungen in Bereichen wie Gesichtserkennung, autonomen Fahrzeugen, Überwachungssystemen und medizinischer Bildgebung geführt. Diese Systeme ermöglichen es uns, komplexe Aufgaben automatisch zu bewältigen, indem sie Objekte in Bildern oder Videos erkennen und lokalisieren. In dieser Studienarbeit steht der Vergleich verschiedener Architekturfamilien für Objekterkennungssysteme im Fokus: YOLO (You Only Look Once), Faster R-CNN (Region-based Convolutional Neural Network) und SSD (Single Shot MultiBox Detector).

Die kontinuierliche Verbesserung der KI-gestützten Objekterkennung hat enorme Auswirkungen auf unsere Gesellschaft. Insbesondere die zuverlässige Erkennung von Masken spielt eine entscheidende Rolle beim Schutz der öffentlichen Gesundheit und der Eindämmung von Infektionskrankheiten. Durch den Vergleich der genannten Architekturfamilien können wir herausfinden, welches Modell sich am besten für die Maskenerkennung eignet. Diese Erkenntnisse tragen dazu bei, zukünftige Entwicklungen in diesem Bereich voranzutreiben und innovative Lösungen zur Bekämpfung von Infektionskrankheiten bereitzustellen.

Zusätzlich eröffnet die fortschreitende Integration von künstlicher Intelligenz neue Möglichkeiten für die Erkennung des Tragens von Mundschutzen. Daher wird in dieser Arbeit auch die Entwicklung eines Modells zur Erkennung des Mundschutztragens untersucht. Hierbei werden verschiedene Modelle aus den Architekturfamilien YOLO, Faster R-CNN und SSD näher betrachtet, um die bestmögliche Lösung zu ermitteln.

Datensätze

Um die Vergleiche zwischen den Modellen bestmöglich einheitlich zu gestalten, wurden zwei Datensätze verwendet, die beide von Kaggel stammen. Der erste Datensatz ist sehr umfangreich und enthält 20 Klassen mit insgesamt 4320 Fotos im PNG, JPG und JPEG-Format. Die Annotationen liegen im COCO-Format vor und enthalten Bounding Boxes mit den Werten für xmin, ymin, xmax und ymax. Dieser Datensatz ist äußerst vielfältig und enthält viele verschiedene Bilder in verschiedenen Szenarien. Es gibt zahlreiche Kombinationen von Gruppenaufnahmen mit und ohne Mundschutz sowie Einzelpersonen in Nahaufnahmen und "fernen" Aufnahmen (https://www.kaggle.com/datasets/wobotintelligence/face-mask-detection-dataset).

Der zweite Datensatz wurde bereits vorgegeben und ist deutlich kleiner. Er umfasst lediglich 3 Klassen, jeweils mit rund 400 Bildern. Diese variieren untereinander weniger und zeigen hauptsächlich Einzelpersonen. Bei diesem Datensatz wurde ein anderes Format für die Annotationen verwendet. Die vorhandenen Annotationen liegen im XML-Format vor, wobei die Bounding Boxes ebenfalls im Format xmin, ymin, xmax und ymax angegeben sind (https://www.kaggle.com/datasets/andrewmvd/face-mask-detection).

Zusätzlich wurde der XML-Datensatz durch Datenaugmentation dupliziert. Dabei wurden verschiedene Filter auf jedes Bild angewendet, um das Bild deutlich abzudunkeln und leicht zu verrauschen, um Aufnahmen unter schlechten Lichtverhältnissen zu simulieren. Das Ziel dieser Maßnahme ist es, herauszufinden, wie gut das Modell bei schlechten Lichtverhältnissen noch Erkennungsergebnisse erzielt.

Erstellung des dunklen Datensatzes

Es stellte sich heraus, dass es schwierig war, verschiedene Online-Datensätze zu finden, die auf unterschiedlichen Szenarien basierten und sich auf die Erkennung von Gesichtsmasken konzentrierten. Viele verfügbare Datensätze wiesen eine ähnliche Zusammensetzung auf, was es schwierig machte, die Leistung der Modelle unter verschiedenen Bedingungen zu vergleichen.

Um die Low-Light-Performance der Modelle zu untersuchen, wurde daher beschlossen, den Dark-Datensatz künstlich zu erstellen. Durch die Anwendung von Bildtransformationen, die den Eindruck von Aufnahmen im Dunkeln erzeugen, konnten wir eine Vielzahl von Bildern generieren, die sich von den Bildern in den vorhandenen Datensätzen unterscheiden. Dies ermöglichte es uns, die Leistung der Modelle unter solchen simulierten Bedingungen zu bewerten und zu vergleichen.

Der Dark-Datensatz wurde somit speziell entwickelt, um die Fähigkeit der Modelle zur Erkennung von Gesichtsmasken unter Low-Light-Bedingungen zu testen. Durch diesen Ansatz konnten wir eine breitere Palette von Szenarien abdecken und die Performance der Modelle in realistischeren Situationen bewerten.

Für die Erstellung des Dark-Datensatzes wurden verschiedene Bildtransformationen angewendet, um den Eindruck zu erzeugen, dass die Bilder im Dunkeln aufgenommen wurden. Diese Transformationen wurden mithilfe der Funktion apply_low_light() aus der Datei generate_low_light_dataset.ipynb durchgeführt.

Zunächst wurde die Helligkeit der Bilder angepasst, indem ein Multiplikationsfaktor verwendet wurde. Dadurch wurde die Helligkeit auf einen niedrigeren Wert reduziert. Anschließend wurde eine Gamma-Korrektur angewendet, um den Kontrast anzupassen und die Bildqualität anzupassen. Ein Weichzeichner wurde verwendet, um die Details in den Bildern zu verringern und einen unscharfen Effekt zu erzeugen.

Darüber hinaus wurde eine Farbverschiebung eingeführt, um den Bildern einen blauen Farbton zu verleihen. Dies wurde erreicht, indem der Blaukanal verstärkt wurde. Zusätzlich wurde Gaußsches Rauschen hinzugefügt, um den Eindruck von Bildrauschen im Dunkeln zu erzeugen. Dabei wurden Parameter wie der Mittelwert und die Standardabweichung des Rauschens angepasst, um das gewünschte Ausmaß des Rauschens zu erreichen.

Die Transformationen wurden auf jeden Datensatz angewendet, indem der Code über die Bilder im Verzeichnis iteriert wurde. Die Parameter der Transformationen, wie der Helligkeitsfaktor, der Gamma-Korrekturwert, die Weichzeichner-Kernelgröße, der Farbverschiebungsfaktor und die Rauschparameter, konnten angepasst werden, um das gewünschte Ausmaß der Dunkelheit und des Rauschens zu erreichen.

Durch die Anwendung dieser Transformationen konnte ein Dark-Datensatz erstellt werden, der für das Training der Modelle unter simulierten dunklen Bedingungen verwendet wurde. Dies ermöglichte es den Modellen, auch in solchen Umgebungen effektiv Gesichtsmasken zu erkennen.

Verwendete Parameter:

	Helligkeit reduzieren	Gamma reduzieren	Unschärfe Kernel Größe	Farbverschiebung	Rauschen Median	Rauschen Standardabweichung	Verstärkung blau Kanal
Faktor	0,9	0,7	5	0,8	(0,0,0) (R,G,B)	(2,2,2) (R,G,B)	1,6

YOLO

Dieser Abschnitt befasst sich mit der theoretischen Grundlage des Studienprojektes. Für das Trainieren der beiden Modelle wird der YOLOv4 und der YOLOv5 Objekterkennungsalgorithmus verwendet. Dabei wird sowohl auf die Architektur der beiden Modelle eingegangen als auch deren Unterschiede detailliert erläutert.

YOLO bedeutet “you only look once” und ist ein Algorithmus zur Objekterkennung in Echtzeit, der 2015 von Joseph Redmon, Santosh Divvala, Ross Girshick und Ali Farhadi entwickelt wurde. Dieser kann ein ganzes Bild in ein Convolutional Neural Network (CNN) einspeisen und das Ergebnis in einem Durchgang vorhersagen. Dies wird auch One-Stage Algorithmus gennant. Hierbei sagt das CNN sogenannte Labels, Bounding Boxes und Confidence Probabilities voraus.

- Labels: Klassen oder Kategorien der erkannten Objekte (z.B. Hund, Katze).
- Bounding Boxes: Eine Box, die das erkannte Objekt einschließt und dessen Größe und Position speichert.
- Confidence Probability: Wahrscheinlichkeit, dass eine Bounding Box ein Objekt korrekt vorhersagt.

YOLOv4

YOLOv4 wurde ursprünglich im Darknet-Framework implementiert, das von Alexey Bochkovskiy entwickelt wurde. Diese Version ist speziell für die Implementierung von YOLO entworfen worden. Es ist eine Verbesserung der dritten YOLO Version. Für diese Studienarbeit wurde ein Pytorch_YOLOv4 Repository von Wong Kin Yiu verwendet. [1]

Loss-Funktion:

Die Loss-Funktion, die in der Abbildung 1 zu sehen ist, von YOLOv4 besteht aus der Summe der quadratischen Fehler für die Lokalisierung (Bounding Box-Offsets) und die Klassifizierung (bedingte Klassenwahrscheinlichkeiten). Dies kann man in der unteren Abbildung sehen. Zwei Skalierungsparameter werden verwendet, um die Bedeutung der Bounding Box-Koordinaten-Vorhersagen und die Vertrauenspunktvorhersagen für Boxen ohne Objekte zu kontrollieren. Diese Loss-Funktion wird auch bei YOLOv5 verwendet.

$$\mathcal{L}_{\text{loc}} = \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$$

$$\mathcal{L}_{\text{cls}} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} (\mathbb{1}_{ij}^{\text{obj}} + \lambda_{\text{noobj}}(1 - \mathbb{1}_{ij}^{\text{obj}}))(C_{ij} - \hat{C}_{ij})^2 + \sum_{i=0}^{S^2} \sum_{c \in \mathcal{C}} \mathbb{1}_{i}^{\text{obj}} (p_i(c) - \hat{p}_i(c))^2$$

$$\mathcal{L} = \mathcal{L}_{\text{loc}} + \mathcal{L}_{\text{cls}}$$

Abbildung 1 YOLO Loss-Funktion

https://lilianweng.github.io/posts/2018-12-27-object-recognition-part-4/

YOLOv5

YOLOv5 wurde von Glenn Jocher und Ultralytics in PyTorch implementiert [2]. Der Algorithmus wurde in fünf verschiedenen Versionen veröffentlicht die wie folgt heißen:

- YOLOv5**n** bezeichnet ein ultrakleines (Nano-)Modell.
- YOLOv5**s** ist ein kleines Modell.
- YOLOv5**m** ist ein mittelgroßes Modell.
- YOLOv5**l** ist ein großes Modell.
- YOLOv5**x** weist auf ein übergroßes Modell hin.

Die YOLOv5-Modelle sind alle skalierte Varianten derselben Architektur. Sie unterscheiden sich lediglich in der Anzahl der Schichten. Die folgende Abbildung 2 zeigt die 5 Modelle, die mit dem COCO-Datensatz trainiert wurden. Hierbei wurde die durchschnittliche Präzision (AP) berechnet, um die Leistung der Modelle in Abhängigkeit von der GPU-Geschwindigkeit zu vergleichen. Größere Modelle verfügen in der Regel über mehr Parameter und Ebenen, wodurch bessere Ergebnisse für detaillierte Features erzielt werden können. Allerdings können kleinere Modelle viel schneller trainiert und ausgeführt werden. [2]

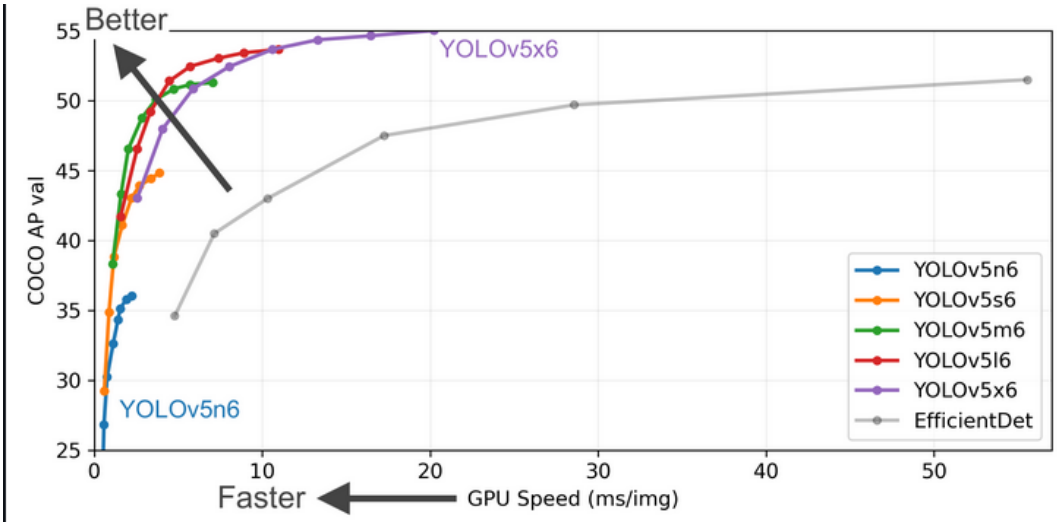


Abbildung 2 https://github.com/ultralytics/yolov5

Architektur YOLOv4 & YOLOv5

Die Architektur von YOLOv4 und YOLOv5 sind sich ähnlich und bauen auf denselben Grundkomponenten auf. Die drei Hauptkomponenten sind:

- Backbone (CSP-Darknet53)
- Neck (SPP und PANet) und
- Head (YOLOv3 Head).

Dabei wird die CSP-Darknet53-Architektur als Backbone verwendet. Dieser ist dafür zuständig Schlüsselmerkmale aus dem Eingangsbild zu extrahieren. Der Neck besteht aus zwei Hauptkomponenten. Einmal dem Spatial Pyramid Pooling (SPP) und dem Path Aggregation Network (PANet). Hierbei werden die Merkmale aus dem Backbone verwendet und optimiert. Der Head ist für die abschließende Objekterkennung verantwortlich wie zum Beispiel für die Berechnung der Bounding Box-Koordinaten, die Aktivierungsfunktionen für die Confidence Probability und die Anwendung von Non-Maximum Suppression (NMS) zur Reduzierung redundanter Vorhersagen. Der Hauptunterschied zwischen den beiden Versionen liegt in ihrer Implementierung und Optimierung. Davor werden diese einzelnen Konzepte, die beschrieben wurden, etwas näher erläutert.

CSP-Darknet53

Der Backbone besteht aus einer Kombination aus zwei verschiedenen Konzepten. Einmal für Cross Stage Partial und dem Darknet53.

Cross Stage Partial:

Besteht aus einer Reihe von Convolutional Layers, um den Informationsfluss in zwei separate Pfade aufzuteilen. Ein Hauptzweig und ein Nebenzweig. Die Verantwortung für die Hauptverzweigung liegt in der Berechnung des Hauptmerkmals. Der Nebenzweig ist dafür zuständig, die Informationen des Hauptzweiges zu verfeinern. Diese Kombination aus Hauptzweig und Nebenzweig bietet die Möglichkeit, Netzwerkressourcen besser zu nutzen. Es verbessert auch die Lernfähigkeit von Convolutional Neural Network.

Darknet53

Das Darknet53 zeichnet sich durch Tiefe und Kapazität zur Merkmalextraktion aus. Es besteht aus 53 Schichten, so wie in der unteren Abbildung zu sehen ist. Jede Schicht ist für eine bestimmte Aufgabe verantwortlich. Darknet53 besteht aus einer Sequenz von Convolutional Layern und Residual Layern, die wiederum auch aus Convolutional Layern besteht. Die Architektur setzt diese Struktur fort und erhöht schrittweise die Anzahl der Ausgabekanäle in den Convolutional Layern, um detaillierte Merkmale extrahieren zu können. Diese Convolutional Layer verwenden die Leaky ReLU (Rectified Linear Unit) Aktivierungsfunktion.

Spatial Pyramid Pooling (SPP)

Wird im Neck verwendet und ist dafür zuständig, Bilder in unterschiedlichen Größen zu verarbeiten. In der unteren Abbildung wird das Eingangsbild vom Convolutional Layer verarbeitet und man erhält Feature Maps. Das SPP teilt diese Feature Maps in verschiedene Bereiche und führt sogenannte Pooling Operationen durch. Bei den drei Schichten erhält man je nach a Aufteilungen a*256-dimensionale Vektoren. Diese werden anschließend zu einem eindimensionalen Vektor umgewandelt. [3]

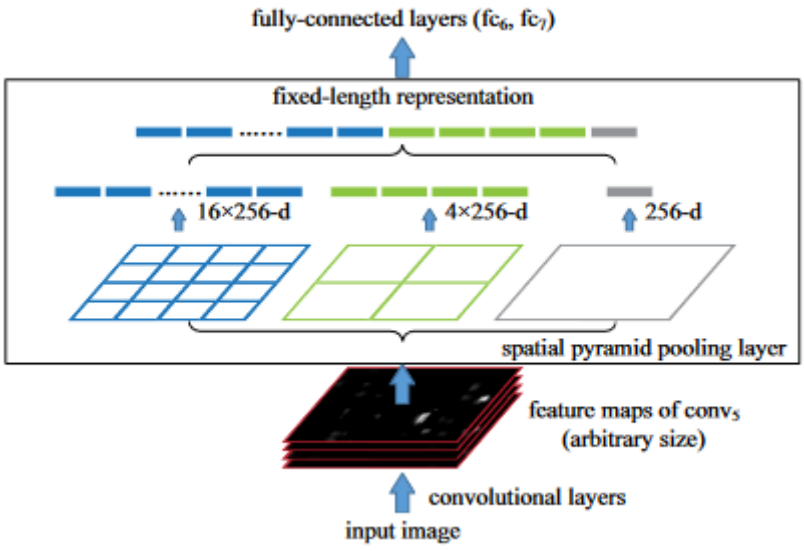


Abbildung 3 Architektur eines SPP [3]

Path Aggregation Network (PANet)

Ist der Hauptbestandteil des Necks, der eine genauere Objekterkennung für Objekte unterschiedlicher Größe ermöglicht. PANet nutzt Pyramid Network (FPN)-Funktionen als Kernarchitektur und erweitert diese um weitere Schichten. Dieser besteht aus einem Bottom-Up-Pfad und einem Top-Down-Pfad. Im Bottom-up-Pfad werden Features gesammelt, verarbeitet und auf verschiedenen Ebenen in ihrer Größe reduziert. Dieser besteht aus einer Folge von Convolutional Layern und einem Top-Down-Pfad. Beim Bottom-up-Pfad werden auf den verschiedenen Ebenen Merkmale erfasst, verarbeitet und die Merkmalsgröße verringert. Diese besteht aus einer Abfolge von Convolutional Layern sowie dem Top-down-Pfad. Dieser ist zuständig, um Upsampling-Operationen durchzuführen, um die Merkmalskarten auf höhere Auflösungen zu bringen. Zudem gibt es Lateral-Verbindungen, um die Merkmale aus dem Bottom-up-Pfad mit den hoch skalierten Merkmalen aus dem Top-down-Pfad zu verbinden. Schließlich wird der Bottom-Up-Pathway erneut verwendet, um detaillierte Informationen aus dem Bottom-Up- und Top-Down-Pathway zu kombinieren. Danach folgt ein Adaptive Feature Pooling, damit die verschiedenen Skalenebenen effektiv aggregiert werden. Danach werden in Fully-Connected Layern die Merkmale kombiniert und es können Vorhersagen getroffen werden.

Unterschied der beiden Versionen

- YOLOv5 ist im Gegensatz zu YOLOv4 benutzerfreundlichere und bietet eine leichtere Schnittstelle an.
- Zudem bietet es schnellere Trainingsergebnisse und erfordert weniger Rechenressourcen im Vergleich zu YOLOv4.
- Die fünfte YOLO Version bietet auch eine schnellere Inferenz Geschwindigkeit.

Datensätze

In YOLOv5 werden drei Verzeichnisse erwartet, jeweils für Trainings-, Validierungs- und Testdaten. Optional können auch Testdaten verwendet werden. Innerhalb dieser Verzeichnisse sollten zwei Unterordner existieren: ein Ordner für die Bilder und ein Ordner für die Annotationsdateien im ".txt"-Format. Jedes Bild muss eine Annotationsdatei mit demselben Namen haben.

Abbildung 4 zeigt ein Beispiel für das YOLO-Annotationsformat:

```
1 0.45775 0.4083333333333333 0.3595 0.336
7 0.469125 0.48533333333333334 0.29075 0.15166666666666667
```

Abbildung 4 YOLO-Annotationsformat

- Die erste Spalte enthält die Indexe der Objektklassen.
- Die zweite bis fünfte Spalte repräsentieren die Koordinaten der Bounding Box.
- Die Box-Koordinaten müssen normalisiert werden. Die Koordinaten sind wie folgt definiert:
    - x: Die x-Koordinate des Mittelpunkts der Bounding Box, relativ zur Bildbreite.
    - y: Die y-Koordinate des Mittelpunkts der Bounding Box, relativ zur Bildhöhe.
    - w: Die Breite der Bounding Box, relativ zur Bildbreite.
    - h: Die Höhe der Bounding Box, relativ zur Bildhöhe.

Die Annotationsdaten sollten somit entsprechend dieser Struktur vorliegen.

In dieser Studienarbeit wurden zwei Datensätze verwendet, die ursprünglich als JSON- und XML-Formate vorlagen. Um diese Datensätze für die Verwendung mit YOLOv5 und YOLOv4 anzupassen, wurden sie in das erforderliche TXT-Format konvertiert. Dabei wurde für jeden Datensatz ein "images"-Ordner für die Bilder und ein "label"-Ordner für die Annotationsdateien erstellt. Zudem wird eine YAML-Datei zur Konfiguration der YOLO-Modelle benötigt. Diese Datei enthält die Pfade zu den Trainings- und Testbildern und muss während des Trainings angegeben werden.

Die Daten wurden anschließend in die Unterverzeichnisse "train", "val" und "test" aufgeteilt, wobei das Splitting Verhältnis auf 70% für das Training, 15% für den Test und 15% für die Validierung festgelegt wurde. Auf diese Weise wurden die Daten entsprechend den Anforderungen von YOLOv5 und YOLOv4 organisiert, um eine effiziente Nutzung der Modelle zu ermöglichen. Für die Datenvorbereitung wurde ein separates Jupyter Notebook erstellt.

Vergleich der YOLO-Versionen

In diesem Abschnitt befassen wir uns mit dem Vergleich der beiden YOLO-Versionen, um festzustellen, welche Version für den spezifischen Anwendungsfall der Maskenerkennung auf verschiedenen Datensätzen besser geeignet ist. Es wird einmal für den großen Datensatz für alle Klassen und für die Face-Klassen verglichen. Dann auf den Dunklen Datensatz und anschließend auf die Einzelpersonen.

- Alle YOLOv5 Modelle arbeiten mit der YOLOv5s und die Batchsize beträgt 5.
- Alle YOLOv4 Modelle arbeiten mit yolov4-pacsp-s-mish.cfg und die Batchsize beträgt

Um das beste Modell für beide Versionen zu erhalten, wurde zuerst untersucht, wie viele Epochen das Training dauern sollte. Dazu wurden beide Versionen zunächst auf 100 Epochen trainiert. Dies half, festzustellen, ab welcher Epoche die Modelle konvergieren und ob ein Overfitting auftritt, wenn das Training weiter fortgesetzt wird. Um Overfitting zu verhindern, wurde auch eine "Early Stopping"-Methode implementiert. Für YOLOv5 wurde diese Methode in der YAML-Datei mit Parametern wie "patience" und "delta" festgelegt.

Die Abbildung 5 zeigte bei YOLOv5, dass der Wert für die "val/cls_loss" ab Epoche 70 zu steigen begann, was darauf hindeutet, dass das Modell zu diesem Zeitpunkt anfällig für Overfitting war. Basierend auf diesen Ergebnissen wurden die folgenden Epochen für die einzelnen Szenarien ausgewählt:

- Alle Klassen: 20 Epochen
- Gesichtsklassen: 20 Epochen
- Dunkler Datensatz: 50 Epochen
- Einzelpersonen: 35 Epochen

Der Vergleich wurde auch für YOLOv4 durchgeführt, wobei die Anzahl der Epochen bei 100 belassen wurde, wie in Abbildung 6 zu sehen ist. Grund dafür war das das Training sehr rechenintensiv war und für weniger Epochen keine weiteren Trainingsdurchläufe durchgeführt werden konnten.

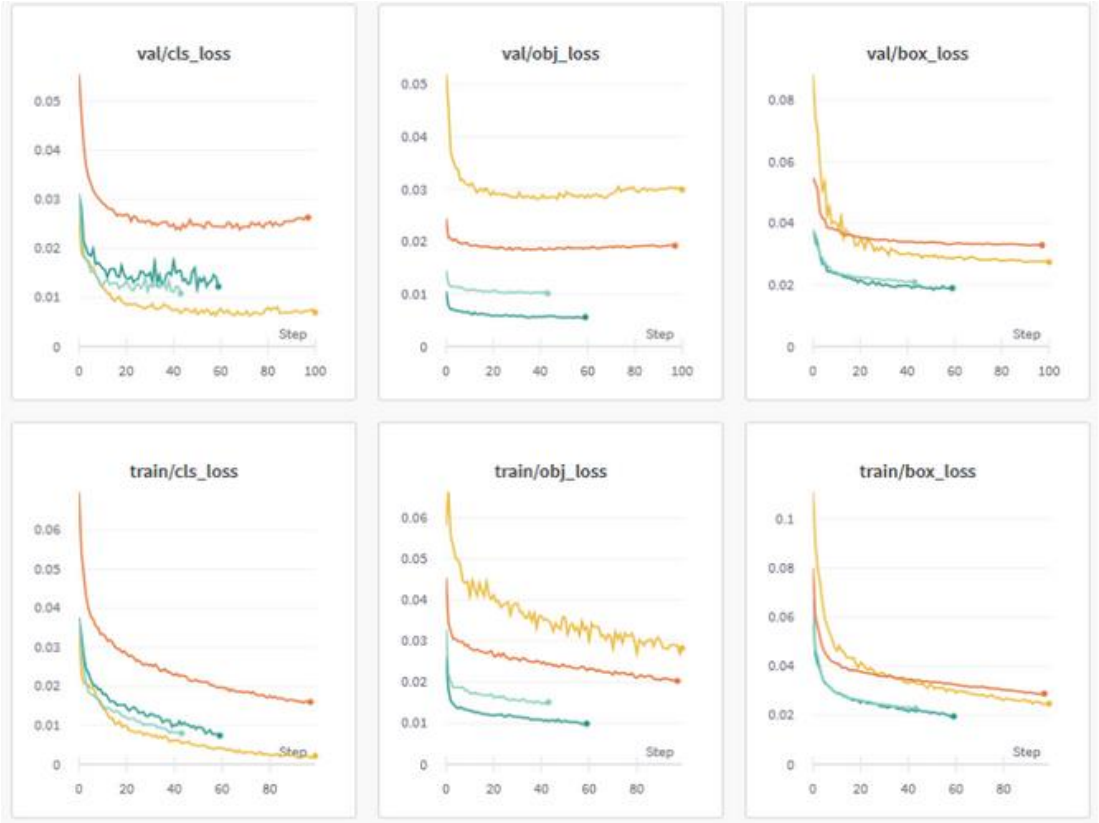


Abbildung 5 Rote Kurve: Alle-Klassen, Hellgrüne Kurve: Face-Klassen, Gelbe Kurve: Dark-Dataset, dunkelgrüne Kurve: Einzelperson von YOLOv5

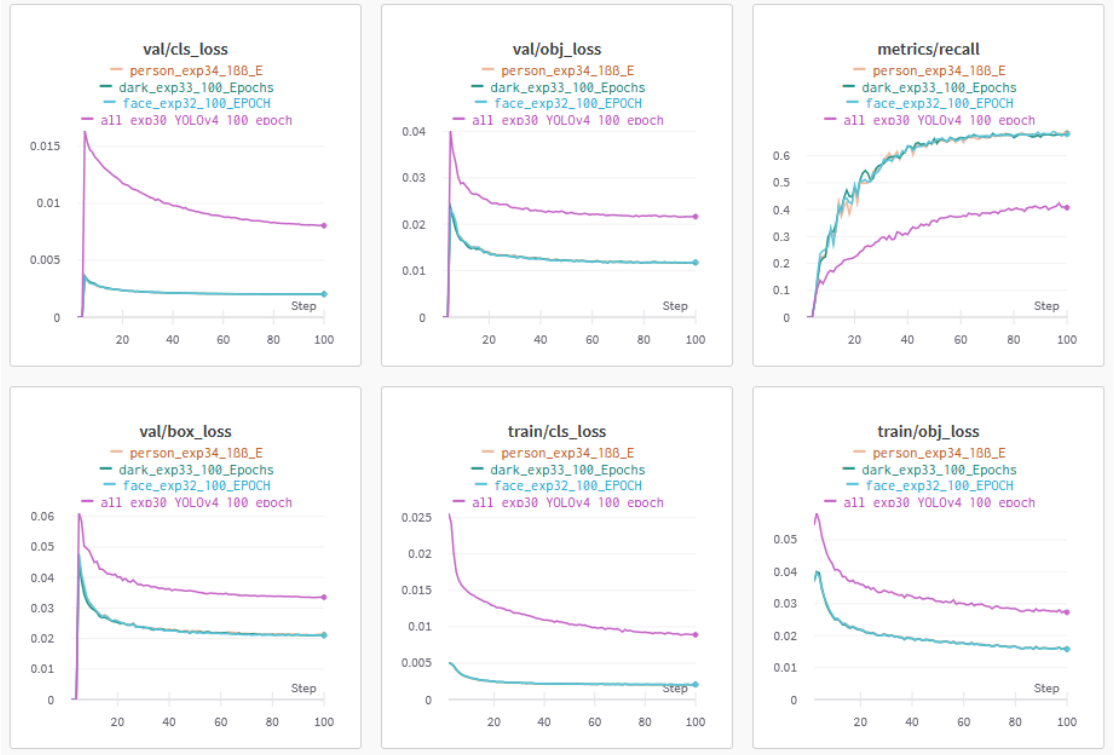


Abbildung 6 Lila Kurve: Alle-Klassen, dunkelgrüne: Dark-Dataset, türkise Kurve: Face-Klassen und gelbe Kurve: Einzelperson von Yolov4. Dunkelgrüne, gelbe und türkisene Kurve liegen fast aufeinander.

Vergleich aller Klassen gegen Gesichts Klassen

Bei der Abbildung 7 wurden zunächst die beiden YOLOv5-Modelle miteinander verglichen und das beste Modell wurde anschließend mit dem besten Modell der vierten Version verglichen. Dabei wurde festgestellt, dass das YOLOv5 Face-Modell die besten Ergebnisse erzielt hat. Dieses Ergebnis kann darauf zurückgeführt werden, dass YOLOv5 besser in der Lage ist, mit einem kleinen Datensatz und mehreren Klassen Vorhersagen zu treffen, wohingegen YOLOv5 bei größeren Datensätzen mit mehr Klassen möglicherweise mehr Schwierigkeiten hat. Daher ist das YOLOv5 Face-Modell, das nur vier Klassen enthält, besser für das Training geeignet.
Im Vergleich dazu zeigt sich, dass YOLOv4 mit mehr Klassen besser trainieren kann und somit für größere Datensätze mit umfangreichen Klassenstrukturen geeignet ist.

Modelle	Precision	Recall	mAP_0.5	mAP_0.5:0.95
YOLOv5_All	0.68	0.45	0.46	0.28
YOLOv5_Face	**0.87**	**0.64**	**0.73**	**0.52**

Modelle	Precision	Recall	mAP_0.5	mAP_0.5:0.95
YOLOv4_All	**0.21**	**0.41**	**0.33**	**0.20**
YOLOv4_Face	0.22	0.29	0.29	0.21

Modelle	Precision	Recall	mAP_0.5	mAP_0.5:0.95
YOLOv5_Face	**0.87**	**0.64**	**0.73**	**0.52**
YOLOv4_All	0.21	0.41	0.33	0.20

Abbildung 7 Ergebnisse vergleich aller Klassen und Gesichtsklassen

Vergleich des Dunklendatensatzes

In der untenstehenden Abbildung 8 ist deutlich zu erkennen, dass YOLOv5 das überlegene Modell ist. Es zeigt höhere Werte für Precision, mAP_0.5 und mAP_0.5:95. Die einzige Ausnahme bildet der deutlich höhere Recall-Wert von YOLOv4, während es bei den anderen Metriken deutlich schlechter abschneidet.

Modelle	Precision	Recall	mAP_0.5	mAP_0.5:0.95
YOLOv5_Dark	**0.66**	**0.58**	**0.59**	**0.35**
YOLOv4_Dark	0.22	0.68	0.29	0.21

Abbildung 8 Ergebnisse des Dunklendatensatzes

Vergleich der Einzelpersonen

Auch hier in der Abbildung 9 ist klar ersichtlich, dass YOLOv5 die besten Ergebnisse erzielt. Interessanterweise zeigt sich, dass YOLOv4 bei Recall, mAP_0.5 und mAP_0.5:95 fast genauso gut abschneidet wie die fünfte Version, jedoch eine deutlich geringere Precision aufweist, ähnlich wie beim oberen Datensatz.

Modelle	Precision	Recall	mAP_0.5	mAP_0.5:0.95
YOLOv5_Person	**0.91**	**0.68**	**0.29**	**0.21**
YOLOv4_Person	0.22	0.68	0.28	0.20

Abbildung 9 Ergebnisse vergleich der Einzelpersonen

Fazit

Zusammenfassend lässt sich feststellen, dass YOLOv5 in allen betrachteten Datensätzen die beste Leistung erzielt hat. Die Modelle von YOLOv5 zeigten sowohl eine höhere Präzision als auch eine bessere Fähigkeit zur Objekterkennung im Vergleich zu YOLOv4. Die Entscheidung für YOLOv5 wurde durch die Tatsache unterstützt, dass das Training schneller verlief und weniger Rechenressourcen erforderte als YOLOv4.

Es ist jedoch zu beachten, dass YOLOv4 noch Potenzial hätte, mehr aus den Datensätzen herauszuholen. Aufgrund von Zeitbeschränkungen war es nicht möglich, eine umfassende Optimierung des Modells durchzuführen und eine Vielzahl von Modellen zu trainieren, um den optimalen Wert zu ermitteln. Dies könnte eine Erklärung dafür sein, warum YOLOv4 trotz einiger beeindruckender Ergebnisse nicht mit der Gesamtleistung von YOLOv5 mithalten konnte.

Faster R-CNN

Neben den Modellen YOLO (You Only Look Once) und SSD (Single Shot MultiBox Detector) wurde sich auch auf das Faster R-CNN (Region-based Convolutional Neural Network) konzentriert. Das Faster R-CNN-Modell ist bekannt für seine hohe Genauigkeit bei der Erkennung und Lokalisierung von Objekten in Bildern.

Die grundlegende Idee des Faster R-CNN besteht darin, eine effiziente und genaue Objekterkennung zu ermöglichen, indem es sowohl die Regionen, die potenzielle Objekte enthalten, vorschlägt (RPN), als auch diese Regionen genau klassifiziert und lokalisiert (RCNN).

Zunächst wird das Eingangsbild durch ein Convolutional Neural Network (CNN) geleitet, um Merkmale auf verschiedenen Ebenen des Netzwerks zu extrahieren. In der klassischen Faster R-CNN-Architektur wird häufig ein vortrainiertes CNN wie ResNet oder VGGNet als Backbone verwendet, um eine effektive Merkmalsextraktion zu gewährleisten.

Die RPN-Komponente ist ein eigenständiges Netzwerk, das auf den extrahierten Merkmalen des Backbones arbeitet. Es bewertet eine Vielzahl von rechteckigen Ankerboxen, die über das Bild verteilt sind, und generiert Region Proposals basierend auf deren Wahrscheinlichkeit, dass sie Objekte enthalten. Das RPN verwendet dazu sowohl Klassifikations- als auch Regressionsverlustfunktionen, um die Qualität der Region Proposals zu verbessern.

Die aus dem RPN generierten Region Proposals werden dann an den RCNN-Teil weitergegeben. Im RCNN werden die extrahierten Merkmale der Region Proposals durch RoI Pooling in feste Größen umgewandelt, um sie für die Klassifizierung und Lokalisierung zu verwenden. Diese Merkmale werden dann durch zusätzliche Schichten verarbeitet, um die endgültigen Vorhersagen für jedes Region Proposal zu generieren. [4]

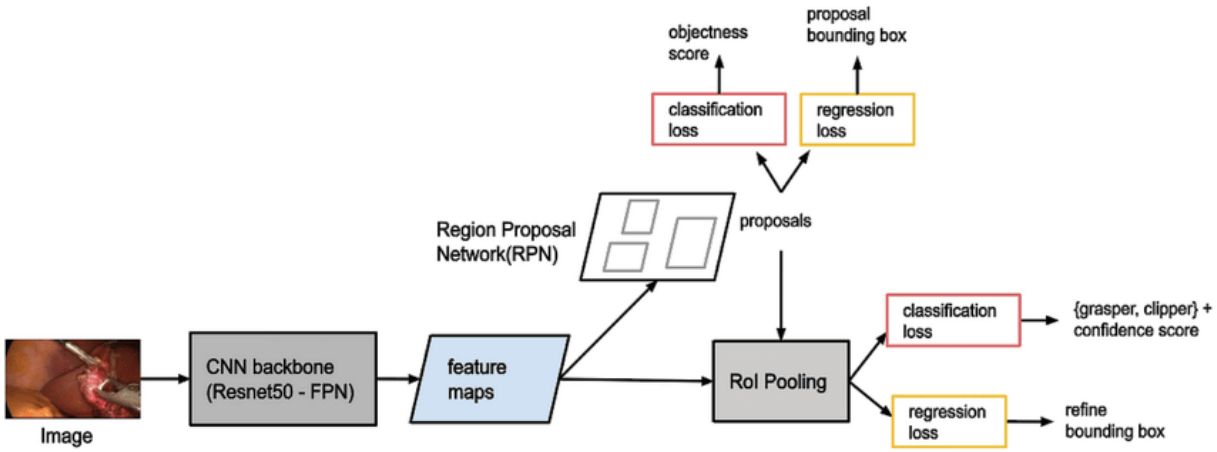


Abbildung 10 Aufbau Modell mit Resnet50 Backbone

https://www.researchgate.net/publication/349805499_Automation_of_surgical_skill_assessment_using_a_three-stage_machine_learning_algorithm

Um die Wirksamkeit des Faster R-CNN-Ansatzes zu bewerten, wurden zwei spezifische Modelle aus der PyTorch-Bibliothek verwendet: Faster R-CNN ResNet-50-FPN und Faster R-CNN MobileNetV3 Large-FPN. Diese Modelle bieten eine gute Balance zwischen Genauigkeit und Rechenleistung, wobei das Faster R-CNN ResNet-50-FPN-Modell auf dem leistungsstarken ResNet-50-Backbone basiert und das Faster R-CNN MobileNetV3 Large-FPN-Modell auf dem effizienten MobileNetV3 Large-Backbone aufbaut.

Die Wahl dieser beiden Modelle as der Faster RCNN Architekturfamilie ermöglicht es uns, sowohl die Genauigkeit als auch die Effizienz des Faster R-CNN-Ansatzes zu untersuchen. Während das Faster R-CNN ResNet-50-FPN-Modell für präzisere Ergebnisse bekannt ist, bietet das Faster R-CNN MobileNetV3 Large-FPN-Modell eine schnellere Ausführungsgeschwindigkeit und eine geringere Anforderung an die Rechenressourcen. Durch den Vergleich dieser beiden Modelle können wir wertvolle Einblicke in die Kompromisse zwischen Genauigkeit und Effizienz gewinnen.

Dataset-klassen

Im Zuge dieser Studienarbeit wurden zwei verschiedene Datensätze verwendet, die unterschiedliche Formate für die Annotationsinformationen bereitstellen. Um diese Datensätze in einer für das Modell verwendbaren Form zu bringen, wurden zwei Dataset-Klassen implementiert: MaskDetectionDatasetJSON und MaskDetectionDatasetXML. Hierfür wurde eine abgeleitete klasse der Abstrakten Klasse *Dataset* aus *torch.utils.data.Dataset* implementiert um kompatibilität mit den Toch Dataloader Klassen zu gewährleisten.

Die MaskDetectionDatasetJSON-Klasse dient der Verarbeitung des Datensatzes, bei dem die Annotationsinformationen im JSON-Format vorliegen. Bei der Initialisierung der Klasse werden das Stammverzeichnis des Datensatzes, eine Klassen-Label-Mapping-Datenstruktur und die gewünschte Zielgröße der Bilder angegeben. Optional können auch Flags gesetzt werden, um bestimmte Gesichtstypen zu filtern möglich ist hier die Verwendung von ausschließlich Einzelgesichter oder

Mehrfachgesichter. Die Methode load_annotations liest die Annotationsdateien aus dem Verzeichnis "annotations" ein und extrahiert die relevanten Informationen. Die Methode getitem lädt für einen gegebenen Index ein Bild und gibt die entsprechenden Zielinformationen (Boxen und Labels) zurück.

Die MaskDetectionDatasetXML-Klasse hingegen wird verwendet, um einen Datensatz mit Annotationsinformationen im XML-Format zu verarbeiten. Ähnlich wie bei der MaskDetectionDatasetJSON-Klasse werden beim Initialisieren das Stammverzeichnis des Datensatzes, das Klassen-Label-Mapping und die Zielgröße der Bilder angegeben. Die Methode load_annotations liest die Annotationsdateien aus dem Verzeichnis "annotations" ein und extrahiert die relevanten Informationen aus den XML-Dateien. Die Methode getitem gibt ein Bild und die entsprechenden Zielinformationen für den angegebenen Index zurück.

Hintergrund der Dataset Klassen war, dass das Faster R-CNN-Modell in pytorch spezifische Anforderungen an die Eingabe hat. Die Bilder sollten als Liste von Tensoren vorliegen, wobei jede Tensorform [*C*, *H*, *W*] hat und der Wertebereich im Bereich von 0 bis 1 liegen sollte. Dabei beschreibt *C* die Anzahl der Kanäle, *H* die Höhe der Bilder und *W* die Breite. Während des Trainings erwartet das Modell zusätzlich zu den Bildern eine Zielstruktur, die die Ground-Truth-Boxen und Labels enthält. Während der Inferenz gibt das Modell eine Liste von Dictionaries zurück, die die vorhergesagten Boxen, Labels und Scores für jede Detektion enthalten.

Vergleich der gewählten Modelle (Resnet-50 vs MobilenetV3-Large)

Das Faster R-CNN-Modell mit ResNet-50-FPN-Backbone basiert auf der Kombination zweier Schlüsselkomponenten: dem Region Proposal Network (RPN) und dem Region of Interest (RoI)-Head. Das ResNet-50-FPN-Backbone besteht aus einem ResNet-50-Netzwerk, das speziell für die Bildklassifizierung entwickelt wurde. Es besteht aus mehreren Residualblöcken, die es dem Netzwerk ermöglichen, tiefere Darstellungen der Eingangsbilder zu lernen. Darüber hinaus verwendet es das Feature Pyramid Network (FPN), um ein pyramidenförmiges Netzwerk von Merkmalen unterschiedlicher Skalierung zu erstellen. Diese Merkmale werden dann sowohl für die Generierung von Regionenproposals als auch für die Klassifikation und Regression der RoIs verwendet.

Im Gegensatz dazu verwendet das Faster R-CNN-Modell mit MobileNet die MobileNet-V3-Large-Architektur als Backbone. MobileNet-V3-Large ist ein schlankeres und leichteres Netzwerk, das für den Einsatz auf mobilen Geräten optimiert ist. Es verwendet Depthwise Separable Convolution, um effizientere Merkmalsextraktion bei geringerem Rechenaufwand zu ermöglichen. Dies ermöglicht eine schnellere Verarbeitung und eine geringere Anzahl an Parametern im Vergleich zum ResNet-50-FPN-Backbone. [5]

Ein weiterer Unterschied besteht in der Anzahl der Merkmalskanäle, die von den beiden Backbones erzeugt werden. Das ResNet-50-FPN-Backbone erzeugt Merkmale mit 256 Kanälen, während das MobileNet-V3-Large-Backbone Merkmale mit 128 Kanälen erzeugt. Dieser Unterschied kann Auswirkungen auf die Fähigkeit der Modelle haben, feine Details in den Bildern zu erfassen und kann sich auf die Genauigkeit bei der Erkennung kleiner Objekte auswirken.

Modelltrainig

Das Training der Modelle erfolgt in der Datei "training models.ipynb", während die Auswertung in der Datei "model eval.ipynb" durchgeführt wird. Zur Implementierung des Trainingsprozesses wurde die Funktion "train_one_epoch" aus der Datei "engine.py" des PyTorch-Vision-Repositories verwendet. Diese Funktion wird verwendet, um ein Modell für eine Epoche zu trainieren. Sie nimmt das Modell,

den Optimizer, den Data Loader, das Gerät (CPU oder GPU), die Epoche, die Druckfrequenz und einen optionalen Scaler (für die automatische Mischgenauigkeit) als Eingabe entgegen.

Die "train_one_epoch"-Funktion setzt das Modell in den Trainingsmodus und initialisiert einen Metriken-Logger, der für die Aufzeichnung von Verlusten und anderen Metriken während des Trainings verwendet wird. Die Funktion durchläuft den Data Loader, der die Trainingsdaten bereitstellt, und führt für jede Iteration den Vorwärts- und Rückwärtsdurchlauf durch. Dabei werden die Eingabebilder und Zielinformationen auf das entsprechende Gerät übertragen. Der Verlust wird berechnet und auf allen GPUs reduziert, um die Metriken zu aktualisieren und den Verlustwert zu erhalten. Die Gradienten werden zurückgesetzt, und der Optimizer führt entweder einen Schritt mit dem Scaler (falls vorhanden) oder direkt mit den Gradienten durch. Optional wird auch ein Lernratenplaner verwendet, um die Lernrate anzupassen. Am Ende jeder Epoche werden die Metriken und die Lernrate ausgegeben.

Die Funktion "evaluate" wird verwendet, um das trainierte Modell auf dem Testdatensatz zu evaluieren. Sie nimmt das Modell, den Data Loader und das Gerät als Eingabe entgegen. Die Funktion setzt das Modell in den Evaluierungsmodus und initialisiert einen Metriken-Logger. Sie durchläuft den Data Loader, führt den Vorwärtsdurchlauf für jede Iteration durch und sammelt die Vorhersagen des Modells. Anschließend werden die Vorhersagen mit den Ground-Truth-Informationen verglichen und die Leistung des Modells anhand verschiedener Metriken wie dem "Intersection over Union" (IoU) bewertet. Am Ende werden die aggregierten Statistiken und die Zusammenfassung der Evaluierung ausgegeben.

Für das Training der Modelle wurde eine Trainingsgröße von 80% verwendeten Gesamtdatensatzes angewandt. Das Training erfolgte über 10 Epochen, da sich in der Validierungsphase keine signifikanten Verbesserungen von Recall und Precision zeigten und der Loss bei allen Modellen ab diesem Punkt konvergierte.

Um das Training zu optimieren, wurde eine initiale Learning Rate von 0,005 festgelegt. Nach mehreren Versuchen wurde der Optimizer *torch.optim.SGD* (Stochasitc Gradient Descent) mit den folgenden Parametern verwendet, Momentum von 0,9 und Weight Decay von 0,0005. Durch das Ausprobieren verschiedener Werte für diese Parameter konnte festgestellt werden, dass diese Kombination die besten Ergebnisse lieferte.

Die Batch Size wurde auf 12 festgelegt, was bedeutet, dass in jedem Trainingsschritt 12 Bilder gleichzeitig verarbeitet wurden. Größere Batchsizes führten teilweise zu schlechteren ergebnissen. Die Maximale probierte Batchsize war bei Modellen mit Mobilenet 30 und bei Resnet 16.

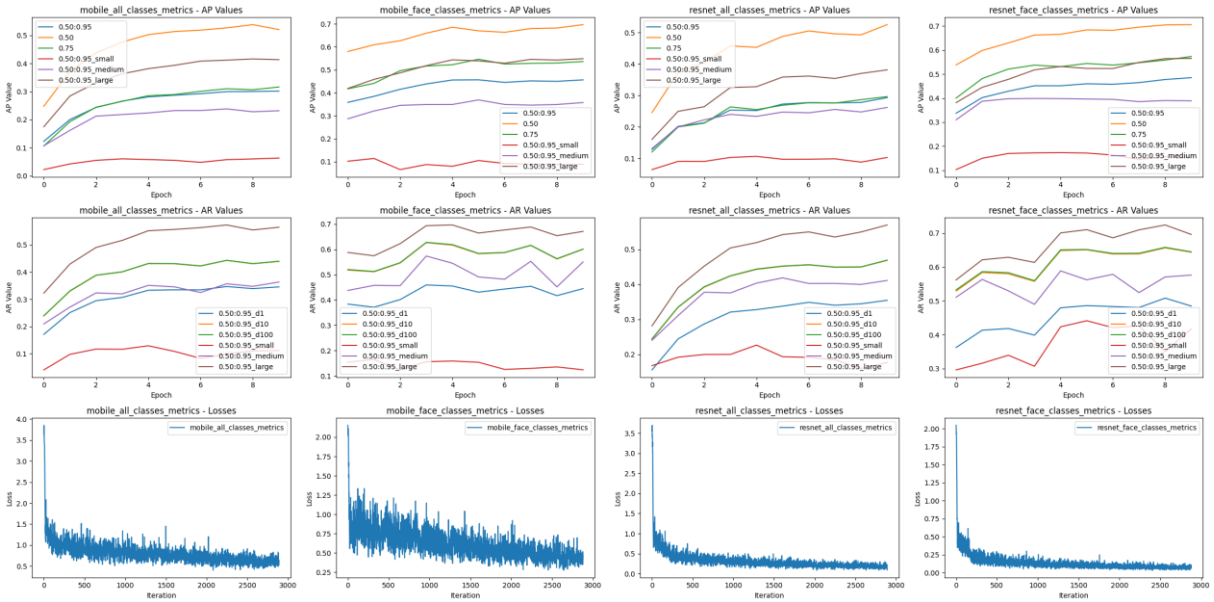
Außerdem wurden die Bilder auf eine Große von 512x512 resized. Durch Probieren mehrerer Größen ergab sich diese Größe als die beste hinsichtlich der Ergebnisse und der Speicherbedarfsreduktion die dadurch erreicht wurde.

Ergebnisse

Im Modelltraining wurden beide Modelle, mit ResNet und MobileNet Backbone, auf insgesamt 3 Testfälle trainiert. Zunächst wurde das Training auf den JSON Datensatz durchgeführt und 20 verschiedene Klassen enthält, nicht nur Gesichtsmasken. Anschließend wurde derselbe Datensatz verwendet, wobei jedoch alle irrelevanten Klassen entfernt wurden, sodass nur noch die Klassen "face_with_mask", "face_no_mask", "face_other_covering" und "face_with_mask_incorrect" übrigblieben. Dadurch sollte untersucht werden, welchen Unterschied es macht, welche Klassen für

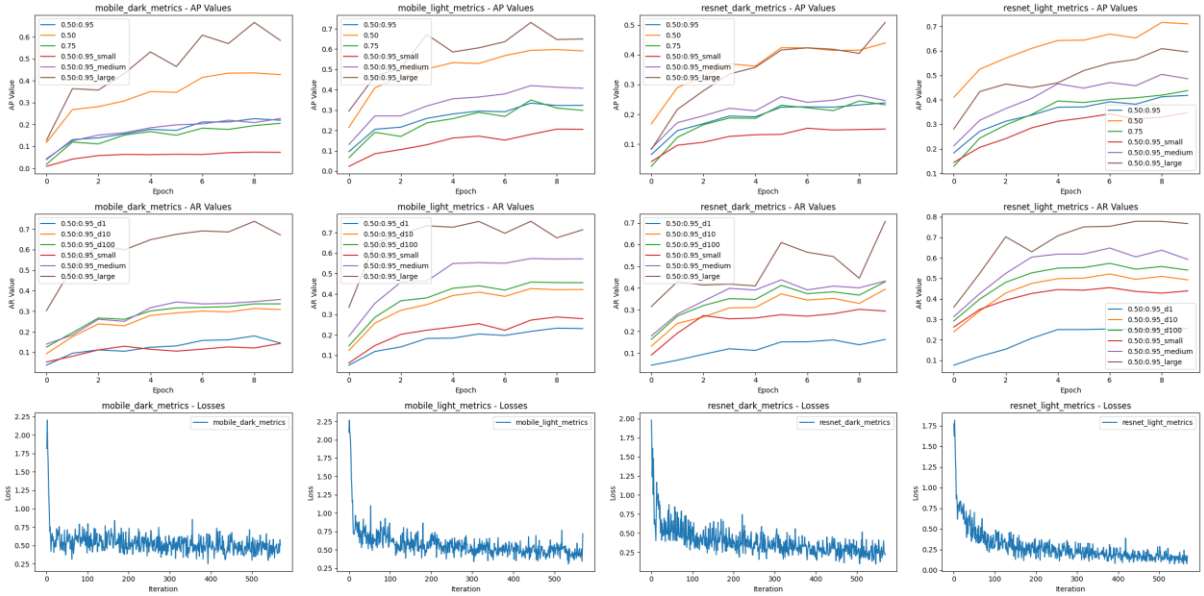
das Training verwendet werden und ob das Modell besser wird, wenn es ausschließlich auf die Erkennung von Gesichtsmasken trainiert wird.

Die Ergebnisse zeigten, dass insgesamt eine bessere Precision und Recall erreicht werden konnte, wenn das Modell speziell auf die Erkennung von Gesichtsmasken trainiert wurde. Bei einem Vergleich der beiden Architekturen erwies sich ResNet als etwas besser, insbesondere bei kleinen Bounding Boxes.



Im zweiten Vergleich wurden beide Modelle auf den XML Datensatz trainiert. Es gab eine Version des Datensatzes, die unverändert blieb, und eine Version, bei der die Bilder künstlich so verändert wurden, dass sie den Eindruck erweckten, im Dunkeln aufgenommen worden zu sein. Beide Modelle wurden sowohl auf die unveränderte als auch auf die künstlich veränderte Version des Datensatzes trainiert.

Die Ergebnisse auf den jeweils trainierten Datensätzen waren intuitiv: ResNet war immer etwas besser im Vergleich zu MobileNet auf demselben Datensatz. Wenn man die Leistung der Modelle auf den gleichen Architekturen zwischen dem dunklen und hellen Datensatz verglich, stellte man fest, dass die Erkennung auf den hellen Bildern immer spürbar besser war.



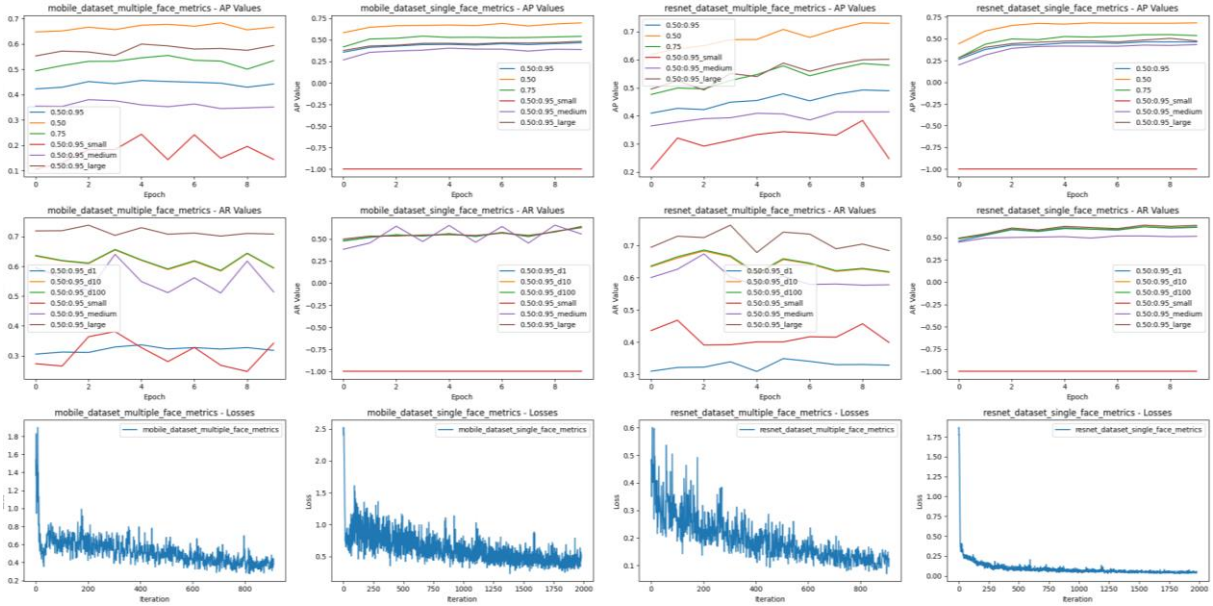
Eine weitere Untersuchung war die Anwendung der auf dem hellen Datensatz trainierten Modelle auf den dunklen Datensatz. Dabei zeigte sich, dass die Modellleistung deutlich abnahm, aber überraschenderweise war MobileNet hier etwas besser als ResNet. Umgekehrt, wenn Modelle, die auf dem dunklen Datensatz trainiert wurden, auf den hellen Datensatz angewendet wurden, waren sie immer noch verwendbar, wenn auch nicht so gut wie die Modelle, die direkt auf den hellen Datensatz trainiert wurden. In der Folgenden Darstellung werden die Vorhersagen der Modelle auf dem dunklen Datensatz verglichen. Jede Spalte steht für ein Modell. Die überschrift der Spalte zeigt,

auf welchen Datensatz das Modell trainiert wurde.

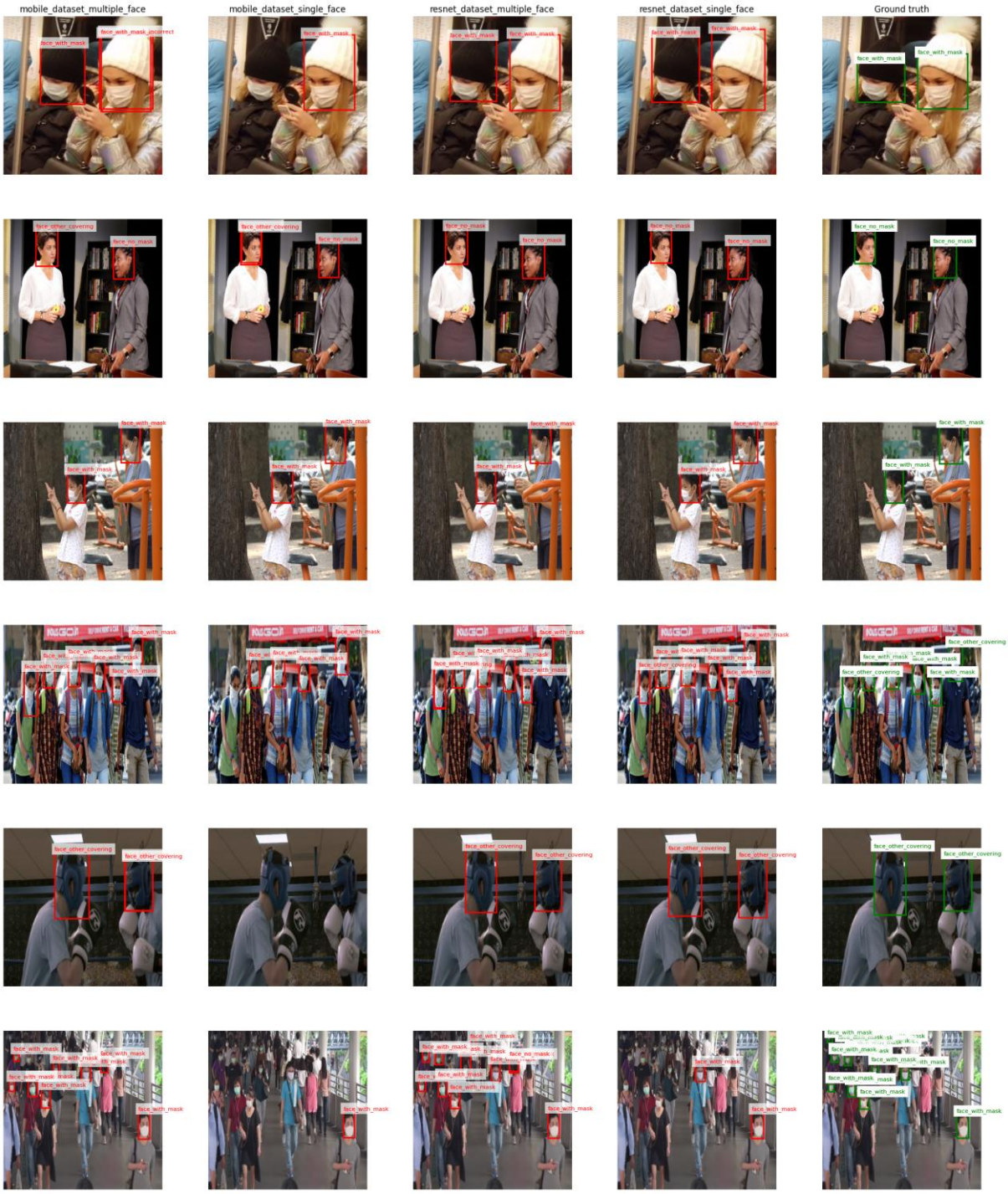
Im dritten Szenario wurde erneut der JSON Datensatz verwendet, jedoch nur mit den Gesichtsklassen ("face_with_mask", "face_no_mask", "face_other_covering", "face_with_mask_incorrect"). Diesmal wurden die Modelle jedoch ausschließlich auf Bildern trainiert, die nur eine einzelne Person abbilden. Ziel war es zu überprüfen, ob die Modelle immer noch gut in der Lage sind, Gesichtsmasken auf Bildern mit mehreren Personen zu erkennen. Dazu wurden zwei Datensätze erstellt, einer mit Bildern von einzelnen Personen und einer mit Bildern von mehreren Personen. Beide Modelle wurden auf dem Datensatz mit einzelnen Personen trainiert und auf dem Datensatz mit mehreren Personen angewendet.

Es stellte sich heraus, dass die Modelle, selbst wenn sie nur Bilder mit einzelnen Personen kannten, immer noch gut in der Lage waren, Gesichtsmasken auf Bildern mit mehreren Personen zu erkennen, wobei ResNet leicht bessere Ergebnisse erzielte. Die ergebnisse waren trotzdem schlechter als bei Modellen darauf trainiert wurden Masken bei mehrere Menschen zu erkenn.

Wenn die Modelle auf dem Datensatz angewendet wurden, auf dem sie trainiert wurden (also nur mit einzelnen Personen), waren beide Modelle ähnlich gut wie auf den anderen Datensätzen, wobei erneut ResNet leicht bessere Ergebnisse lieferte.


mobile_dataset_multiple_face_metrics - AP Values
mobile_dataset_single_face_metrics - AP Values
resnet_dataset_multiple_face_metrics - AP Values
resnet_dataset_single_face_metrics - AP Values
mobile_dataset_multiple_face_metrics - AR Values
mobile_dataset_single_face_metrics - AR Values
resnet_dataset_multiple_face_metrics - AR Values
resnet_dataset_single_face_metrics - AR Values
mobile_dataset_multiple_face_metrics - Losses
mobile_dataset_single_face_metrics - Losses
resnet_dataset_multiple_face_metrics - Losses
resnet_dataset_single_face_metrics - Losses
Epoch
Iteration
AP Value
AR Value
Loss
0.50:0.95
0.50
0.75
0.50:0.95_small
0.50:0.95_medium
0.50:0.95_large
0.50:0.95_d1
0.50:0.95_d10
0.50:0.95_d100
mobile_dataset_multiple_face_metrics
mobile_dataset_single_face_metrics
resnet_dataset_multiple_face_metrics
resnet_dataset_single_face_metrics

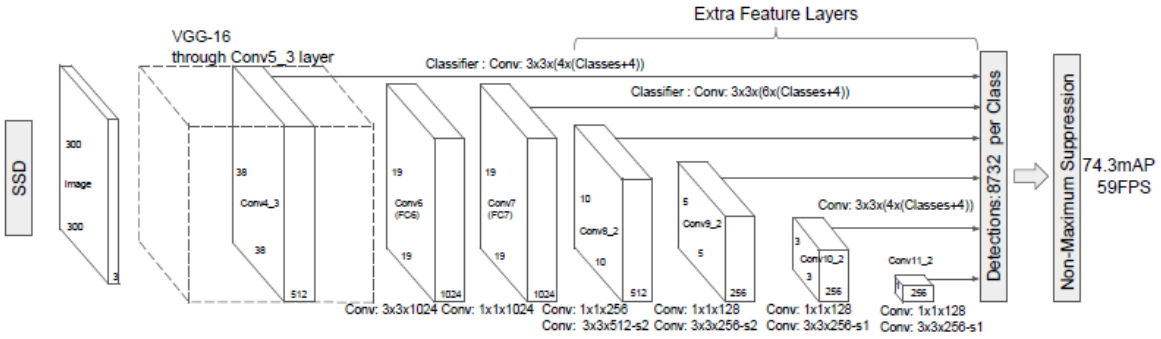
Die Folgende Abbildung zeigt die Anwendung der Modelle (Trainiert ausschlißlich auf einzelne oder ausschlißlich auf mehr Personen) auf den Datensatz mit ausschließlich mehrern Menschen im Bild.



SSD

Der Single Shot MultiBox Detector (SSD) ist ein leistungsstarkes Modell zur Objekterkennung in Bildern. Es wurde entwickelt, um effizient und präzise verschiedene Objekte in Echtzeit zu erkennen. Im Gegensatz zu anderen Ansätzen, die mehrere Regionen in einem Bild vorschlagen und klassifizieren, ist SSD ein einstufiges Verfahren, das auf einem einzigen Durchlauf durch das Netzwerk basiert.

Der Aufbau dieser SSD-Modelle besteht aus einem vorgeschulten Backbone-Netzwerk, wie zum Beispiel VGG16 oder MobileNetV3, das als Feature-Extraktor fungiert. Das Backbone-Netzwerk wird verwendet, um das Eingangsbild zu verarbeiten und Merkmale auf verschiedenen Ebenen des Bildes zu extrahieren. Diese Ebenen haben unterschiedliche räumliche Auflösungen und erfassen Merkmale auf verschiedenen Skalen.



https://pytorch.org/hub/nvidia_deeplearningexamples_ssd/

Anschließend werden auf jeder dieser Ebenen sogenannte "Convolutional Feature Maps" erzeugt. Diese Feature Maps enthalten Informationen über die vorhandenen Merkmale und ihre Positionen im Bild. Für jede Position auf den Feature Maps werden dann verschiedene Ankerboxen oder Prior-Boxes definiert. Diese Ankerboxen haben unterschiedliche Größen und Seitenverhältnisse und dienen dazu, potenzielle Objekte in unterschiedlichen Skalen und Aspektraten zu erfassen.

Die nächsten Schritte im SSD-Modell sind die Klassifizierung und die Feinabstimmung der Ankerboxen. Dafür werden auf den Feature Maps Convolutional Layers angewendet, um sowohl die Wahrscheinlichkeiten der verschiedenen Objektklassen als auch die Anpassungen der Ankerboxen an die tatsächlichen Objekte zu berechnen. Hierbei wird das SSD-Modell sowohl zur Klassifizierung als auch zur Regression verwendet. Die Klassifizierung erfolgt normalerweise mit einem Konvolutionslayer und einem nachgeschalteten Aktivierungsfunktion wie der softmax-Funktion, um die Wahrscheinlichkeiten der verschiedenen Objektklassen zu berechnen. Die Regression wird durchgeführt, um die Anpassungen der Ankerboxen an die tatsächlichen Positionen und Größen der Objekte im Bild zu bestimmen. Dafür werden separate Konvolutionslayer verwendet, die die Parameter für die Anpassungen der Ankerboxen schätzen.

Die Vorhersagen des SSD-Modells bestehen aus den erkannten Klassenlabels, den zugehörigen Wahrscheinlichkeiten und den Anpassungen der Ankerboxen. Um die endgültigen Objekterkennungen zu erhalten, werden Schwellenwerte für die Wahrscheinlichkeiten und für die Überlappung der Ankerboxen mit den tatsächlichen Objekten festgelegt. Nur die Vorhersagen, die diese Schwellenwerte überschreiten, werden als endgültige Erkennungen akzeptiert.

SSD bietet mehrere Vorteile. Da es ein einstufiges Verfahren ist, ist es schneller als zweistufige Ansätze wie Faster R-CNN. Das Modell kann Objekte in Echtzeit erkennen und ist daher für

Anwendungen in Echtzeit-Objekterkennungssystemen geeignet. Außerdem ist SSD sehr effizient, da es Ankerboxen auf verschiedenen Ebenen verwendet und dadurch Objekte unterschiedlicher Skalen und Aspektraten erfasst. Es ist auch robust gegenüber variierenden Objektgrößen und -positionen im Bild. Ein potenzieller Nachteil von SSD ist, dass es möglicherweise nicht so genau ist wie zweistufige Ansätze. Durch die Verwendung von Ankerboxen können bestimmte Objekte möglicherweise nicht gut erfasst werden, insbesondere solche mit ungewöhnlichen Aspektraten oder kleinen Größen. Ein weiterer Nachteil ist, dass SSD eine feste Anzahl von Ankerboxen für jede Feature-Map-Ebene verwendet, was die Flexibilität des Modells einschränken kann.

SSD300_VGG16

Das SSD300 mit dem VGG16-Backbone ist eine spezifische Variante des Single Shot MultiBox Detectors (SSD), bei der das VGG16-Netzwerk als Backbone verwendet wird. Der Aufbau des SSD300 besteht aus zwei Hauptkomponenten: dem Feature-Extraktor (Backbone) und dem Detektionshead.

Das VGG16-Netzwerk dient als Feature-Extraktor und besteht aus mehreren Convolutional Layern und Fully Connected Layern. Das Netzwerk wurde zuvor auf einem großen Klassifikationsdatensatz vortrainiert, z. B. dem ImageNet-Datensatz. Das VGG16-Netzwerk zeichnet sich durch seine Fähigkeit aus, verschiedene Merkmale auf verschiedenen Skalen zu extrahieren, was für die Objekterkennung von Vorteil ist.

Der Detektionshead ist für die Generierung von Vorhersagen auf den verschiedenen Ebenen der Feature Maps verantwortlich. Das SSD300 verwendet mehrere Feature Maps mit unterschiedlichen räumlichen Auflösungen, die auf den verschiedenen Ebenen des VGG16-Backbones generiert werden. Diese Ebenen werden als Konvolutionsebenen des VGG16-Netzwerks verwendet und erfassen Merkmale auf verschiedenen Skalen.

Das VGG16-Netzwerk als Backbone bietet eine gute Balance zwischen Genauigkeit und Geschwindigkeit. Darüber hinaus profitiert der SSD300 von der Verwendung des VGG16-Netzwerks, das auf einem großen Klassifikationsdatensatz vortrainiert wurde und dadurch gute Merkmale extrahieren kann. Darüber hinaus erfordert das VGG16-Netzwerk als Backbone mehr Berechnungsressourcen und Speicherplatz im Vergleich zu leichteren Backbone-Netzwerken.

SSDLite320_MobileNetv3

Das SSDLite320 mit dem MobileNetV3-Backbone ist eine spezifische Variante des Single Shot MultiBox Detectors (SSD), bei dem das MobileNetV3-Netzwerk als Backbone verwendet wird. Dieses Modell kombiniert die Vorteile des SSD-Frameworks mit den effizienten Merkmalsextraktionseigenschaften des MobileNetV3-Netzwerks. Der Aufbau des SSDLite320 ähnelt dem des SSD300, da er auch aus einem Feature-Extraktor (Backbone) und einem Detektionshead besteht. Das MobileNetV3-Netzwerk wird als Feature-Extraktor verwendet, um die Merkmale des Eingangsbildes auf verschiedenen Ebenen zu extrahieren. Das MobileNetV3-Netzwerk zeichnet sich durch seine Leichtigkeit und Effizienz aus, was es ideal für ressourcenbeschränkte Umgebungen macht. Es wurde speziell entwickelt, um eine gute Balance zwischen Genauigkeit und Recheneffizienz zu erreichen.

Der Detektionshead im SSDLite320 generiert Vorhersagen auf den verschiedenen Ebenen der Feature Maps, ähnlich wie beim SSD300. Ankerboxen werden auf jeder Ebene definiert, um potenzielle Objekte zu erfassen. Das SSDLite320 verwendet ebenfalls Ankerboxen verschiedener Größen und Seitenverhältnisse, um Objekte mit unterschiedlichen Skalen und Aspektraten zu erfassen.

Die Klassifikation und Regression werden auf den Feature Maps durchgeführt, um die Wahrscheinlichkeiten der Objektklassen zu berechnen und die Anpassungen der Ankerboxen an die tatsächlichen Objekte zu bestimmen. Das MobileNetV3-Netzwerk ermöglicht eine effiziente Merkmalsextraktion, während die Konvolutionslayer des Detektionsheads die spezifischen Vorhersagen generieren.

Im Vergleich zum SSD300 mit dem VGG16-Backbone bietet der SSDLite320 mit dem MobileNetV3-Backbone möglicherweise eine etwas geringere Genauigkeit, da das MobileNetV3-Netzwerk weniger komplexe Merkmale extrahieren kann. Dies kann dazu führen, dass einige Objekte mit feineren Details möglicherweise nicht so gut erkannt werden wie mit dem VGG16-Backbone. Allerdings wird dies durch die Vorteile der Recheneffizienz und der schnelleren Inferenzzeit ausgeglichen, was den SSDLite320 zu einer guten Wahl für Anwendungen in Echtzeit-Objekterkennungssystemen macht.

Programmaufbau

Das Programm ist so strukturiert, dass es mit der Erstellung eines Modells beginnt. Das Modell wird durch die Funktion "setup_model" initialisiert, bei der verschiedene relevante Variablen übergeben werden. Dazu gehören zunächst die gewünschte Modellvariante (SSD300 oder SSDLite320) sowie der Datensatz (JSON oder XML). Anschließend werden Parameter wie Batchsize, Momentum, Nesterov, Normalize und Resize festgelegt. Darüber hinaus wird die Zuordnung von Klassen ("class_mapping") definiert, die im weiteren Verlauf verwendet wird.

Während des Setups wird automatisch ein Trainings- und Test-Dataloader erstellt, wobei die zuvor festgelegten Größen berücksichtigt werden. Dabei werden die Bilder und die zugehörigen Annotationen als Zielwerte eingelesen und in Tensoren umgewandelt. Die Bildtensoren werden in eine Form von [C, H, W] gebracht, beispielsweise (3, 300, 300), wobei 3 die Anzahl der Kanäle repräsentiert (da das Bild im RGB-Format vorliegt), und H und W die Höhe und Breite des Bildes darstellen. Die Größe wird zuvor durch eine Umskalierungsfunktion festgelegt. Die Tensoren für die Bounding-Boxen müssen die Form (FloatTensor[N, 4]) aufweisen, wobei N die Anzahl der Bounding-Boxen ist und 4 die Ground-Truth-Boxen in Form von [x1, y1, x2, y2] repräsentiert. Die Labels werden als (Int64Tensor[N]) erwartet, wobei N für die Anzahl der Klassenlabel für jede Ground-Truth-Box steht. Es werden jedoch nur die Labels eingelesen, die im Klassenzuordnungs-Mapping hinterlegt sind. Bilder, die keine der im Klassenzuordnungs-Mapping definierten Labels aufweisen, werden nicht eingelesen. Auf diese Weise kann festgelegt werden, auf welche Klassen das Modell trainiert werden soll.

Während des Setups erfolgt eine entscheidende Schrittfolge, bei der die Bilder zunächst auf eine einheitliche Größe skaliert werden, bevor sie in Tensoren umgewandelt werden. Es ist jedoch wichtig zu beachten, dass dieser Skalierungsfaktor auch auf die Koordinaten der Bounding-Boxen angewendet werden muss.

Durch mehrere Versuche wurde festgestellt, dass eine bestimmte Reihenfolge beim Einlesen der Datensamples zu signifikant besseren Ergebnissen führt. Zunächst wird das Skalieren der Bilder auf die gewünschte Größe durchgeführt, bevor sie in Tensoren umgewandelt werden. Anschließend wird eine Normalisierung angewendet, bei der die Pixelwerte mit einem Mittelwert (MEAN) von [0.485, 0.456, 0.406] und einer Standardabweichung (STD) von [0.229, 0.224, 0.225] angepasst werden. Diese Normalisierung erfolgt nach der Umwandlung der Bilder in Tensoren, wobei die Werte auf den Bereich von 0 bis 1 skaliert werden.

Diese spezifische Reihenfolge und Kombination von Schritten hat sich als besonders effektiv erwiesen und führt zu verbesserten Ergebnissen bei der Verarbeitung der Daten.

Die Konfiguration des ausgewählten Modells ist der nächste Schritt. Für das SSD300_VGG16-Modell werden die vortrainierten Gewichte "SSD300_VGG16_Weights.Default" verwendet, die auf dem COCOV1-Datensatz trainiert wurden. Als Backbone dient das VGG16-Modell mit den "VGG16_Weights.Default", die auf dem Imagenet1k_V1-Datensatz trainiert wurden.

Für das SSDLite320_MobilenetV3-Modell werden "SSDLite320_MobileNet_V3_Large_Weights.Default" verwendet, die auf dem COCO_V1-Datensatz trainiert wurden. Als Backbone dient das MobileNetV3-Modell mit "MobileNet_V3_Large_Weights.DEFAULT", die auf dem IMAGENET1K_V2-Datensatz trainiert wurden.

Zuletzt wird der Optimizer mit den zuvor festgelegten Parametern initialisiert. Der Optimizer ist eine Methode, die während des Trainings verwendet wird, um die Modellparameter anzupassen und die Fehlerfunktion zu minimieren. Die Lernrate wird automatisch angepasst und startet hier dabei immer bei 0.00001. Bei größeren Werten für die Lernrate wurde häufig beobachtet, dass es zu Überanpassungen kommt, was zu starken Schwankungen im Verlust führt.

Das Training wird durch eine Funktion umgesetzt, die das zuvor konfigurierte Modell, den Trainings- und Test-Dataloader, den Optimizer, das Gerät (CPU oder GPU) für die Berechnungen und die Anzahl der zu trainierenden Epochen als Eingabe erhält. Das Gerät wird zuvor mit Hilfe der Anweisung `device = torch.device("cuda" if torch.cuda.is_available() else "cpu")` festgelegt.

Für jede der festgelegten Epochen wird die Funktion "train_one_epoch" aus der PyCocoTools-Bibliothek einmal ausgeführt. Diese Funktion liefert ein Dictionary mit allen berechneten Verlusten zurück. Nach Abschluss einer Epoche wird das Modell mithilfe der Funktion "evaluate" aus Coco_eval auf dem Test-Dataloader getestet. Dabei werden der mittlere Average Precision (mAP) und der mittlere Average Recall (mAR) für verschiedene IoU-Schwellenwerte für die aktuelle Epoche berechnet.

Nach Abschluss aller Epochen erhält man drei Dictionaries: Zwei davon enthalten den mAP und den mAR für jede Epoche, während das dritte alle Verluste für jeden Batch während des gesamten Trainings enthält.

Verwendete Lossfunktionen

Der hier verwendete Lokalisierungsverlust (Localization Loss) wird berechnet, um die Genauigkeit der Regression der vorhergesagten Bounding Boxes auf die entsprechenden Ground-Truth-Koordinaten positiv abgeglichener Objekte zu bewerten. Bei negativen Übereinstimmungen gibt es keine Ground-Truth-Koordinaten, was sinnvoll ist, da das Modell nicht darauf trainiert werden sollte, Boxen um leeren Raum zu zeichnen.

Der Lokalisierungsverlust wird durch den gemittelten Smooth L1-Verlust zwischen positiv abgeglichenen Lokalisierungskoordinaten und ihren Ground-Truth-Werten ausgedrückt.

$$L_{loc} = \frac{1}{n_{positives}} \left(\sum_{positives} Smooth\ L_1\ Loss \right)$$

https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Object-Detection

Jede Vorhersage hat ein Ground-Truth-Label, unabhängig davon, ob sie positiv oder negativ ist. Das Modell soll sowohl Objekte erkennen als auch das Fehlen von Objekten. Angesichts der Tatsache, dass in einem Bild normalerweise nur wenige Objekte vorhanden sind, besteht die Herausforderung darin, die Anzahl der negativen Übereinstimmungen in der Verlustfunktion zu begrenzen. Eine Lösung

dafür ist das sogenannte Hard Negative Mining. Dabei werden nur die Vorhersagen verwendet, bei denen es dem Modell am schwersten fällt zu erkennen, dass keine Objekte vorhanden sind. Die Anzahl der schweren Negativübereinstimmungen ist oft ein Vielfaches der Anzahl der positiven Übereinstimmungen. Durch die Auswahl der schwierigsten Negativübereinstimmungen wird der Confidence-Verlust bestimmt. Der Confidence-Verlust ist die Summe der Verluste der positiven Übereinstimmungen und der ausgewählten schweren Negativübereinstimmungen. Dabei wird der Verlust durch die Anzahl der positiven Übereinstimmungen gemittelt. Dieser Ansatz trägt dazu bei, das Modell sowohl in der Objekterkennung als auch im Erkennen des Fehlens von Objekten zu verbessern.

$$L_{conf} = \frac{1}{n_{positives}} \left(\sum_{positives} CE\ Loss + \sum_{hard\ negatives} CE\ Loss \right)$$

https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Object-Detection

```
return {
    "bbox_regression": bbox_loss.sum() / N,
    "classification": (cls_loss[foreground_idxs].sum() + cls_loss[background_idxs].sum()) / N,
}
```

Der Multibox-Verlust bzw. Total Loss ist eine Kombination aus zwei Verlusten, die im Verhältnis α aggregiert werden. In der Regel ist es nicht notwendig, einen spezifischen Wert für α festzulegen, da es sich um einen erlernbaren Parameter handeln kann. Im Fall des SSD-Modells wird jedoch α = 1 verwendet und somit addieren sich einfach die beiden Verluste.

$$L = L_{conf} + \alpha \cdot L_{loc}$$

https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Object-Detection

Nach Abschluss des Trainings wird der Verlauf des Losses über das Training hingweg grafisch dargestellt. Darüber hinaus wird die mAP und der mAR für die festgelegten IoU-Schwellenwerte jeder Epoche angezeigt, um einen guten Einblick in den Verlauf des Trainings und die Güte des Modells zu erhalten. Zusätzlich besteht die Möglichkeit, Vorhersagen des Modells mit den Ground-Truth-Bounding-Boxen auf Bildern aus dem Testdatensatz zu vergleichen. Dies bietet eine visuelle Überprüfung der Modellleistung.

Parameteroptimierung

Um die Güte des Modells zu optimieren, wurde eine Parameteroptimierung durchgeführt, bei der verschiedene Batch-Größen, unterschiedliche Momentum-Werte, die Verwendung des Nesterov SGD-Optimierers und die Nutzung der Normalisierung untersucht wurden.

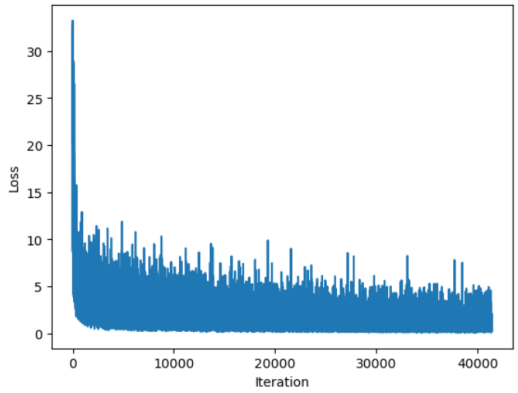
Es hat sich herausgestellt, dass eine Batch-Größe von 1 oder maximal 2 zu besseren Modellen geführt hat. Jegliche Größen darüber hinaus führten zu deutlich schlechteren Ergebnissen. Ein Momentum-Wert von 0.95 hat sich ebenfalls als optimal für diesen Fall erwiesen. Die Verwendung der Normalisierung und des Nesterov SGD-Optimierers haben in jedem Optimierungsschritt zu Verbesserungen des Losses und der mAP geführt.

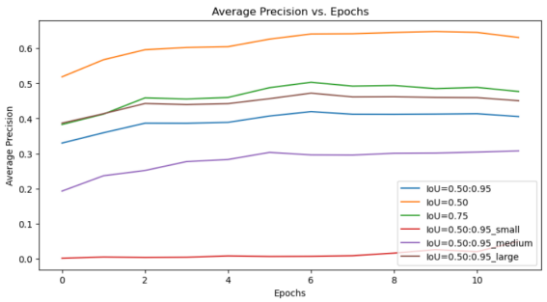
Evaluation

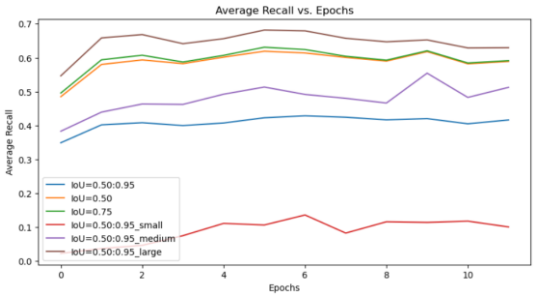
Vergleich der Modelle bei unterschiedlicher Inputauflösung

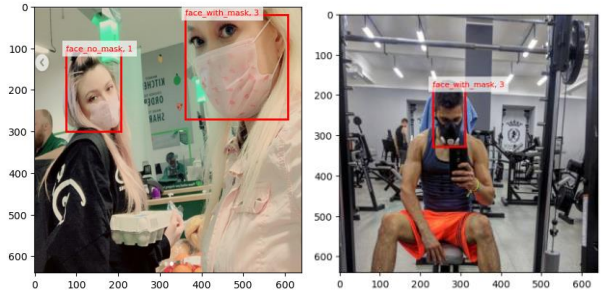
In einem ersten Experiment wurde untersucht, ob es einen Unterschied macht, ob die Bilder auf die Größe skaliert werden, für die das Modell ursprünglich trainiert wurde. Hierfür wurde beim SSD300-Modell unter Verwendung des JSON-Datensatzes und nur vier Klassen einmal eine Skalierung auf 640x640 und einmal auf die tatsächlichen 300x300 durchgeführt. Erstaunlicherweise lieferten beide Durchläufe sehr ähnliche Ergebnisse. Das Modell mit den Bildern, die auf 640x640 skaliert wurden, schnitt sogar minimal besser in Bezug auf die Präzision ab, jedoch war der Recall des Modells mit den Bildern, die auf 300x300 skaliert wurden, höher. Der durchschnittliche Loss war bei beiden Durchläufen gleich. Es konnte jedoch festgestellt werden, dass das Modell mit den auf 300x300 skalierten Bildern schneller trainiert wurde (4 Minuten unterschied).

Resolution: 640x640

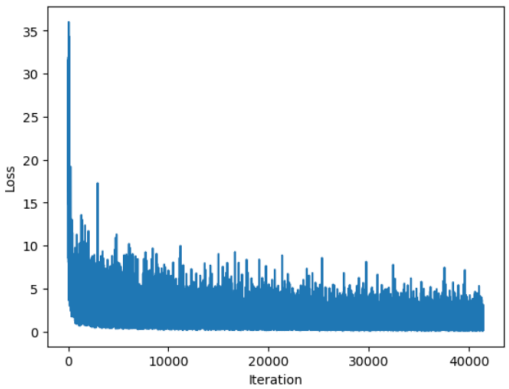


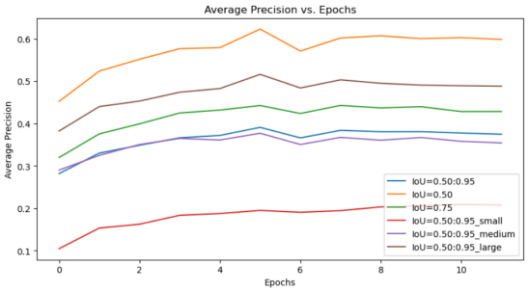


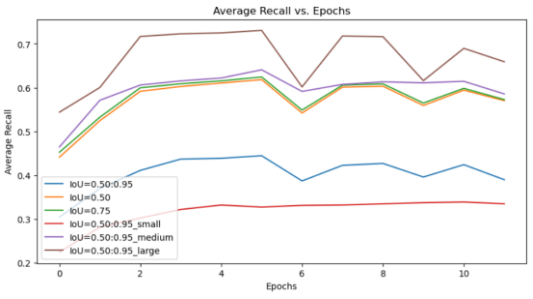


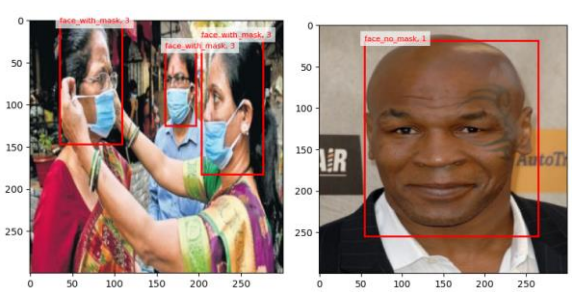


Resolution: 300x300



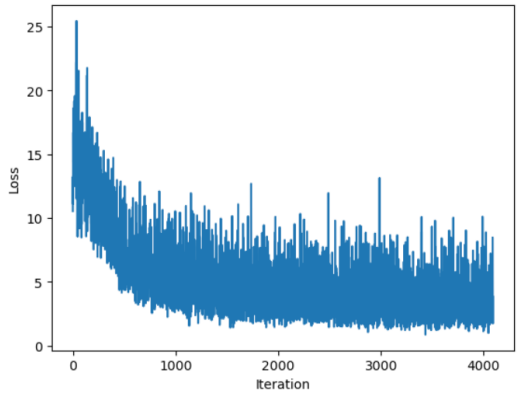


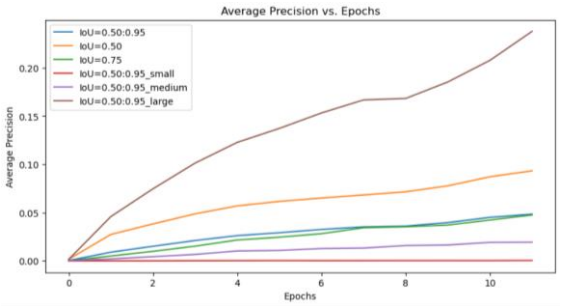


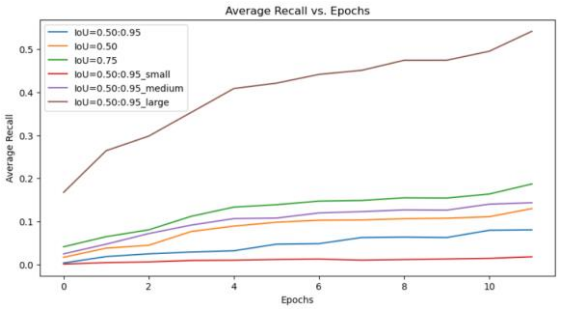


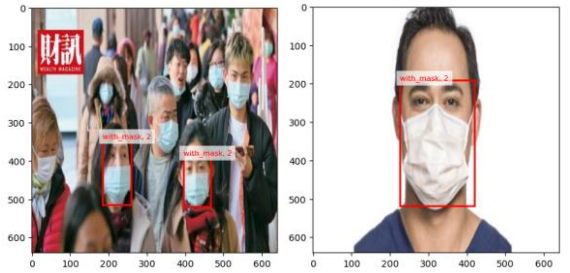
Das gleiche Experiment wurde auch für das SSD320Lite_MobileNetV3-Modell durchgeführt, wobei jedoch der XML-Datensatz verwendet wurde. Die Bilder wurden einmal auf 640x640 und einmal auf 320x320 skaliert. Hier hat sich deutlich gezeigt, dass Bilder, die auf die vorgesehene Größe skaliert wurden, deutlich bessere Ergebnisse liefern, obwohl sie deutlich kleiner sind (1/4). Der durchschnittliche Loss ist zwar relativ ähnlich, aber die Präzision sowie der Recall sind bei dem Modell mit einer Größe von 320x320 deutlich besser. Auch die Trainingsdauer ist aufgrund der Verwendung kleinerer Bilder schneller.

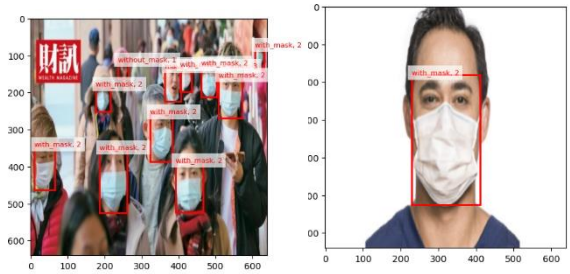
Resolution: 640x640



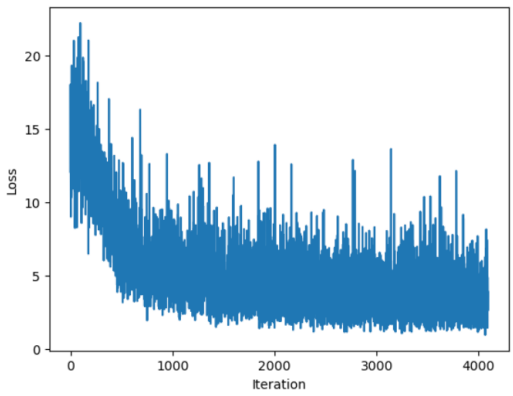


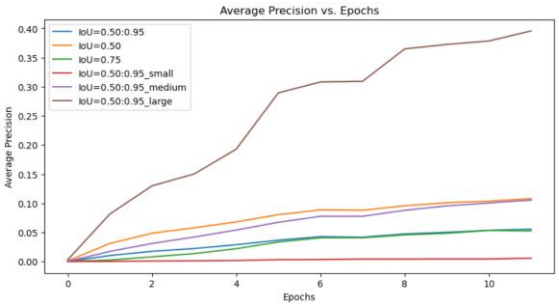


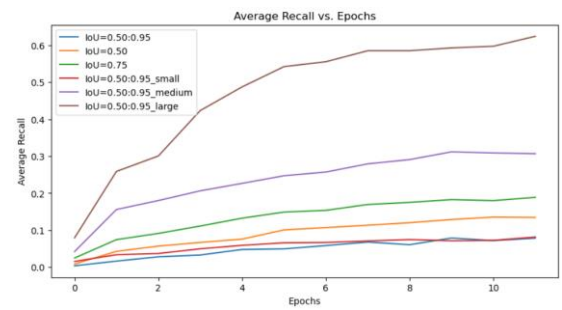


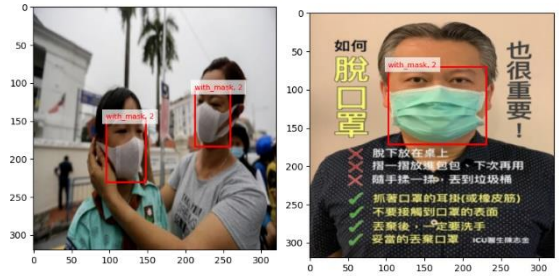


Resolution: 300x300





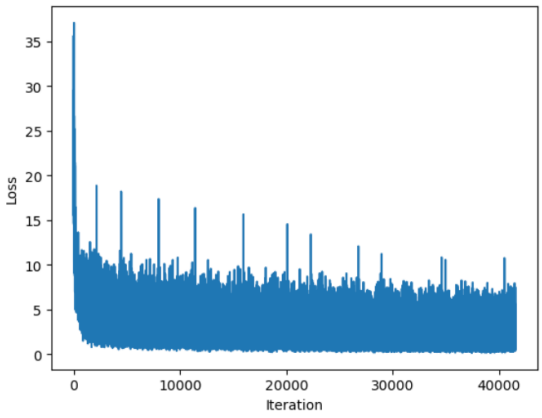


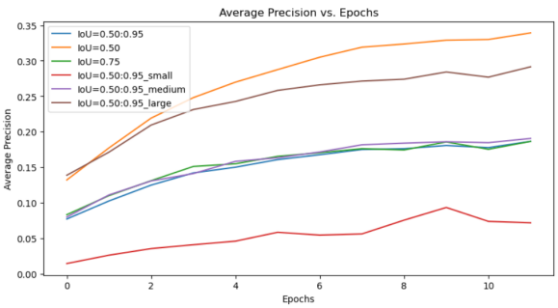


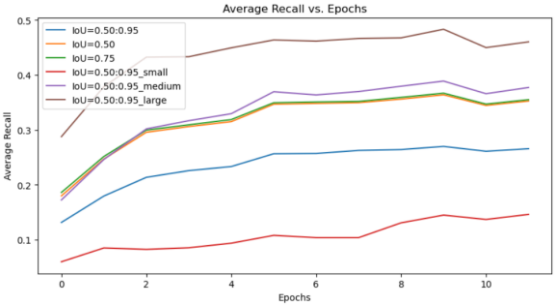
Vergleich zwischen ssd300 und ssdLite an Json Datensatz mit allen Klassen

Beim Vergleich des SSDLite320-MobileNetV3 mit dem SSD300-VGG16 auf einem trainierten JSON-Datensatz unter Verwendung aller Klassen wurde festgestellt, dass das SSDLite-Modell deutlich schlechtere Leistung erbracht hat als das SSD300-Modell mit dem VGG16-Backbone. Auch hier wurden die Bilder auf die Größe skaliert, die dem jeweiligen Modell entspricht, und das Training wurde über 12 Epochen durchgeführt. Leider verbesserte sich die Performance nicht signifikant, und die Trainingszeit für das SSDLite-Modell betrug nur etwa 43 Minuten im Vergleich zu 47 Minuten für das SSD300-Modell bei einem Datensatz von über 4300 Fotos. Der Loss des SSD300-Modells ist erkennbar geringer und konzentrierter als der des SSDLite-Modells, und die Genauigkeit ist fast doppelt so hoch. Zusammenfassend kann gesagt werden, dass das SSDLite-Modell hier sowohl in Bezug auf die Trainingszeit als auch auf die Leistung etwas enttäuscht hat.

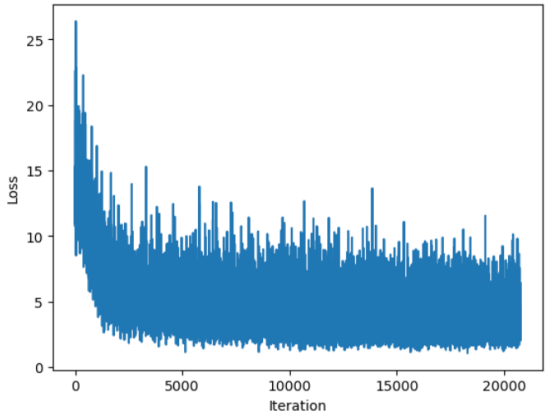
SSD300

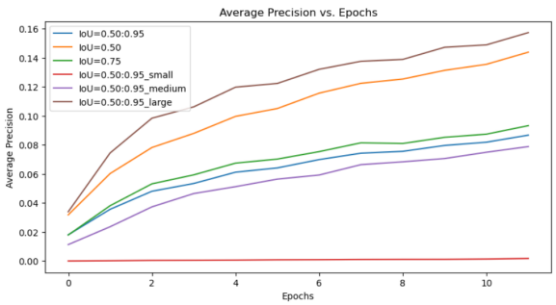


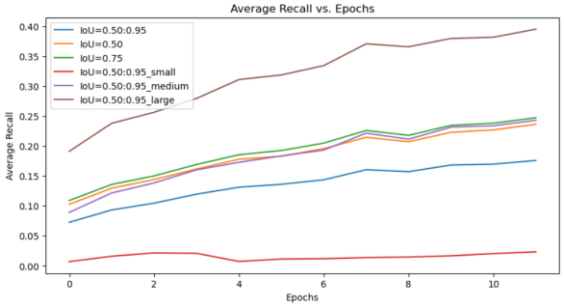


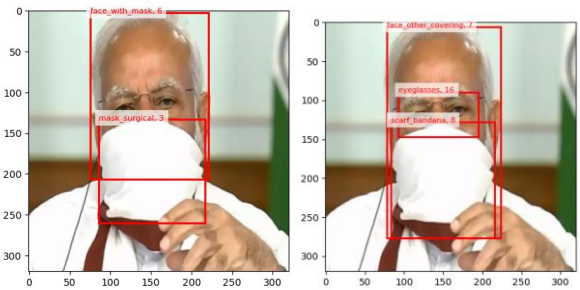


SSDLite





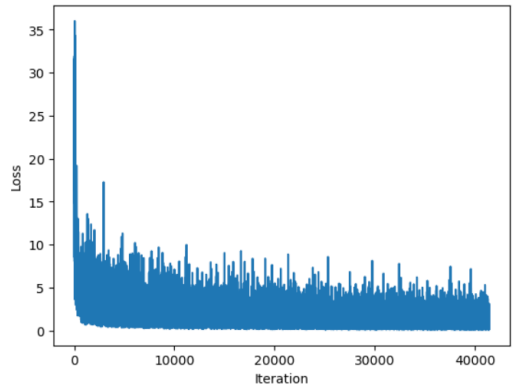
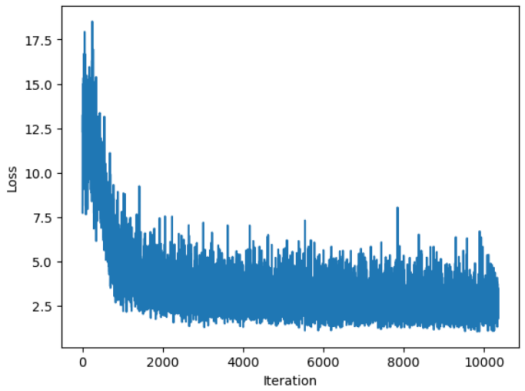
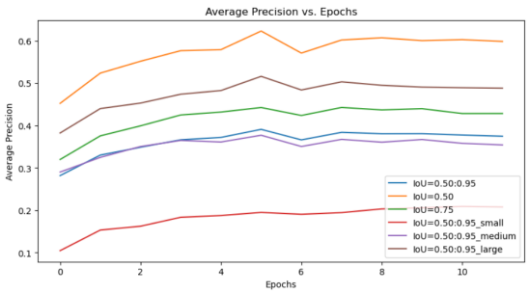
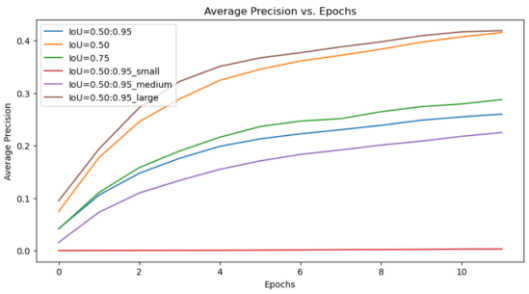
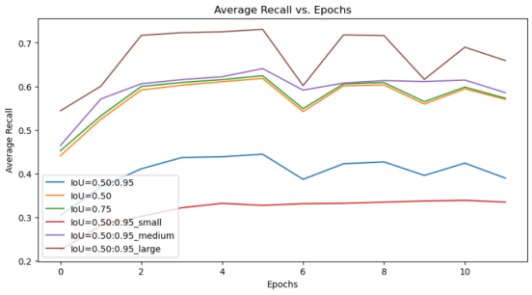
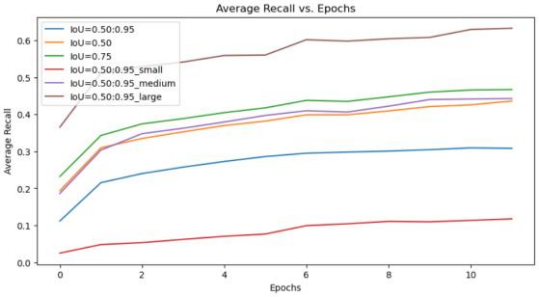
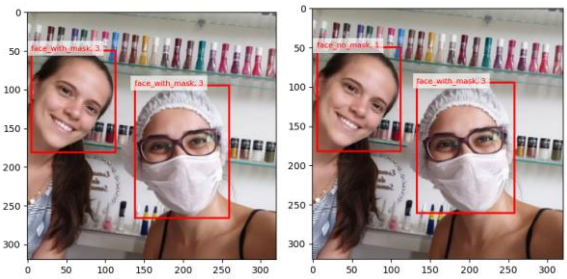




Vergleich zwischen ssd300 und ssdLite an Json Datensatz mit nur vier Klassen

Es wurde beschlossen, die Modelle nur auf die relevanten Klassen im JSON-Datensatz zu trainieren, da die Erkennung von Mundschutzen von besonderer Bedeutung ist. Daher wurden alle anderen 16 Klassen verworfen. Dies hatte den Zweck sicherzustellen, dass sich das Modell stärker oder ausschließlich auf die wichtigen Klassen konzentriert und keine Anpassungen für die fehlerhafte Erkennung anderer Klassen vornimmt. Genau dieser Gedanke hat sich bestätigt, da beide Modelle mit nur 4 Klassen deutlich besser performt haben. Dies zeigte sich sowohl in der Trainingsdauer als auch in den Ergebnissen der Evaluation.

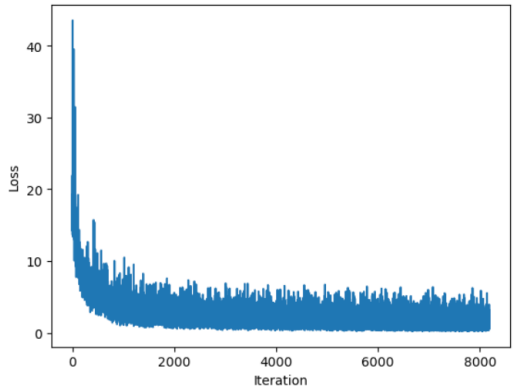
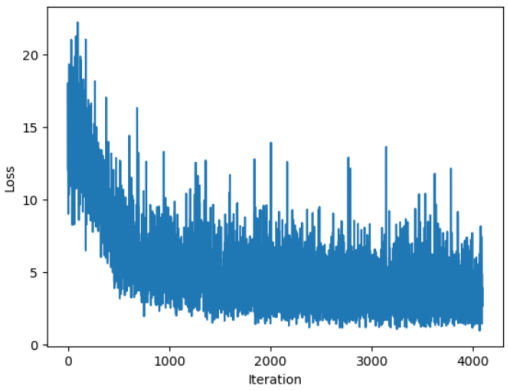
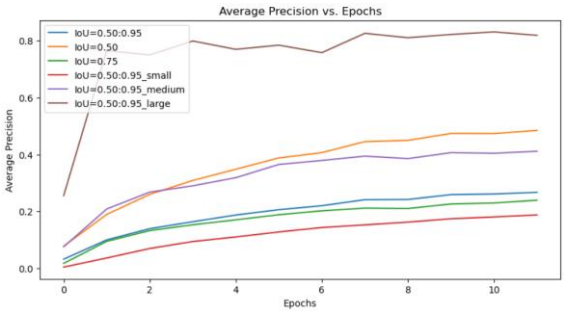
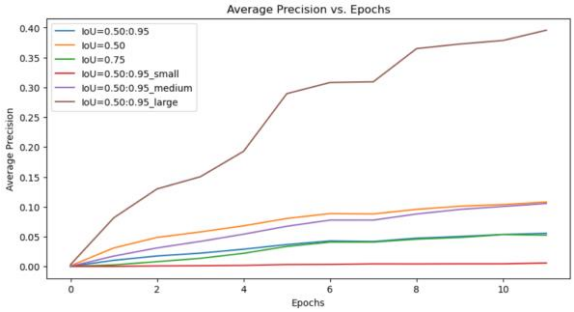
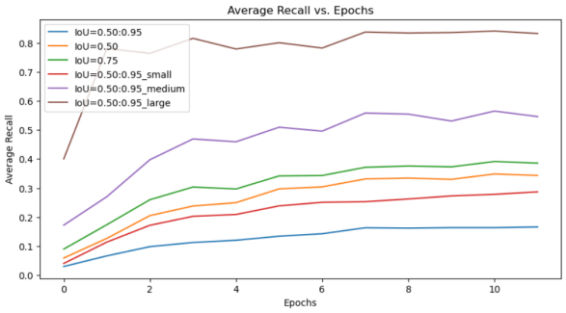
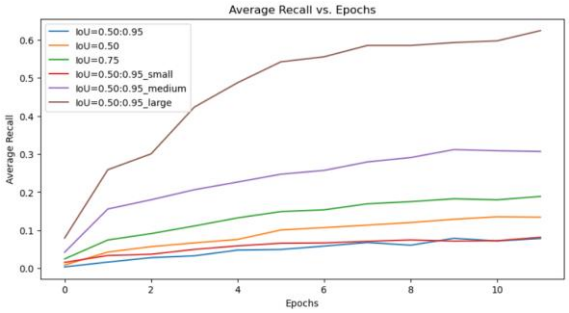
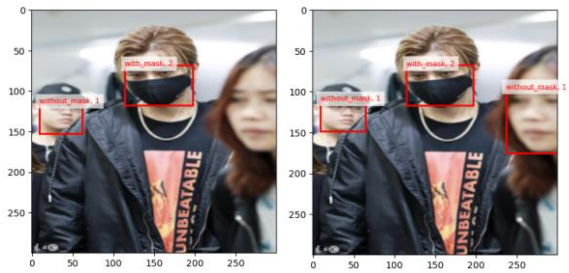
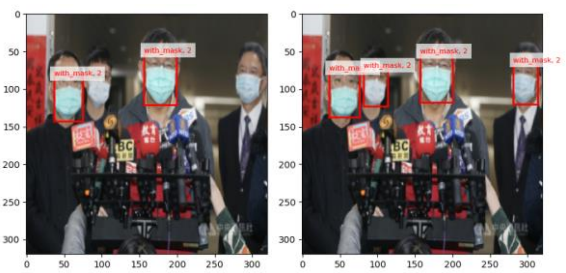
Selbst mit nur 4 Klassen dauerte das Training des SSDLite320-Modells 37 Minuten, während das SSD300-Modell 46 Minuten benötigte. Wie zu erwarten, war das SSD300-Modell auch deutlich besser in Bezug auf die Evaluierungsergebnisse. Es erreichte eine mAP_50_95 von 0.397 und eine mAP_50 von 0.607, während das SSDLite-Modell Werte von 0.26 und 0.415 aufwies.

SSD300_VGG16	SSDLite320_MobileNetV3
	
	
	
	

Vergleich zwischen ssd300 und ssdLite an XML Datensatz

Unter Verwendung des deutlich kleineren XML-Datensatzes wurden, wie erwartet, aufgrund der geringen Anzahl an Datensamples deutlich schlechtere Ergebnisse erzielt. Das SSDLite-Modell erreichte nach 12 Epochen lediglich eine mAP von 0.055, was vergleichsweise niedrig ist. Das SSD300-Modell hingegen erreichte mit dem kleinen Datensatz eine mAP von 0.267, was deutlich besser ist, jedoch immer noch nicht zufriedenstellend.

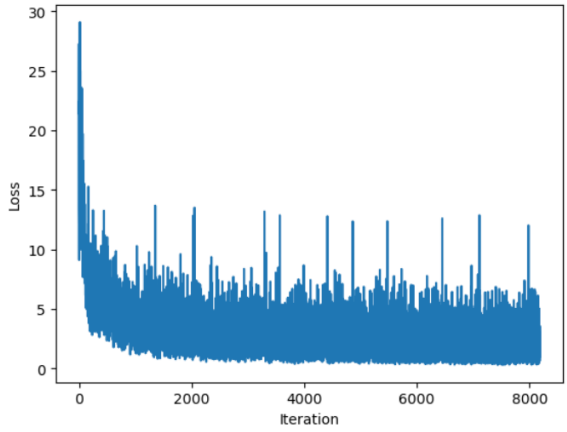
Die Laufzeit des Trainings unterscheidet sich kaum, wobei das SSDLite-Modell mit 7 Minuten nur 35 Sekunden schneller war als das SSD300-Modell. Dieser geringe Zeitunterschied kann jedoch die deutlich schlechtere Leistung des SSDLite-Netzwerks nicht rechtfertigen.

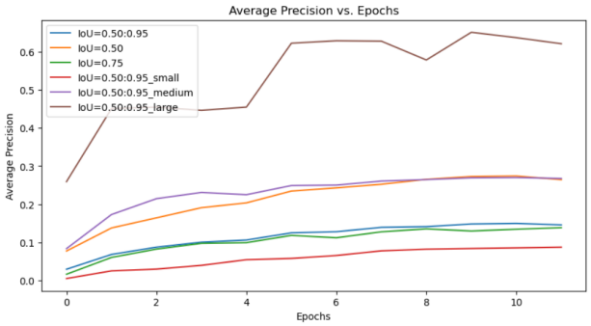
SSD300_VGG16	SSDLite320_MobileNetV3
	
	
	
	

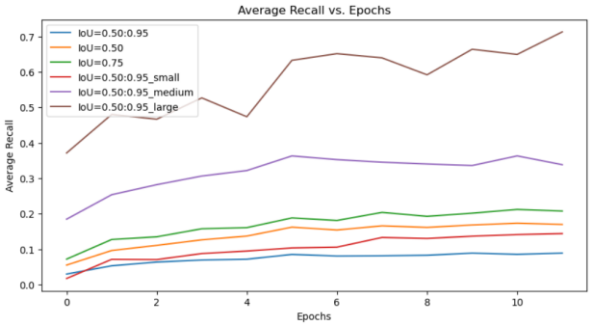
Vergleich zwischen ssd300 und ssdLite bei verdunkeltem XML Datensatz

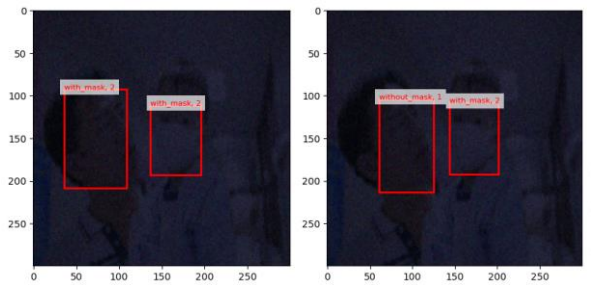
Bei der letzten Phase des Trainings wurden beide Modelle auf den datenargumentierten dunklen XML-Datensatz trainiert. Ähnlich wie bei den vorherigen Experimenten schnitt auch hier das SSD300-Modell besser ab als das SSDLite-Modell. Beim dunklen Datensatz erreichte das SSD300-Modell jedoch nur eine mAP von 0.146, während das SSDLite-Modell lediglich eine mAP von 0.041 erzielte. Das SSDLite-Modell schnitt somit nur etwas schlechter ab als beim hellen Datensatz, konnte aber dennoch gelegentlich Masken und Gesichter richtig erkennen. Es ist jedoch wichtig zu beachten, dass die Bilder im dunklen Datensatz teilweise sehr dunkel waren, wodurch es bereits für das menschliche Auge schwierig war, alles richtig zu erkennen.

SSD300_VGG16

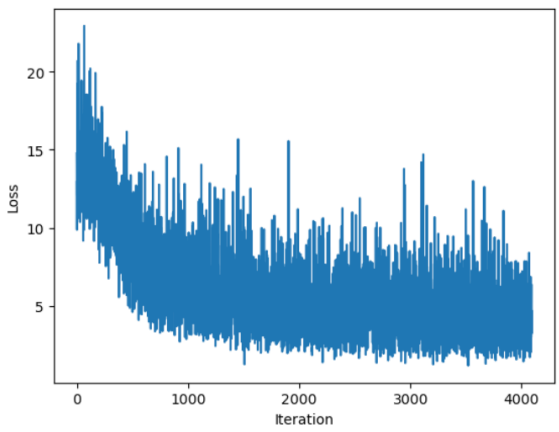


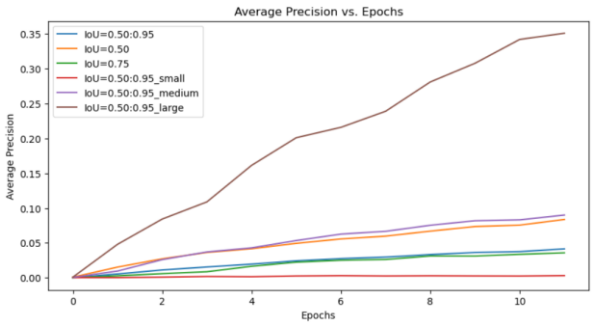


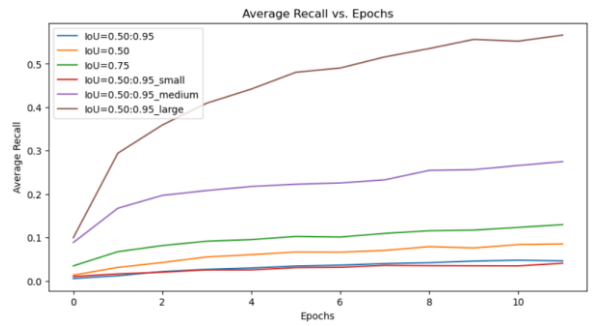




SSDLite320_MobileNetV3







Beste Ergebnisse beider Modelle

Das beste Ergebnis wurde mit dem SSD300_VGG16-Modell erzielt, das auf dem JSON-Datensatz trainiert wurde und nur 4 Klassen umfasste. Das SSDLite320_MobileNetV3-Modell erzielte ebenfalls das beste Ergebnis mit dem JSON-Datensatz und nur 4 Klassen. Allerdings war das Ergebnis des SSDLite320-Modells gerade einmal so gut wie das des SSD300_VGG16-Modells, das auf dem kleinen XML-Datensatz trainiert wurde.

Grundlegender Ergebnisvergleich beider Modelle

Grundsätzlich lässt sich sagen, dass das SSD300_VGG16-Modell deutlich bessere Leistung erbracht hat als das SSDLite320_MobileNetV3-Modell. Unabhängig von den Umständen waren der Loss, die mAP und der durchschnittliche Recall immer besser beim SSD300-Modell im Vergleich zu SSDLite. Leider konnte auch die Performance von SSDLite nicht wirklich überzeugen. Zwar war es etwas schneller, aber nur marginal. SSDLite gibt leider zu viel an Genauigkeit auf, um nur geringfügig besser in der Performance zu sein. Das SSD300-Modell hingegen hat insgesamt gut abgeschnitten.

Allerdings waren auch die typischen Schwächen von SSD erkennbar, wie das Erkennen kleiner Objekte oder Gesichter sowie das Überlappen von Bounding Boxes bei eng beieinander liegenden Gesichtern. Durch längeres Training, also mehr Epochen, könnten jedoch Verbesserungen in der Leistung des Modells erzielt werden. Ebenso könnte die Anwendung von Data Augmentation helfen, das Modell robuster und vielseitiger zu machen.

Vergleich der Ergebnisse

JSON Datensatz alle Klassen

Als erstes wird die Performance der Modelle verglichen, indem auf Datensatz 1 trainiert wird unter Berücksichtigung aller Klassen. Wie bereits oben beschrieben, wurden pro Architektenfamilie immer zwei Modelle erstellt. Für die Vergleiche unter den Modellen wird jeweils das bessere Modell der beiden herangezogen.

	Precision 0.5:0.95	Precision 0.5	Recall 0.5:0.95	Precision	Recall	Epochen	BatchSize
YOLOv5	0.287	0.460		0.681	0.455	20	5
Faster RCNN Resnet-50	0.2930	0.525	0.469			10	12
SSD300_VGG16	0.187	0.339	0.355			12	1

Die in der Tabelle präsentierten Ergebnisse ermöglichen einen Vergleich der Leistung verschiedener Modelle für die Objekterkennung. Dabei können folgende Schlussfolgerungen gezogen werden:

Der Precision-Wert in YOLOv5 deutet darauf hin, dass das Modell in der Lage ist, Objekte präzise zu erkennen und ihre Position genau zu lokalisieren.

Das Faster RCNN-Modell mit dem Resnet-50-Backbone erreicht eine vergleichbare Precision bei einem mAP von 0,5 bis 0,95. Dies deutet darauf hin, dass es ähnlich wie YOLOv5 eine gute Fähigkeit zur präzisen Objekterkennung aufweist. Der Recall-Wert zeigt, dass das Modell in der Lage ist, einen Großteil der relevanten Objekte zu erkennen.

Im Gegensatz dazu erzielt das SSD300_VGG16-Modell eine niedrigere Precision. Dies deutet darauf hin, dass es Schwierigkeiten hat, Objekte präzise zu identifizieren. Der Recall-Wert zeigt, dass das Modell nur eine moderate Fähigkeit besitzt, relevante Objekte zu erkennen.

Zusammenfassend lässt sich sagen, dass YOLOv5 und Faster R-CNN im Allgemeinen bessere Ergebnisse in Bezug auf die mAP erzielen als SSD.

JSON Datensatz nur Gesichtsklassen

Im folgenden Fall werden die Modelle erneut auf Datensatz 1 verglichen, jedoch mit dem Fokus auf Gesichtsmasken-Erkennung. Das bedeutet, dass die Modelle ausschließlich auf diejenigen Klassen trainiert werden, die Gesichter und damit relevante Informationen für die Gesichtsmasken-Erkennung enthalten. Der Zweck dieser Untersuchung besteht darin, festzustellen, ob die Hinzufügung weiterer Klassen die Genauigkeit der Modelle beeinflusst.

	Precision 0.5:0.95	Precision 0.5	Recall 0.5:0.95	Precision	Recall	Epochen	BatchSize
YOLOv5	0.529	0.733		0.874	0.643	20	5
Faster RCNN Resnet-50	0.485	0.705	0.645			10	12
SSD300_VGG16	0.405	0.631	0.591			12	1

Die neuen Werte in der Tabelle zeigen Verbesserungen in der Präzision und dem Recall für alle Modelle im Vergleich zur vorherigen Tabelle.
YOLOv5 erzielt eine höhere Präzision zuvor. Der Recall steigt ebenfalls deutlich. Ähnliche Verbesserungen sind auch beim Faster RCNN-Modell mit Resnet-50-Backbone zu beobachten.
Das SSD300_VGG16-Modell zeigt ebenfalls eine bessere Präzision im Vergleich zu den vorherigen Werten, schneidet jedoch wieder am schlechtesten ab.
Insbesondere YOLOv5 zeigt signifikante Verbesserungen in der Präzision und dem Recall und erzielt insgesamt die besten Ergebnisse.
Letztlich lässt sich daraus schließen, dass eine allgemeine verbesserung erzielt werden kann, wenn die Modelle nur auf Gesichtsmasken relevante Klassen trainiert werden.

XML Datensatz

Im nächsten Beispiel werden die Modelle auf den zweiten deutlich kleineren Datensatz trainiert.

	Precision 0.5:0.95	Precision 0.5	Recall 0.5:0.95	Epochen	BatchSize
Faster RCNN Resnet-50	0.417	0.709	0.54	10	12
SSD300_VGG16	0.267	0.485	0.386	12	1

Das Faster RCNN-Modell mit Resnet-50-Backbone zeigt eine höhere Precision bei mAP von 0,5 bis 0,95 im Vergleich zum SSD300_VGG16-Modell. Das deutet darauf hin, dass das Faster RCNN-Modell eine bessere Fähigkeit besitzt, Objekte präzise zu erkennen und zu lokalisieren.
Gleiches lässt sich bei einem mAP von 0,5, sagen. Dies zeigt, dass das Faster RCNN-Modell eine höhere Genauigkeit bei der Klassifizierung von Objekten aufweist.

In Bezug auf den Recall erreicht das Faster RCNN-Modell ebenfalls einen höheren Wert. Auch dies deutet darauf hin, dass das Faster RCNN-Modell eine bessere Fähigkeit hat, eine größere Anzahl relevanter Objekte zu erkennen und zu erfassen.
Zusammenfassend lässt sich sagen, dass das Faster RCNN-Modell mit Resnet-50-Backbone bessere Leistungswerte in Bezug auf Präzision und Recall aufweist als das SSD300_VGG16-Modell. Das deutet darauf hin, dass das Faster RCNN Modell in diesem Anwendungsfall bessere und zuverlässige Ergebnisse liefert.

Datensatz XML dunkel

In diesem Beispiel werden die Modelle auf Datensatz 2 trainiert, allerdings auf die verdunkelte Variante, um low-light Performance zu testen.

	Precision 0.5:0.95	Precision 0.5	Recall 0.5:0.95	Precision	Recall	Epochen	BatchSize
YOLOv5	0.352	0.595		0.665	0.588	50	5
Faster RCNN Resnet-50	0.238	0.439	0,428			10	12
SSD300_VGG16	0.146	0.264	0.208			12	1

Im dunklen Datensatz zeigt das Faster RCNN-Modell eine Verschlechterung der Präzision und des Recalls im Vergleich zum hellen Datensatz. Dies deutet darauf hin, dass das Modell Schwierigkeiten hat, Objekte unter schlechten Lichtverhältnissen genau zu erkennen und zu klassifizieren.
Ähnlich wie das Faster RCNN-Modell zeigt auch das SSD300_VGG16-Modell eine Verschlechterung der Präzision und des Recalls im dunklen Datensatz. Dies weist darauf hin, dass das Modell Schwierigkeiten hat, Objekte in schlecht beleuchteten Szenarien präzise zu identifizieren.
Jedoch zeigt das YOLO Modell deutlich bessere Güte als die beiden anderen Ansätze. Dies könnte aber an der deutlich längeren Trainingszeit liegen.
Eine weiter Logische Schlussfolgerung ist, dass SSD und Faster RCNN auf den verdunkelten Datensatz wegen der schlechteren Verhältnisse auch deutliche einbußen in der Pefromance hinnehmen im vergleich zum gleichen Datensatz mit hellen Daten

Datensatz 1 einzelne Gesichter

Außerdem wurde untersucht, wie gut Modelle abschneiden, wenn nur auf einzelne Gesichter trainiert wird. Es sollte herausgefunden werden, ob Bilder mit mehreren Menschen grundsätzlich eine Hürde für die Modelle darstellen. Falls das der Fall ist, sind hier bessere Ergebnisse zu erwarten.

	Precision 0.5:0.95	Precision 0.5	Recall 0.5:0.95	Precision	Recall	Epochen	BatchSize
YOLOv5	0.210	0.29		0.915	0.682	35	5
Faster RCNN MobileNetV3	0.467	0.66	0.628			10	12

Es lässt sich sagen, dass sowohl das YOLOv5-Modell als auch das Faster RCNN-Modell eine gewisse Fähigkeit zur Erkennung einzelner Gesichter aufweisen. Das YOLOv5-Modell erzielt jedoch insgesamt eine höhere Präzision und einen höheren Recall im Vergleich zum Faster RCNN-Modell. Im vergleich zum training auf alle Gesichtsklassen, sieht man bei Faster RCNN kaum veränderung und bei YOLO eine leichte Verbesserung.

Vergleich der besten Modelle mit MobileNet

Abschließend wird noch verglichen, für welche Mobile Anwendung sich welches Modell am besten eignet, da sowohl die Faster RCNN als auch die SSD Modelle jeweils einmal mit Mobilenet getestet wurden. Der hier dargestellte vergleich bezieht sich auf das Training auf den ersten Datensatz und nur auf Klassen, die Gesichter enthalten.

	Precision 0.5:0.95	Precision 0.5	Recall 0.5:0.95	Epochen	BatchSize
Faster RCNN MobileNetV3	0.456	0.696	0.600	10	12
SSDLite320 MobileNetV3	0.260	0.415	0.467	12	2

Basierend auf diesen Ergebnissen lässt sich feststellen, dass das Faster RCNN-Modell mit MobileNetV3-Backbone eine bessere Leistung bei der Einzelgesichtserkennung aufweist als das SSDLite320-Modell mit MobileNetV3-Backbone. Das Faster RCNN-Modell erzielt insgesamt höhere Präzisions- und Recall-Werte, was auf eine effektivere Erkennung einzelner Gesichter hinweist.

Abbildungsverzeichnis

Literaturverzeichnis

[1] W. K. Yiu, „https://github.com/WongKinYiu/PyTorch_YOLOv4,“ [Online].

[2] U. Glenn Jocher, *YOLOv5 Github.*

[3] X. Z. S. R. a. J. S. Kaiming He, *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition.*

[4] S. Ren, K. He, R. Girshick, J. Sun und S. Ren, *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.*

[5] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen und M. Tan, *Searching for MobileNetV3.*