

Bachelorarbeit

Felix Stöbel

Neuronale Indoor-Objekterkennung für mobile Roboter

Felix Stöbel

Neuronale Indoor-Objekterkennung für mobile Roboter

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter: Prof. Dr. Tim Tiedemann

Eingereicht am: 04. September 2021

Felix Stößel

Thema der Arbeit

Neuronale Indoor-Objekterkennung für mobile Roboter

Stichworte

Neuronale Faltungsnetzwerke, Objekterkennung, NVIDIA Jetson Nano, YOLO, COCO, Open Images, Darknet, Training von neuronalen Netzen

Kurzzusammenfassung

Die Objekterkennung ermöglicht Anwendungen im Bereich der Kollisionserkennung bei autonomen Fahrzeugen und Robotern. Diese Arbeit zeigt das Training eines neuronalen Faltungsnetzwerkes auf Daten des Open-Images- und COCO-Datensatzes. Das in dieser Arbeit trainierte YOLO-Netzwerk wird auf einem NVIDIA Jetson Nano im Darknet Framework getestet.

Felix Stößel

Title of Thesis

Neural indoor object detection for mobile robots

Keywords

Convolutional Neural Networks, object detection, NVIDIA Jetson Nano, YOLO, COCO, Open Images, Darknet, training of neural networks

Abstract

Object detection enables applications in the field of collision detection for autonomous vehicles and robots. This work shows the training of a convolutional neural network on data of the Open-Images- and COCO-dataset. The YOLO-network trained in this work will be tested in the Darknet Framework on the NVIDIA Jetson Nano.

Inhaltsverzeichnis

Abbildungsverzeichnis	vi
Tabellenverzeichnis	viii
1 Einleitung	1
1.1 Zielsetzung	2
1.2 Aufbau der Arbeit	2
2 Grundlagen	4
2.1 Convolutional Neural Network - Allgemein	4
2.1.1 Layertypen	6
2.1.2 Training von neuronalen Netzen	10
3 Stand der Technik	15
3.1 Entwicklung von Wettbewerbs-Datensätzen	15
3.1.1 VOC - Pascal Visual Object Classes Challenge (2005-2012)	15
3.1.2 ILSVRC - The ImageNet Large Scale Visual Recognition Challenge (2010-2017)	16
3.1.3 MS-COCO - Common Objects in Context Object Detection Chal- lenge (2015 - Heute)	19
3.1.4 OID - Open Images Detection Challenge (2018 - Heute)	20
3.2 Entwicklung von Objekterkennung/-lokalisierung	22
3.2.1 YOLOv1 (2015)	25
3.2.2 YOLOv2 (2016)	28
3.2.3 YOLO9000	29
3.2.4 YOLOv3 (2018)	30
3.2.5 YOLOv4 (2020)	32
4 Training des Netzes	39
4.1 Auswahl und Aufbereitung des Datensatzes	39

4.2	Vorbereitung von Darknet	41
4.3	Erster Trainingsdurchlauf	44
4.3.1	Auswertung des Trainings	44
4.4	Fehlerursache	47
4.5	Training mit verbesserten Daten	49
5	Objekterkennung auf dem Jetson Nano	52
5.1	Vorbereitung des Gerätes	52
5.2	Performanceanalyse	53
5.3	Ausblick	53
6	Auswertung im Bezug auf die Aufgabenstellung	56
6.1	Diskussion der Ergebnisse	56
6.2	Vergleich mit anderen Lösungen	57
6.3	Future Work	57
	Literaturverzeichnis	58
A	Anhang	65
A.0.1	Aufsetzen einer Maschine für YOLO	65
A.0.2	OIDv4 ToolKit	69
A.0.3	coco manager	70
	Selbstständigkeitserklärung	77

Abbildungsverzeichnis

2.1	v.l.n.r. Transformation, Skalierung, Rotation, Strichstärke	4
2.2	Übereinstimmung des Buchstaben X	5
2.3	Untersuchung von Merkmalen des Buchstaben	5
2.4	Features	5
2.5	Convolution und Feature Map	6
2.6	Beispielaufbau eines CNN	6
2.7	Feature-Maps	7
2.8	Normalisierung der Feature-Maps	7
2.9	v.l.n.r. ReLU, Sigmoid, Tanh	8
2.10	v.l.n.r. ELU, Leaky ReLu	8
2.11	Max-Pooling auf Feature-Maps angewandt	8
2.12	Fully-Connected-Layer	9
2.13	Gradientenabstieg zum Minimum der Fehlerfunktion	11
2.14	Gradientenabstieg zu einem lokalen Minimum der Fehlerfunktion	12
2.15	Beispiel der Delta-Lernregel nach 300 Iterationen	13
3.1	Beispielbilder des VOC2007 Datensatzes	16
3.2	Beispielbilder der ImageNet Datenbank	18
3.3	Beispielbilder des COCO Datensatzes	19
3.4	Beispielbilder des Open Images Datensatzes	20
3.5	Übersicht von Objekterkennungsmethoden [54]	22
3.6	Beispiel für die Körpererkennung mithilfe von DPM [13] v.l.n.r Ganzkörperfilter(root), Körperfilter in fünf Teilen(part), Kostenfilter, Ergebnis des Detektors	23
3.7	Pipeline von R-CNN [15]	24
3.8	Bounding Boxes mithilfe von Selective Search	24
3.9	Pipeline von YOLO [34]	25
3.10	Architektur von YOLO [34]	26

3.11	Übersicht Objekterkennungssysteme [34]	27
3.12	Übergang von YOLO zu YOLOv2 [35]	29
3.13	Darknet-19 [35]	30
3.14	Vergleich von YOLOv3 mit anderen Objektdetektoren [36]	31
3.15	Architektur von Darknet-53 [36]	32
3.16	Aufbau von One- und Two-Stage Detektoren [2]	33
3.17	Cross-Stage-Partial-Connections [46]	33
3.18	Verschiedene Skalierungen von neuronalen Netzen [44]	34
3.19	Vergleich von ResNext50, Darknet53 und EfficientNet-B3	34
3.20	Aufbau eines Path Aggregation Network (PAN)[28]	35
3.21	Ablationsstudie der Bag of Freebies Methoden	37
3.22	Ablationsstudie der Bag of Specials Methoden	37
3.23	Vergleich von verschiedenen Objektdetektoren	38
4.1	Ordnerstruktur von Darknet	43
4.2	Loss- und mAP-Graph des Trainings mit vier Klassen	45
4.3	Loss- und mAP-Graph des Trainings mit einer Klasse	46
4.4	mAP-Berechnung des Trainings mit wenig Trainingsbildern	46
4.5	mAP-Berechnung des Trainings mit vielen Trainingsbildern	47
4.6	Beispiel für das Erkennen von false-negatives	47
4.7	Beispiel für das Erkennen von false-positives	48
4.8	Erkennungsbeispiel	48
4.9	mAP nach dem Aufräumen der Annotations	49
4.10	Training von YOLOv3 auf 500 Bildern	49
4.11	Training von YOLOv4 auf 500 Bildern	50
4.12	Training von YOLOv4-tiny auf 500 Bildern	51
5.1	Aufbau des NVIDIA Jetson Nano	52
A.1	GCC Version	65
A.2	Download von CUDA	66
A.3	CUDA Installation. Kein Driver, keine Samples.	66

Tabellenverzeichnis

3.1	Bildverteilung der VOC Datensätze	15
3.2	Vergleich von Eigenschaften des VOC-2012 und ILSVRC-2013 Datensatzes	18
3.3	Bildverteilung der ImageNet Datensätze für Objekterkennung	18
3.4	Bildverteilung der COCO Datensätze	20
3.5	Bildverteilung der Open Images Datensätze	21
3.6	Gesamtübersicht der Bildverteilung der erwähnten Datensätze	21
5.1	Performance der YOLO-Netze mit Netzinput von 416×416 Pixel	53
5.2	Fortschritt der Jetson-Reihe. *Innerhalb der Serie Abweichungen. (Quelle: https://developer.nvidia.com/embedded/jetson-modules)	54
5.3	Ausschnitt von Jetson Benchmarks mit Deepstream (Quelle: https://docs.nvidia.com/metropolis/de)	
5.4	Ausschnitt von Jetson Benchmarks mit TensorRT (Quelle: https://docs.nvidia.com/deeplearning/te)	

1 Einleitung

Die Objekterkennung findet heutzutage in vielen Bereichen Anwendung. Von Verkehrs- und Nummernschilderkennung für autonom fahrende Fahrzeuge[53][33] bis zu automatischer Virenerkennung über Röntgenbilder[32] wird die Objekterkennung eingesetzt. Auch an unterstützenden Anwendungen, zum Beispiel zur Fortbewegung durch unbekannte Orte oder beim Finden der richtigen Medikamente für Blinde oder Sehbeeinträchtigte, wird geforscht [48][7]. Oftmals wird eine Objekterkennung auf Bildern durchgeführt. Andere Ansätze basieren zum Beispiel auf Punktwolkedaten aus einem Light-Detection-And-Ranging-Sensor [3]. Für diese Arbeit ist besonders der Ansatz basierend auf Bildern, beziehungsweise Videos, von Bedeutung.

Früher wurde eine Objekterkennung mittels klassischer Bilderkennungsverfahren durchgeführt [19][20][50]. Heutzutage werden dafür vermehrt neuronale Netze (*engl. Neural Networks*) benutzt [8].

Die Verarbeitung der Informationen findet dabei direkt im Fahrzeug, in Robotern oder kleineren Endgeräten statt. Letzteres ermöglicht die Erweiterung von alten Technologien, um zum Beispiel nachträglich eine Objekterkennung zu ermöglichen. Mit zunehmendem technologischen Fortschritt wird es immer erschwinglicher teure Rechenoperationen nicht nur auf Hochleistungsrechnern durchzuführen. Schon im Jahr 2019 veröffentlichte die NVIDIA Corporation ein Endgerät, den NVIDIA Jetson Nano. Dieser bietet eine vergleichsweise leistungsstarke Plattform, besonders für die Arbeit mit neuronalen Netzen. Diese Plattform wurde bereits für mobile Roboter benutzt [1].

Auch in dieser Arbeit soll der NVIDIA Jetson Nano als Plattform dienen, um in Zukunft ein trainiertes neuronales Netz auf einem mobilen Roboter zu nutzen.

1.1 Zielsetzung

Ziel dieser Arbeit ist es, eine Objekterkennung zu realisieren. Diese Objekterkennung soll auf Videomaterial einer Kamera arbeiten. Als grundlegender Ansatz soll für diese Arbeit die Objekterkennung von You-Only-Look-Once (YOLO) durchgeführt werden. Die Datenauswahl und das Training des YOLO-Netzes sollen dabei eine besondere Rolle spielen. Diese Objekterkennung soll Einsatz in einer Innenraumumgebung finden, weshalb das Netz besonders Fokus auf Gegenständen dieser Umgebungen liegen soll.

Ein Augenmerk dieser Arbeit soll auf der Dokumentation und Reproduzierbarkeit der Ergebnisse liegen.

Als Plattform für die Objekterkennung dieser Arbeit, basierend auf neuronalen Netzen, wurde der NVIDIA Jetson Nano gewählt. Der Jetson Nano ist mit seiner Größe von $7\text{cm} \times 4,5\text{cm}$ geeignet als Erweiterung für diverse mobile Roboter, wie zum Beispiel den Segway Loomo Robot. Aus diesem Grund wurde diese Plattform für diese Arbeit gewählt.

Die in dieser Arbeit erlangten Kenntnisse sollen eine Grundlage schaffen, um eine eventuelle Anwendung im Bereich der selbstständigen Fortbewegung, Orientierung und Kollisionsvermeidung eines Segway Loomo Robots innerhalb der Räumlichkeiten der Hochschule für Angewandte Wissenschaften Hamburg zu ermöglichen.

1.2 Aufbau der Arbeit

Diese Arbeit ist, inklusive Einleitung, in sechs Kapitel unterteilt.

In Kapitel 2 werden die allgemeinen Grundlagen von Convolutional Neural Networks (CNN), häufig benutzte Arten von Schichten (engl. Layer) und Trainingsmethoden erklärt.

Das darauffolgende Kapitel 3 beschreibt den Stand der Technik von bekannten Datensätzen und Wettbewerbe, sowie die Funktionsweise des YOLO-Algorithmus, seine Entwicklung und Unterschiede zu anderen bekannten Methoden der Objekterkennung. Insbesondere wird auf die Wettbewerbsdatsätze der Pascal-Visual-Object-Classes-Challenge

(VOC), der ImageNet-Large-Scale-Visual-Recognition-Challenge (ILSVRC), der Common-Objects-in-Context-Object-Detection-Challenge (COCO) und die Open-Images-Detection-Challenge (OID) eingegangen.

Kapitel 4 zeigt den Trainingsablauf von YOLO von der Datenauswahl bis zum Validieren des Netzes.

Kapitel 5 behandelt die Vorbereitung des Jetson Nanos sowie den Betrieb mit YOLO.

Im letzten Kapitel 6 werden abschließend die Ergebnisse ausgewertet und bewertet sowie zukünftige Anwendungen angesprochen.

2 Grundlagen

2.1 Convolutional Neural Network - Allgemein

Ein neuronales Netz, oder auch ein künstliches neuronales Netz, ist ein Algorithmus, der dem menschlichen Gehirn nachempfunden ist. Künstliche Neuronen werden verknüpft, um so diverse Aufgaben zu bewältigen. Die Aufgabe eines neuronalen Netzes ist meistens die Extraktion von Informationen aus zum Beispiel: Texten, Tabellen oder wie im Fall dieser Arbeit: Bildern. Wenn Daten in einer bildähnlichen Form dargestellt werden können, dann kann ein neuronales Netz eventuell Muster erkennen, wenn bestimmte Bedingungen erfüllt sind. Ein neuronales Netz wird auf bestimmte Fälle zugeschnitten, indem es mit passenden Daten trainiert wird. Auf diese Weise kann zum Beispiel einer künstlichen Intelligenz (KI) beigebracht werden Videospiele zu spielen [31]. Eine häufig gewählte Anwendung für neuronalen Netzen ist die Objekterkennung.

Ein Beispiel wäre die Erkennung von handgeschriebenen Buchstaben zum Digitalisieren von handgeschriebenen Texten. Der gleiche Buchstabe kann jedoch mit Abweichungen in Form, Größe, Rotation oder Strichstärke vorkommen, weswegen ein neuronales Netz das gegebene Bild nicht mit einem makellosen Gegenstück vergleichen kann (Siehe Abbildung 2.1). Am Beispiel des Buchstaben X wird im Folgenden der Vorgang innerhalb eines neuronalen Netzes erläutert, speziell wird hier ein CNN benutzt.

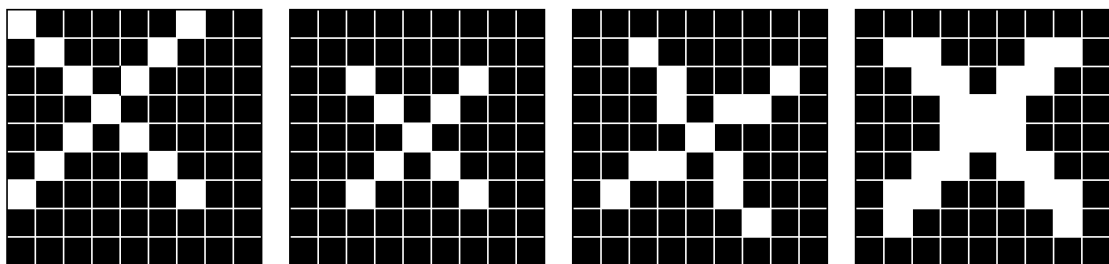


Abbildung 2.1: v.l.n.r. Transformation, Skalierung, Rotation, Strichstärke

Für ein CNN sind die Pixel wie in Abbildung 2.1 nicht Schwarz und Weiß, sondern lediglich eine Zahl, welche im Fall der Bilderkennung einen Farb- oder Grauwert darstellt. Würde ein CNN nun Zahlen eines verzerrten Buchstaben mit der sauberen Darstellung vergleichen, dann würde es keine totale Übereinstimmung feststellen (Siehe Abbildung 2.2).

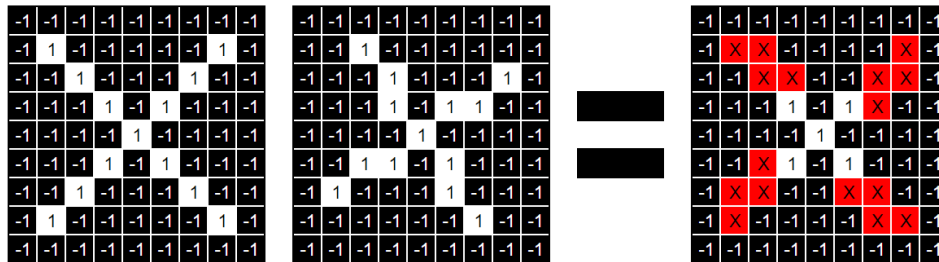


Abbildung 2.2: Übereinstimmung des Buchstaben X

Aus diesem Grund werden in CNNs nur Teilbereiche des Bildes verglichen und auf Merkmale untersucht (Siehe Abbildung 2.3). Diese Teilbereiche werden Features genannt (Siehe Abbildung 2.4). In diesem Beispiel haben die Features eine Größe von 3×3 Pixel.

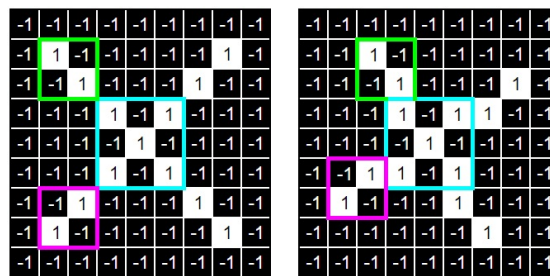


Abbildung 2.3: Untersuchung von Merkmalen des Buchstaben



Abbildung 2.4: Features

Die ausgewählten Features für den Buchstaben werden nun als Filter auf das Bild angewandt. Jeder Wert aus dem Feature-Bild wird mit dem in Relation stehenden Wert im Input-Bild multipliziert. Die neun Werte werden addiert und durch die Anzahl der Pixel im Feature-Bild geteilt. Die Anwendung des Filters auf jede Position im Bild wird Convolution (dt.: Faltung) genannt und ist namensgebend für Convolutional Neural Networks. Das so erzeugte, neue Bild wird Feature-Map genannt (Siehe Abbildung 2.5).

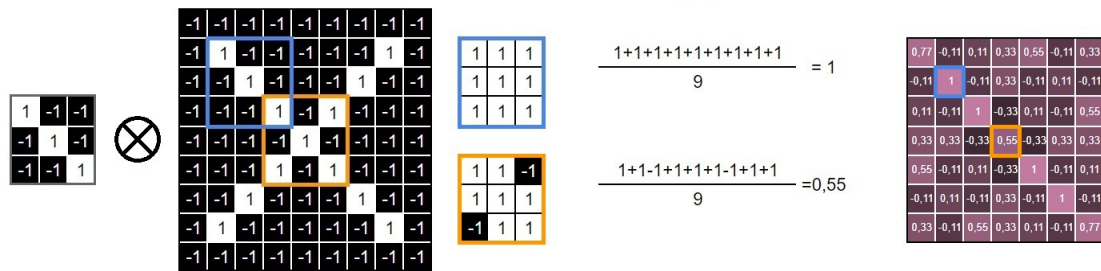


Abbildung 2.5: Convolution und Feature Map

2.1.1 Layertypen

In einem CNN gibt es verschiedene Layer (dt.: Schichten), welche unterschiedliche Aufgaben durchführen. Dabei ist der Output (dt.: Ausgabe) des ersten Layers der Input (dt.: Eingabe) des darauffolgenden Layers. Oftmals werden wiederkehrend, gleiche Kombinationen von Layer-Arten innerhalb eines CNN benutzt (Siehe Abbildung 2.6). Je mehr Layer ein CNN besitzt, desto tiefer ist das CNN. Im Allgemeinen werden der erste und letzte Layer eines CNN, Input- und Output-Layer genannt. Alle Layer dazwischen werden Hidden-Layer genannt, da der Input und Output durch eventuell variierende Layer-Kombinationen, Aktivierungsfunktionen oder gewichtete Inputs nicht immer klar ist. In bekannten CNN-Architekturen, wie beispielsweise AlexNet [23], VGGNet [41], GoogLeNet [42] und ResNet [18], werden vermehrt die im nächsten Abschnitt beschriebenen Layer-Typen verwendet.

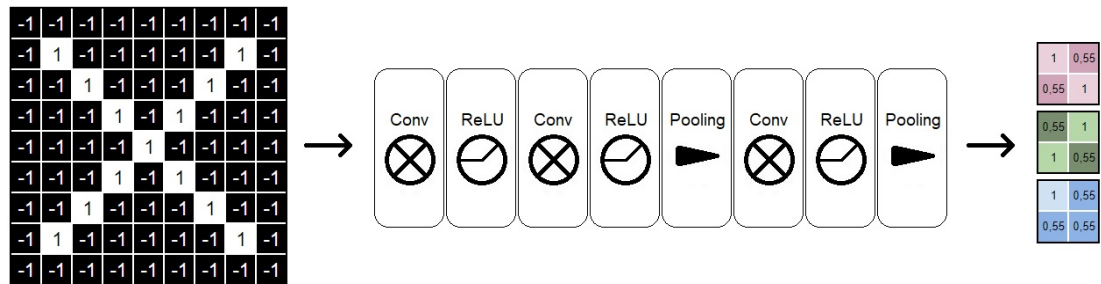


Abbildung 2.6: Beispielaufbau eines CNN

Convolutional-Layer

Als Convolutional-Layer bezeichnet man ein Layer, in welchem alle Feature-Filter angewandt und alle Feature-Maps erstellt werden. Die hier produzierten Feature-Maps (Siehe

Abbildung 2.7) enthalten oftmals negative Werte, weswegen meistens ein Normalization-Layer auf ein Convolutional-Layer folgt.

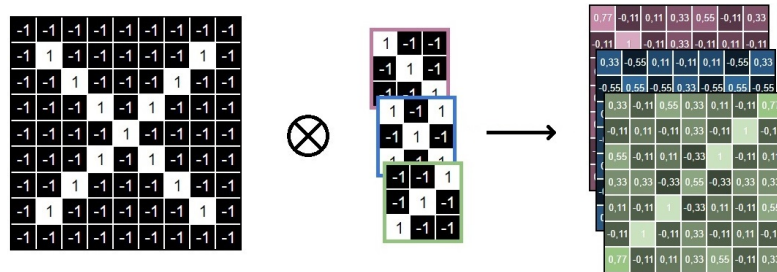


Abbildung 2.7: Feature-Maps

Normalization-Layer

Mithilfe einer Normalisierung des Outputs wird unnötiger Rechenaufwand verhindert, indem Zahlen auf Werte innerhalb eines gewünschten Bereichs gesetzt werden (Siehe Abbildung 2.8). In manchen Fällen kann dies nicht erwünscht sein. Bei diesem Vorgang verändert sich die Bildgröße nicht.

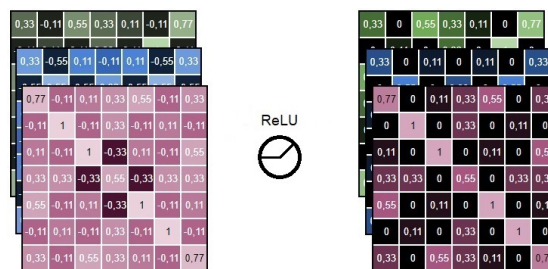


Abbildung 2.8: Normalisierung der Feature-Maps

Oft benutzte Arten der Normalisierung, auch Aktivierungsfunktion genannt, sind unter anderem die Rectified Linear Unit (ReLU), die Logistische-Funktion (Sigmoid) und die Hyperbolic Tangent-Funktion (Tanh) [40]. Jede Aktivierungsfunktion erzeugt einen anderen Output. So liegt bei ReLU der Output zwischen 0 und ∞ , bei Sigmoid zwischen 0 und 1 und bei Tanh zwischen -1 und 1 (Siehe Abbildung 2.9).

Jede dieser Aktivierungsfunktionen hat ihre Vor- und Nachteile, so kann es bei einer ReLU Aktivierungsfunktion vorkommen, dass Neuronen die auf 0 gesetzt werden nie wieder einen Wert höher als 0 erreichen. Dieses Problem wird Dying-ReLU genannt[27]. Um

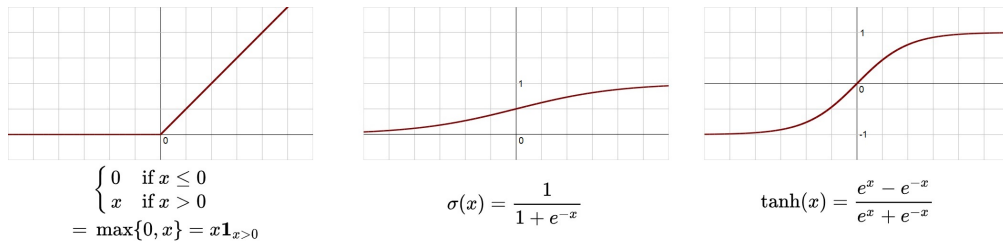


Abbildung 2.9: v.l.n.r. ReLU, Sigmoid, Tanh

dieses Problem zu umgehen, gibt es Varianten von ReLU's bei denen negative Werte zu einem gewissen Grad zugelassen werden. Einige dieser ReLU's sind zum Beispiel Leaky-ReLU oder Exponential-Linear-Unit (ELU) (Siehe Abbildung 2.10)

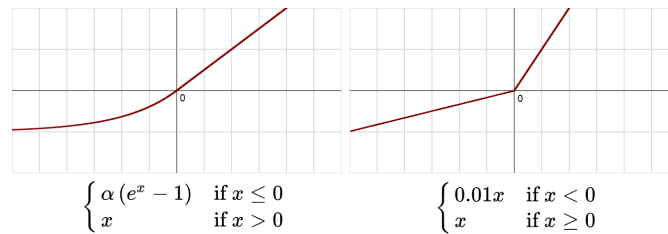


Abbildung 2.10: v.l.n.r. ELU, Leaky ReLU

Pooling-Layer

Große Input-Bilder sind in der Objekterkennung aufwändig zu verarbeiten, weswegen der Input verkleinert werden soll. Dabei gehen Details verloren, aber man gewinnt Geschwindigkeit. Der Vorgang des verkleinern des Bildstapels nennt sich Pooling.

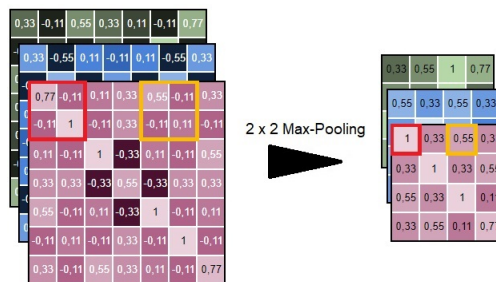


Abbildung 2.11: Max-Pooling auf Feature-Maps angewandt

Bei diesem Vorgang wird ein Window (dt.: Fenster) von beispielsweise 2×2 oder 3×3 Pixel des Input-Bildes angeschaut und ein Wert für das Ergebnisbild bestimmt. Beim sogenannten Max-Pooling wird immer der höchste Wert des Bereichs gewählt. Mithilfe von verschiedenen Step-Größen (dt.: Schrittweiten) wird dafür gesorgt, dass Pixel nicht mehrmals angeschaut werden. Auffällige Merkmale gehen bei diesem Vorgang nicht verloren, so kann man zum Beispiel immer noch die Diagonale in der Feature-Map nach dem Pooling erkennen (Siehe Abbildung 2.11).

Fully-Connected-Layer

Am Ende des Netzes steht ein Fully-Connected-Layer. Der Input dieses Layers ist das Ergebnis aller verarbeiteten Feature-Maps und bildet einen bewertenden Zustand ab. Eine gewisse Kombination aus Werten wird vom CNN einem bestimmten Buchstaben zugewiesen. Der Mittelwert aller Werte, welche einen Einfluss auf die Erkennung haben, bildet die Genauigkeit, mit welcher der Buchstabe erkannt wurde. Damit das CNN lernt, welche Kombination von Werten zu welchem Buchstabe gehört, muss das CNN trainiert werden und den gesuchten Buchstaben im Training kennenlernen.

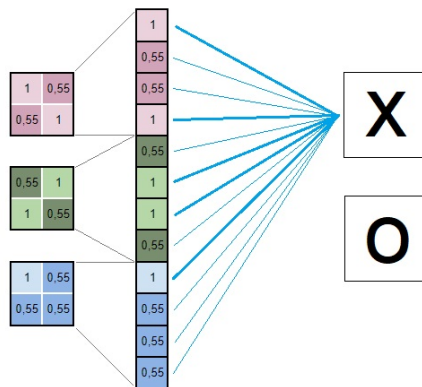


Abbildung 2.12: Fully-Connected-Layer

2.1.2 Training von neuronalen Netzen

Es gibt mehrere Arten von Training für CNNs. Bekannte Beispiele dafür sind: Supervised-Learning, Unsupervised-Learning, Self-Supervised-Learning oder Reinforcement-Learning [4].

Supervised-Learning ist die am weit verbreitetste Trainingsart [4]. Hierbei werden zuvor annotierte Daten als Trainingsdaten benutzt und das CNN lernt mithilfe dieser Daten bestimmte Objekte zu erkennen und zu klassifizieren, indem es Voraussagen trifft, welche mit den annotierten Daten verglichen werden und ein Fehler daraus errechnet wird. Anwendungsbeispiele für diese Art des Trainings sind zum Beispiel: Objekterkennung, Bild Segmentierung, Spracherkennung und Übersetzer.

Unsupervised-Learning benutzt keine annotierten Daten für das Training. Das Netz lernt ähnliche Muster zu gruppieren, ohne vor-trainierte Muster zu kennen. Dies ist zum Beispiel nützlich in der Datenanalyse, um Daten zu Clustern (dt.: gruppieren) und die Nutzbarkeit der Daten zu erhöhen.

Self-Supervised-Learning ist eine andere Art des Unsupervised-Learnings. Einem CNN wird eine einfach zu lösende Aufgabe gegeben, um die Gewichte des CNN grob an das gegebene Problem anzupassen. Das Netz wird später für ähnliche, aber kompliziertere Aufgaben benutzt.

Reinforcement-Learning ist eine Methode, bei welcher das CNN eingesetzt wird, um bestimmte Vorgänge durchzuführen, welche am Ende eine Belohnung liefern. Mittels Try-and-Error kann ein CNN lernen, Videospiele zu spielen [31] und immer höhere Punktzahlen erreichen.

Das Lernen des Netzes

Damit das CNN lernt, also der Fehler in den Vorhersagen minimiert wird, müssen die Gewichte der Neuronen angepasst werden. Dies geschieht nach jedem Durchlauf des CNNs. Hier wird mithilfe einer Loss-Funktion (dt.: Verlust-Funktion) ein Wert bestimmt, welcher den Fehler des CNNs hinsichtlich der Trainingsdaten repräsentiert. Dieser Wert wird während des Trainings mehrfach berechnet und hilft bei der Korrektur einzelner Gewichte. Der endgültige Fehler des CNNs wird auch Cost-Funktion (dt.: Kosten-Funktion)

genannt, welche aus allen Loss-Functions des CNNs gebildet wird. Dies kann beispielsweise durch Berechnung des mittleren quadratischen Fehlers (MSE) geschehen. Je nach Anwendung sind unterschiedliche Loss- und Cost- Functions anwendbar.

Gradientenabstiegsverfahren

Das Gradientenabstiegsverfahren ist eine Verallgemeinerung von verschiedenen Lernregeln. Ziel ist es, den möglichst kleinsten Fehlerwert für die Erkennung zu erreichen, also das Minimum der Loss-Function [4]. Indem die Veränderung des Fehlers nach Veränderung von Gewichten beobachtet wird, kann eine Richtung festgestellt werden, in welche man auf der Loss-Function wandert. Durch iterative Veränderungen der Gewichte nähert man sich dem Minimum der Loss-Function (Siehe Abbildung 2.13).

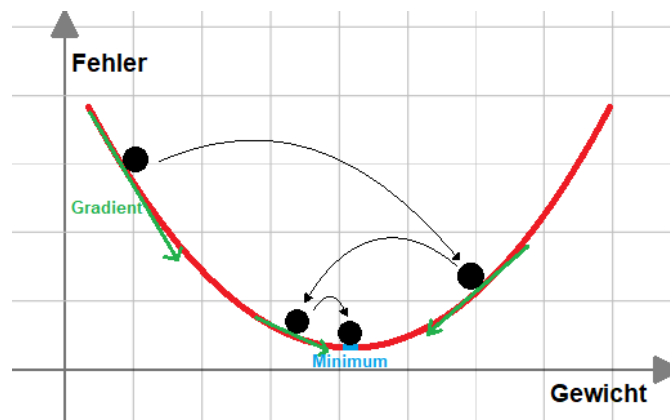


Abbildung 2.13: Gradientenabstieg zum Minimum der Fehlerfunktion

Eine Loss-Function kann aber auch mehrere lokale Minima aufweisen. Es kann passieren, dass man sich mit dem Gradientenabstiegsverfahren einem lokalen Minimum annähert und nicht das gewünschte globale Minimum erreicht. Außerdem kann der Gradient an einer Stelle so groß sein, dass das globale Minimum übersprungen wird (Siehe Abbildung 2.14). Auch ein Zurückkehren zum ursprünglichen Ort ist möglich, und es wird nie ein Minimum gefunden. Um zu einem sinnvollen Ergebnis zu kommen, muss der Vorgang mit unterschiedlichen Startwerten wiederholt werden.

Beim Training von CNNs wird die **Lernrate** im Laufe des Trainings verändert. Wie groß die folgenden Veränderungen an den Gewichten sein werden, kann mithilfe der Lernrate beeinflusst werden. Man startet mit einer großen Lernrate, um lokale Minima im besten Fall zu überspringen und nähert sich mit kleiner werdender Lernrate dem globalen

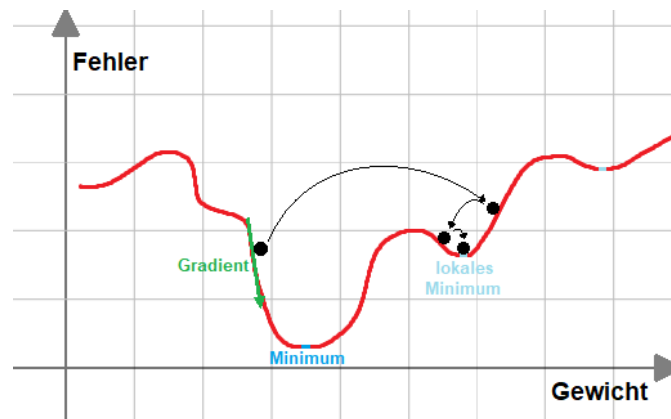


Abbildung 2.14: Gradientenabstieg zu einem lokalen Minimum der Fehlerfunktion

Minimum an. Hierbei ist zu beachten, dass eine geringe Lernrate dazu führt, dass viele Berechnungen durchgeführt werden müssen, um ein Minimum zu erreichen, dies ist teuer in der Berechnung. Eine große Lernrate führt schneller oder auch gar nicht zum Erfolg und ist nicht so genau wie eine kleine Lernrate. Hier muss ausprobiert werden, wie sich ein CNN beim Training mit unterschiedlichen Lernraten verhält.

Delta-Lernregel

Eine Methode zur Fehlerminimierung ist die Delta-Lernregel. Diese findet Anwendung in Netzen in denen es keine Hidden-Neuronen gibt, da zur Anwendung dieser Regel die gewünschte Gewichtung im Training bekannt sein muss, was für variierende Hidden-Layer nicht immer gegeben sein kann. Sie ist eine Form des Gradientenabstiegsverfahrens.

Die Delta-Lernregel lautet wie folgt: $\Delta W_{ik} = \varepsilon * \delta_i * a_k$

ΔW_{ik} = Gewichtsveränderung ΔW zwischen Output Neuron i und Input Neuron k .

ε = Lernrate, bestimmt in wie weit ein Gewicht angepasst wird.

δ_i = Fehler, Differenz zwischen Soll- und Ist-Aktivierungslevel $a_{i_{soll}} - a_{i_{ist}}$

a_k = Aktivierungslevel des Input Neuron k

Das in Abbildung 2.15 zu sehende Beispiel wurde mit einer Lernrate ε von 0,001 und ohne Aktivierungsfunktion durchgeführt. Man sieht, dass der Betrag des Fehlers mit der Zeit geringer wird.

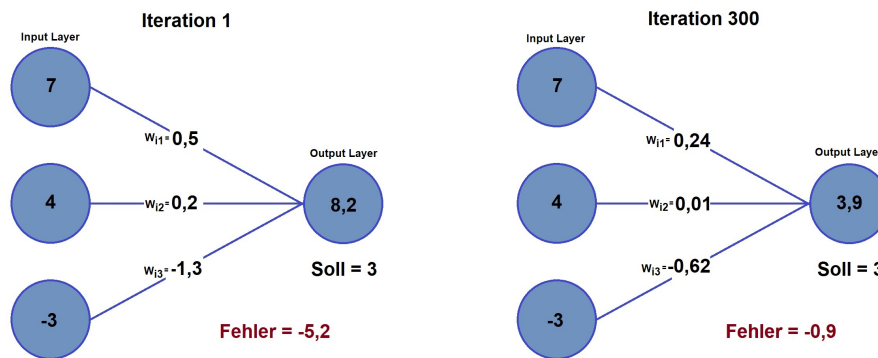


Abbildung 2.15: Beispiel der Delta-Lernregel nach 300 Iterationen

Wird mit einer Aktivierungsfunktion gearbeitet, so muss $\delta_i = a_{i_{soll}} - a_{i_{ist}}$ mit der ersten Ableitung der Aktivierungsfunktion an der Stelle des Netzeinputs (Summe aller Input-Neuronen-Werte) multipliziert werden: $\delta_i = \text{Aktivierungsfunktion}'(\text{Netzeinput}) * (a_{i_{soll}} - a_{i_{ist}})$

Backpropagation

Wird die Delta-Lernregel auf ein CNN mit mehreren Hidden-Layern angewandt, dann tritt folgendes Problem auf: Es gibt keinen Soll-Wert für Neuronen in Hidden-Layern. Somit kann $\delta_i = a_{i_{soll}} - a_{i_{ist}}$ nicht berechnet werden. In diesem Fall wird die **Backpropagation** (dt.: Fehlerrückführung) durchgeführt [37], indem eine Fallunterscheidung in der Berechnung von δ_i eingeführt wird:

$$\delta_i = \begin{cases} a_{i_{soll}} - a_{i_{ist}} , & \text{wenn output} \\ \sum_L \delta_l * w_{li} , & \text{wenn hidden} \end{cases}$$

L = Nächster Layer

l = Neuron des nächsten Layers

δ_l = Delta-Wert des nächsten Layers

w_{li} = Gewichtung von aktuellem Neuron zum nächsten Neuron

Die Backpropagation besteht aus einem Forwardpass und einem Backwardpass. Beim Forwardpass wird ein Input an das CNN gelegt, um irgendeinen Output zu generieren. Beim Backwardpass wird dann ein Fehlerwert δ_i bestimmt, welcher nach hinten durch

das CNN zu den Hidden-Layern gereicht wird, damit auch dort eine Fehlerberechnung und Gewichts Anpassung durchgeführt werden kann. Dies wird auch Chaining genannt [4].

Hierbei ist zu beachten, dass diese Fehlerkorrektur mit gleichbleibendem Input durchgeführt wurde. Die so erzeugten Gewichtungen der Neuronen müssen demnach auf unbekanntem Daten getestet werden, um zu schauen, ob das trainierte CNN generalisieren kann.

Over-, Underfitting

Ein trainiertes CNN wird nach dem Training, im besten Fall, auf Daten getestet, welche es im Training nicht kennengelernt hat. Hierbei können folgende Probleme auftreten: Overfitting und Underfitting [4].

Overfitting kommt dann vor, wenn das CNN gut auf den Trainingsdaten performt, aber nicht gut mit unbekanntem Daten umgehen kann. Das CNN kann nicht generalisieren. Dies kann oft schon während des Trainings festgestellt werden, wenn der Trainings-Fehlerwert signifikant geringer ist als der Validierungs-Fehlerwert. Overfitting tritt meistens dann beim Training auf, wenn die benutzten Trainingsdaten sich zu stark ähneln oder zu wenig Trainingsdaten vorliegen. Außerdem sollten die Trainingsdaten Variationen in den zu trainierenden Objekten beinhalten. Durch Augmentierung der Trainingsdaten können Daten beispielsweise soweit verändert werden, dass die groben Merkmale zum Vorschein kommen, und sich das CNN nicht auf Dinge fokussiert, welche nicht relevant sind.

Von **Underfitting** spricht man, wenn ein CNN nicht nur unbekanntem Daten nicht versteht, sondern schon bei den Trainingsdaten keine guten Ergebnisse erzielt. Um Underfitting zu vermeiden, ist es sinnvoll, durch andere Trainingsdaten dem CNN mehr Features zu geben. Indem Informationen hinzugefügt werden, die im ersten Moment unwichtig erscheinen, kann es dem CNN helfen, besser zu klassifizieren. Wenn man davon ausgeht, dass die Trainingsdaten gut sind, dann kommt ein Underfitting nur vor, wenn das gewählte Model nicht komplex genug ist. Mit dem Hinzufügen von mehr Layern kann man dafür sorgen, dass das CNN die Daten besser verarbeitet.

3 Stand der Technik

3.1 Entwicklung von Wettbewerbs-Datensätzen

3.1.1 VOC - Pascal Visual Object Classes Challenge (2005-2012)

Im Jahr 2005 wurde die erste PASCAL-Visual-Object-Classes-Challenge abgehalten (VOC). Die Aufgabe bestand unter anderem darin, mithilfe des gegebenen Datensatzes eine Methode im Bereich der Objekterkennung und Objektklassifikation zu entwickeln. Der Wettbewerb wurde jährlich veranstaltet und verbessert [10]. Im ersten Jahr bestand der Datensatz aus lediglich 1.578 Bildern mit 2.209 annotierten Objekten und vier Klassen und war somit wenig aussagekräftig [12].

Seit 2007 bis zum letzten Wettbewerb im Jahr 2012 wurde auf 20 Klassen trainiert. Im ersten Fall wurden 9.963 Bilder mit 24.640 Objekten zur Verfügung gestellt. Auf der einen Hälfte des Datensatzes wurde trainiert und validiert, auf der anderen getestet [10].

In 2012 standen 11.530 Bilder mit 27.450 Objekten zur Verfügung und wurden gleichermaßen aufgeteilt [11]. Speziell VOC-2007 und VOC-2012 werden bis heute als Datensatz zum Testen von neuronalen Netzen verwendet.

Die Bilder innerhalb des Datensatzes zeigen zum Großteil nur das zu erkennende Objekt mittig im Bild. Die Bilder sind sowohl horizontal als auch vertikal orientiert und haben unterschiedliche Größen. Die Maximalgröße eines Bildes beträgt 500x375 Pixel.

Datensatz	Jahr	Klassen	Training		Validierung		Train+Val		Test	
			Bilder	Objekte	Bilder	Objekte	Bilder	Objekte	Bilder	Objekte
VOC	2005	4	-	-	-	-	1.578	2.209	654	1.293
VOC	2007	20	2.501	6.301	2.510	6.307	5.011	12.608	4.952	14.976
VOC	2012	20	5.717	13.609	5.823	13.841	11.540	27.450	10.991	-

Tabelle 3.1: Bildverteilung der VOC Datensätze



Abbildung 3.1: Beispielbilder des VOC2007 Datensatzes

In einem abschließendem Bericht im Jahr 2014 kündigten die Gründer des Wettbewerbes M. Everingham et al. an, dass die Testdaten der Wettbewerbe 2008 bis 2012 nicht veröffentlicht werden [9]. Dadurch soll verhindert werden, dass zukünftige Methoden der Objekterkennung, welche mithilfe dieses Datensatzes trainiert wurden, overfitten und an Aussagekraft verlieren. In Kombination mit einem zuvor zur Verfügung gestellten Evaluation Server und damit eingehenden Ranglisten, ist es noch heutzutage möglich relevante Methoden zur Objekterkennung oder anderen Rubriken der VOC Challenge zu veröffentlichen.

3.1.2 ILSVRC - The ImageNet Large Scale Visual Recognition Challenge (2010-2017)

Im Jahr 2009 wurde ImageNet auf der IEEE Conference on Computer Vision and Pattern Recognition (CVPR) vorgestellt [6]. Zu diesem Zeitpunkt umfasste diese Datenbank über 3,2 Millionen Bildern aus 5247 Klassen. Heute umfasst ImageNet über 14 Millionen Hand-annotierte Bilder aus über 21.841 Klassen.

In den veranstalteten Wettbewerben waren diverse Aufgaben vorhanden. Die am häufigsten vorkommenden Rubriken waren die Folgenden: Image Classification, Classification and Localization (später Object localization), Object detection und Object detection from videos [38].

- Image Classification (2010 - 2014)

In dieser Kategorie sollte ein Algorithmus entwickelt werden, welcher für jedes Bild eine Liste aus maximal fünf auf dem Bild erkannten Objekten erstellt wird. Es gab zwei Testkriterien: 1) "non-hierarchical", und 2) "hierarchical". Im ersten Fall gilt die Klassifikation als Fehlerhaft, sobald die Ground-Truth, die per Hand vergebene

Annotation, nicht unter den fünf erkannten Objekten ist. So ist die Fehlerquote entweder 0 oder 1. Im zweiten Fall wird mithilfe der hierarchischen Struktur von WordNet/ImageNet gewertet. So ist es weniger schlimm, wenn der Algorithmus zwar einen Hund, aber die falsche Hunderasse erkennt, als wenn er einen Hund für einen Tisch hält. [39]

Der zum Trainieren gegebene Datensatz dieser Rubrik enthielt 1,2 Millionen Bilder aus 1000 Kategorien. 2010 und 2011 veränderte sich der Datensatz nicht. Im Jahr 2012 wurden die 1000 Kategorien teilweise verändert und blieben bis zum letzten Wettbewerb im Jahr 2017 unverändert.

- Classification and Localization (2011-2017)

Aufbauend auf der Image Classification Rubrik ist es bei dieser Rubrik relevant, eine Bounding Box um jedes der maximal fünf erkannten Objekte zu ziehen. In die Bewertung fließt nun zusätzlich die Abweichung der Bounding Box zur Ground-Truth Bounding Box. [39]

Da sich die Rubriken sehr ähnelten, wurde mit dem Jahr 2015 die alleinstehende Rubrik der Image Classification abgeschafft und als Teil der Classification und Localization Rubrik unter dem Namen Object Localization eingeführt.

In dieser Rubrik wird zum Trainieren der selbe Datensatz benutzt wie in der Image Classification Rubrik.

- Object Detection (2013-2017)

In dieser Rubrik wurden für jedes Bild eine Kombination aus Bounding Box, Klasse und Confidence-Score ermittelt. Mit der letzten VOC-Challenge im Jahr 2012 knüpft diese Rubrik genau dort an. Der im Jahr 2013 zur Verfügung gestellte Datensatz war mit 200 zu identifizierenden Klassen und über 400.000 Bildern für Training und Validierung in diesem Umfang nicht zu vergleichen mit anderen Datensätzen für die Objekterkennung. Die Eigenschaften der Bilder ähnelten denen der VOC-Challenge.

Im darauffolgenden Jahr erschien der ILSVRC-2014 Datensatz, an welchem bis zum letzten Wettbewerb in 2017 keine Veränderungen an Trainings- und Validierungsdaten vorgenommen wurde. Dieser Datensatz beinhaltet 534.309 Objekte aus 200 Klassen auf 476.688 Bildern zum Trainieren und Validieren. Außerdem 40.152 Testbilder im Jahre 2014 und 65.500 Testbilder im Jahre 2017.

3 Stand der Technik

Datensatz	Jahr	Bildauflösung	Klassen pro Bild	Objekte pro Bild	Objektanteil im Bild
VOC	2012	469x387 pixel	1,521	2,711	0,207
ILSVRC	2013	482x415 pixel	1,534	2,758	0,170

Tabelle 3.2: Vergleich von Eigenschaften des VOC-2012 und ILSVRC-2013 Datensatzes

- Object detection from videos (2015 - 2017)

In dieser Rubrik sollten Objekte in einem Video erkannt werden. 30 der 200 Klassen der Object Detection Rubrik wurden für diese Rubrik verwendet. Für jedes Video sollte der Algorithmus ein Set von Annotations generieren, welche die Bild-Nummer, die Klasse, den Confidence-Score und die Bounding Box Koordinaten enthält. Jede der 30 Klassen soll einmal vorkommen. Es wurden Punkte abgezogen, wenn eine Klasse doppelt oder nicht erkannt wurde.

Datensatz	Jahr	Klassen	Training		Validierung		Train+Val		Test	
			Bilder	Objekte	Bilder	Objekte	Bilder	Objekte	Bilder	Objekte
ILSVRC	2014	200	456.567	478.807	20.121	55.502	476.688	534.309	40.152	-
ILSVRC	2017	200	456.567	478.807	20.121	55.502	476.688	534.309	65.500	-

Tabelle 3.3: Bildverteilung der ImageNet Datensätze für Objekterkennung

Zum Zeitpunkt des Verfassens beinhaltet ImageNet über 14 Millionen Bildern aus 21.841 Klassen. Im Bereich der Objekterkennung bietet der Datensatz zu 3000 Klassen jeweils 150 Bilder mit Bounding Boxes [39].



Abbildung 3.2: Beispielbilder der ImageNet Datenbank

3.1.3 MS-COCO - Common Objects in Context Object Detection Challenge (2015 - Heute)

Basierend auf dem 2014 veröffentlichten Datensatz wurde 2015 die erste COCO Object Detection-Challenge durchgeführt. Der angegebene Datensatz wurde speziell für die Objekterkennung erstellt. Die vorangegangenen Datensätze stellten die zu erkennenden Objekte unrealistisch dar, indem das Objekt zum Beispiel mittig im Bild liegt, nicht durch andere Objekte verdeckt wird oder vorteilhaft zur Kamera gedreht ist. Mit COCO wurde versucht die Objekte in einem realistischen Kontext darzustellen, da zum Beispiel durch Größenunterschiede von oft miteinander auftauchenden Objekten auf weitere Informationen geschlossen werden kann [25].

Die Bilder des COCO Datensatzes haben sich seit der ersten Challenge nicht verändert. Lediglich die Aufteilung der Bilder für Training und Validierung wurde im Jahr 2017 verändert und die Anzahl der Testbilder reduziert. Der Datensatz umfasst 886.284 Objekte auf 123.287 Bildern aus 80 Klassen [26]. Außerdem enthält der COCO Datensatz im Durchschnitt 7,7 Objekte und 3,5 Klassen pro Bild [25]. Die Objekte sind außerdem um einiges kleiner und damit schwieriger zu erkennen, als in anderen Datensätzen. Die Maximalgröße eines Bildes beträgt bis zu 640x640 Pixel.



Abbildung 3.3: Beispielbilder des COCO Datensatzes

Im Jahr 2020 war die COCO-Challenge Teil des Joint COCO and LVIS Recognition Challenge Workshops der European Conference on Computer Vision. Für das Jahr 2021 ist keine COCO-Challenge geplant. Stattdessen wird eine Teilnahme bei LVIS 2021 Challenge and Workshop empfohlen.

Datensatz	Jahr	Klassen	Training		Validierung		Train+Val		Test	
			Bilder	Objekte	Bilder	Objekte	Bilder	Objekte	Bilder	Objekte
COCO	2015	80	82.783	604.907	40.504	291.875	123.287	896.782	80.000	-
COCO	2017	80	118.287	860.001	5.000	36.781	123.287	896.782	40.000	-

Tabelle 3.4: Bildverteilung der COCO Datensätze

3.1.4 OID - Open Images Detection Challenge (2018 - Heute)

Im Jahr 2018 fand der erste Open Images Detection Wettkampf statt. Dieser benutzte ein Subset (dt.: Anteil) des Open Images Dataset v4. Der gesamte Datensatz umfasst über neun Millionen klassifizierte Bilder, davon 1,74 Millionen Bilder mit 14,6 Millionen Bounding Boxes für 600 Klassen. Für den Wettbewerb werden jedoch nur 500 Klassen bewertet, da zu häufig auftretende Klassen wie beispielsweise *Clothing* davon ausgeschlossen wurden.[22]



Abbildung 3.4: Beispielbilder des Open Images Datensatzes

2019 wurden dem Datensatz mehr Bilder hinzugefügt, womit nun 1,9 Millionen Bilder im Datensatz zu finden sind. Zusätzlich wurden unter anderem object-segmentation-masks und visual-relationship-annotations hinzugefügt.

Der zu diesem Zeitpunkt aktuellste Datensatz vom Jahr 2020 enthält 16 Millionen Bounding Boxes für 600 Klassen auf 1,9 Millionen Bildern. Außerdem über 3,3 Millionen visual-relationship-annotations, welche Zusammenhänge zwischen Objekten auf einem Bild angeben, wie zum Beispiel: *Frau spielt Gitarre*, *Frau springt*, *Tisch ist aus Holz* [22]. Die Bilder stammen ursprünglich von flickr.com und haben diverse Größen. Für den Open Images Datensatz wurde die Größe dahingegen verändert, dass die längste Seite eines Bildes maximal 1024 Pixel hat, womit die maximale Bildauflösung 1024x1024 Pixel beträgt. Die Bilder des Trainingsdatensatzes haben durchschnittlich 8,4 Bounding Boxes und Klassen pro Bild. [24].

3 Stand der Technik

Im Jahr 2020 war der Datensatz Bestandteil der Robust Vision-Challenge 2020 und es wurde keine separate Open Images-Challenge abgehalten.

Datensatz	Jahr	Klassen	Training		Validierung		Train+Val		Test	
			Bilder	Objekte	Bilder	Objekte	Bilder	Objekte	Bilder	Objekte
OID	2018	600	1.743.042	14.610.229	41.620	204.621	1.784.662	14.814.850	125.436	625.282
OID	2020	600	1.743.042	14.610.229	41.620	303.980	1.784.662	14.914.209	125.436	937.327

Tabelle 3.5: Bildverteilung der Open Images Datensätze

Datensatz	Jahr	Klassen	Training		Validierung		Train+Val		Test	
			Bilder	Objekte	Bilder	Objekte	Bilder	Objekte	Bilder	Objekte
VOC	2005	4	-	-	-	-	1.578	2.209	654	1.293
VOC	2007	20	2.501	6.301	2.510	6.307	5.011	12.608	4.952	14.976
VOC	2012	20	5.717	13.609	5.823	13.841	11.540	27.450	10.991	-
ILSVRC	2014	200	456.567	478.807	20.121	55.502	476.688	534.309	40.152	-
COCO	2015	80	82.783	604.907	40.504	291.875	123.287	896.782	80.000	-
ILSVRC	2017	200	456.567	478.807	20.121	55.502	476.688	534.309	65.500	-
COCO	2017	80	118.287	860.001	5.000	36.781	123.287	896.782	40.000	-
OID	2018	600	1.743.042	14.610.229	41.620	204.621	1.784.662	14.814.850	125.436	625.282
OID	2020	600	1.743.042	14.610.229	41.620	303.980	1.784.662	14.914.209	125.436	937.327

Tabelle 3.6: Gesamtübersicht der Bildverteilung der erwähnten Datensätze

3.2 Entwicklung von Objekterkennung/-lokalisierung

Objekterkennungssysteme können generell in zwei Kategorien eingeteilt werden [21]: Two-stage Detektoren und single-stage Detektoren.

Der Hauptunterschied liegt darin, dass two-stage Detektoren zunächst eine oder mehrere Region's of Interest(RoI) festlegen, welche daraufhin erneut in ein neuronales Netz gegeben werden, um die Klassifizierung der Objekte durchzuführen.

Single-stage Detektoren hingegen generieren in einem Durchlauf Bounding Boxes um potenzielle Objekte herum und klassifizieren diese. Methoden zur Objekterkennung, welche vor der Definierung von two- und one-stage Detektoren erschienen, werden oft traditionelle Methoden genannt.

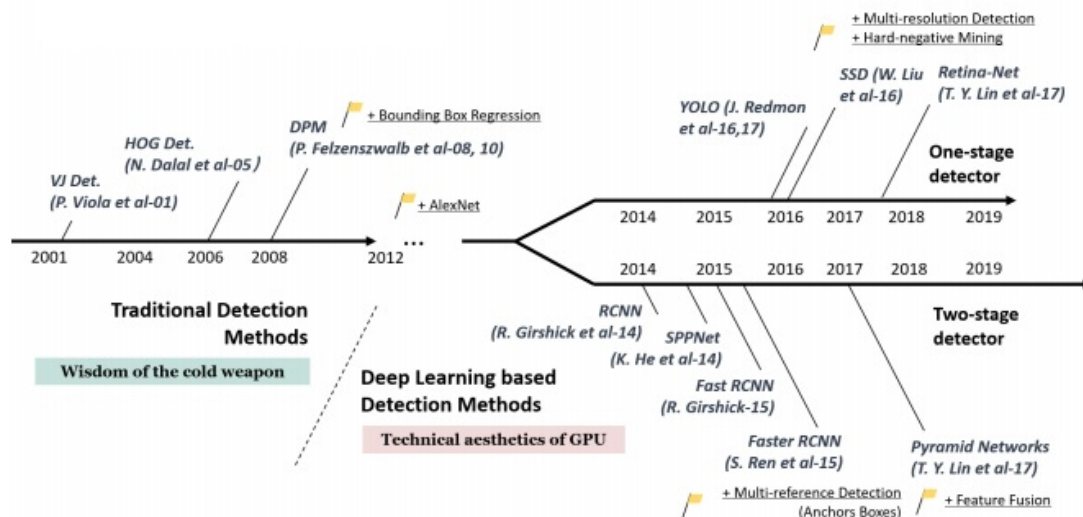


Abbildung 3.5: Übersicht von Objekterkennungsmethoden [54]

YOLO wurde erstmalig mit einer Veröffentlichung von Joseph et al. [34] im Jahr 2015 vorgestellt. Version 2 [35] und Version 3 [36] folgten im Jahr 2016 und 2018.

Zum Zeitpunkt der ersten Veröffentlichung von YOLO wurde der Standard der Objekterkennungssysteme durch die Detektoren DPM(Deformable Part Models) und RCNN(Region-based Convolutional Neural Networks) gebildet [34]. Im Folgenden werden diese Methoden der Objekterkennung, aber hauptsächlich die Entwicklung von YOLO, etwas genauer beschrieben.

Für Informationen über die allgemeine Entwicklung der Objekterkennung lohnt ein Blick in die Veröffentlichungen von Zhengxia Zou et al. [54] oder Licheng Jiao et al. [21] (siehe Abbildung 3.5).

Deformable Parts Models (DPM) wurden 2008 von Pedro F. Felzenszwalb et al. [13] vorgestellt. DPM ist eine Erweiterung, der in 2005 von N. Dalal und B. Triggs vorgestellten Histogram of Oriented Gradients (HOG) Detektoren [5], auch bekannt als Dalal-Triggs Detektor.

Ein DPM-Detektor besteht typischerweise aus einem Root-Filter und einer Reihe an Part-Filtern. Zunächst wird durch Sliding-Window-Detection eine Feature-Map des Inputs generiert, dazu wird die Methode der HOG-Detektoren [5] angewandt um HOG-Features zu extrahieren. Anschließend wird mithilfe eines Klassifizierers festgestellt, ob das gesuchte Objekt im Input zu finden ist oder nicht. Im Fall von HOG-Detektoren wurde dafür eine Support-Vector-Machine (SVM) benutzt. Anschließend werden die Part-Filter auf die vom Root-Filter erkannten Region's of Interest angewandt. Der Input wird dafür auf die doppelte Auflösung skaliert, um genauere Aussagen treffen zu können. Zum Schluss wird ein Kosten-Filter auf die untersuchten Bereiche angewandt, um den Confidence-Score der Detektion zu ermitteln.

Auf DPM basierende Detektoren gewannen die VOC-Challenge im Jahr 2007, 2008 und 2009 und waren somit der Höhepunkt der traditionellen Objekterkennung [54].

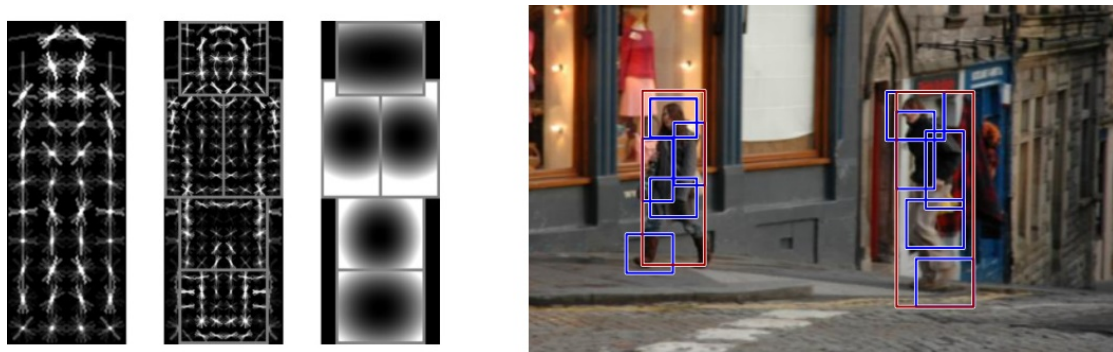


Abbildung 3.6: Beispiel für die Körpererkennung mithilfe von DPM [13]
v.l.n.r Ganzkörperfilter(root), Körperfilter in fünf Teilen(part), Kostenfilter, Ergebnis des Detektors

Für genauere Infos siehe Object Detection with Discriminatively Trained Part Based Models [13].

R-CNN wurde 2014 von R. Girshick et al. im Jahr 2014 vorgestellt. R-CNN besteht aus drei Modulen: Region-Proposal, CNN und Classifier (siehe Abbildung 3.7).

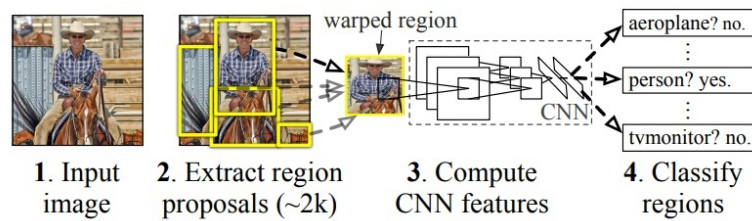


Abbildung 3.7: Pipeline von R-CNN [15]

Mithilfe von Selective Search (dt.: selektive Suche) werden Region Proposals (dt.: Bereichsvorhersagen) erstellt, welche zum Bestimmen von Bounding Boxes benutzt werden. Selective Search ist ein Algorithmus, welcher sowohl exhaustive-search (jede mögliche Position eines Objektes) und Segmentation (anhand von Eigenschaften des Bildes Zusammenhänge finden) kombiniert [45]. Im Falle von R-CNN wurden so ca. 2000 Region Proposals generiert (siehe Abbildung 3.8).



Abbildung 3.8: Bounding Boxes mithilfe von Selective Search

Nun wird jeder als Bounding Box markierter Kandidat von einem CNN (Alexnet/VGG) klassifiziert. Dazu wird der Inhalt einer Bounding Box ausgeschnitten und auf eine Größe von 227x227 Pixel transformiert. Nun existieren ca. 2000 Ergebnisse für verschiedene Objekte im Bild mit unterschiedlichen, durch die Klassifikation erstellten, Confidence-Scores. Am Ende werden alle Bereiche, deren Confidence-Score einen gewissen Wert nicht erreicht hat, verworfen. Übrig bleiben die Ergebnisse mit der höchsten Wahrscheinlichkeit, das klassifizierte Objekt zu enthalten [15].

Sowohl R-CNN als auch DPM benötigen viel Rechenaufwand und iterieren teilweise tausende Male über ein Bild, bevor sie zu einem Ergebnis kommen. Neben der hohen Genauigkeit dieser Netze ist jedoch auch die Dauer der Klassifikation lang.

3.2.1 YOLOv1 (2015)

Zweck von YOLO

Ziel von Joseph R. et al. [34] war ein Netzwerk zu erstellen, welches ein Quellbild nur einmal bearbeiten muss, bevor es zu einem Ergebnis kommt. Um dies zu erreichen, wählten die Autoren einen neuen Ansatz:

- Zunächst wird das zu überprüfende Bild mithilfe eines $S \times S$ Rasters unterteilt. YOLO wählt hierbei $S=7$.
- Die so entstandenen 49 Zellen bestimmen jeweils $B=2$ Bounding Boxes, mit dazugehörigen Confidence-Values. Diese Werte sind die Wahrscheinlichkeit, dass die Bounding Box ein Objekt enthält. Auch Zellen, welche kein Objekt enthalten bestimmen zwei Bounding Boxes, jedoch liegt die dazugehörige Confidence-Value dann unterhalb eines gesetzten Schwellenwertes und wird somit nicht weiter betrachtet.
- Desweiteren bestimmen die 49 Zellen jeweils eine Klassenwahrscheinlichkeit. Dies ist die Wahrscheinlichkeit dafür, dass falls diese Zelle ein Objekt enthält, es der vorausgesagten Klasse angehört.

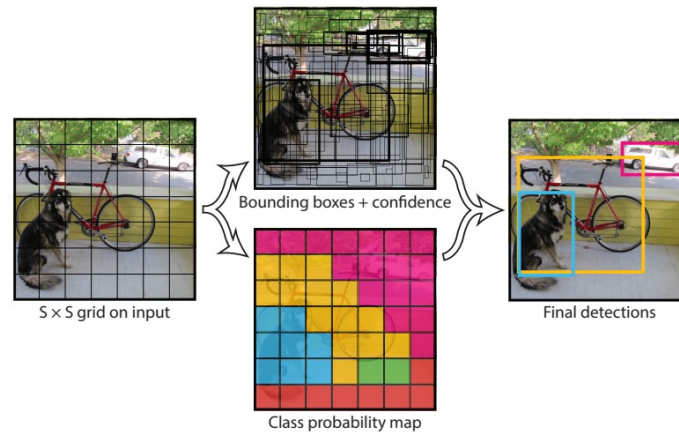


Abbildung 3.9: Pipeline von YOLO [34]

Die gesetzten Parameter ($S=7$, $B=2$) und die Anzahl der Klassen (C) mit denen YOLO trainiert wurde, limitieren die Größe des Output Tensors: $S \times S \times (B * 5 + C)$ Wird YOLO zum Beispiel mithilfe des Pascal-VOC Datensatzes trainiert, welcher 20 zu identifizierende Klassen besitzt, dann entspräche der Output Tensor der Größe $7 \times 7 \times 30$.

Architektur

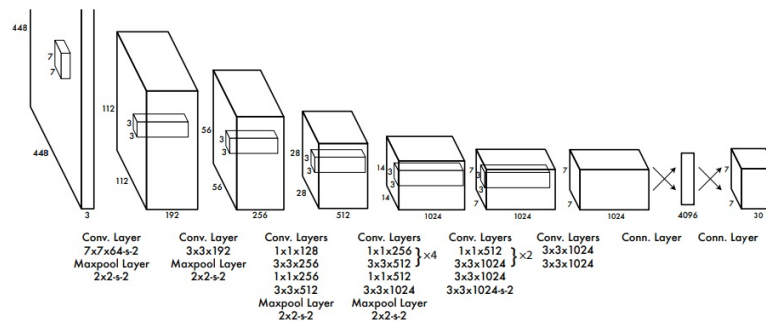


Abbildung 3.10: Architektur von YOLO [34]

- Das Netz ist von GoogleNet inspiriert.
- Der Input für dieses Netz ist ein Bild der Größe 224×224 , welches für die Bounding Box Detection auf 448×448 hoch skaliert wird, um kleinere Objekte besser erkennen zu können.
- Das Netz besteht aus 24 Convolutional- gefolgt von zwei Fully-Connected-Layern.
- Die ersten Convolutional-Layer dienen zur Feature-Extraction.
- Die Fully-Connected-Layers dienen zum Voraussagen von Bounding Box- und Klassenwahrscheinlichkeit.
- Anders als GoogleNet benutzt YOLO abwechselnd 1×1 Reduction-Layers und 3×3 Convolutional-Layers (Siehe Abbildung 3.10).

Neben YOLO wurde außerdem auch eine noch schnellere Version von YOLO vorgestellt: Fast-YOLO. Diese Version von YOLO benutzt nur 9, anstelle von 24, Convolutional-Layern und ist somit schneller, aber auch ungenauer.

Vergleich zu anderen neuronalen Netzen

YOLO erreicht bei einer Geschwindigkeit von 45 FPS eine Genauigkeit von 63,4% (mit einer Titan X GPU). Wie in der Tabelle der Abbildung 3.11 zu sehen, ist YOLO somit

Real-Time Detectors	Train	mAP	FPS
100Hz DPM	2007	16.0	100
30Hz DPM	2007	26.1	30
Fast YOLO	2007+2012	52.7	155
YOLO	2007+2012	63.4	45
<hr/>			
Less Than Real-Time			
Fastest DPM	2007	30.4	15
R-CNN Minus R	2007	53.5	6
Fast R-CNN	2007+2012	70.0	0.5
Faster R-CNN VGG-16	2007+2012	73.2	7
Faster R-CNN ZF	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

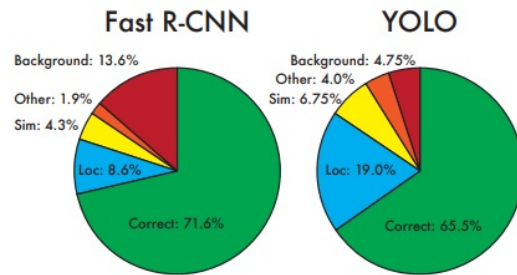


Abbildung 3.11: Übersicht Objekterkennungssysteme [34]

weitaus genauer und schneller als vergleichbare Echtzeit-Systeme dieser Zeit. Sogar Fast-YOLO erreicht mit einer Genauigkeit von 52,7% noch akzeptable Werte, diese sogar mit einer Geschwindigkeit von 155 FPS.

Die Tortendiagramme der Abbildung 3.11 zeigen die Anzahl an korrekten Lokalisierungen (*Correct*), auch true-positives genannt. Außerdem zeigt das Diagramm falsche Klassifizierungen (*Other*), also false-negatives. Zudem sind auch die false-positives (*Background*) aufgezeichnet, das Erkennen von Objekten wo keine sind. Zuletzt werden die Lokalisierungsfehler (*Loc*) angezeigt, also richtige Vorhersagen, welche nicht mit dem Original überlappen. Jeweils für Fast R-CNN und YOLO. Die im Diagramm als *Sim* markierten Fehler entsprechen gleichen Klassifikationen mit einer zu geringen Intersection of Union (IOU).

Die größte Schwäche von YOLO ist die korrekte Lokalisierung von Objekten. Bei YOLO machen diese Fehler 19,0% des Präzisionsverlustes aus und somit mehr als alle anderen Fehlerquellen zusammen. Bei R-CNN liegen die Lokalisierungsfehler nur bei 8,6%. Fast R-CNN erkennt jedoch zu 13,6% false-positives, welche bei YOLO nur 4,75% ausmachen.

Joseph R. et al. [34] kombinierten YOLO und Fast R-CNN. Mithilfe der geringen false-positives von YOLO und der geringeren Lokalisierungsfehler von Fast R-CNN, gelang eine Verbesserung der Genauigkeit von Fast R-CNN um 3,2%, von 71,8% auf 75,0%. Dies hatte keinen Einfluss auf die Geschwindigkeit, da beide Netze unabhängig voneinander ausgeführt wurden und einzig die Ergebnisse kombiniert wurden. Jedoch ist YOLO so schnell, dass es keine signifikanten Geschwindigkeitsverluste einbrachte [34].

Neben der schlechten Objekt-Lokalisierung ist ein weiterer Nachteil von YOLO, dass mit der festen Anzahl an Zellen maximal 49 Objekte erkannt werden können.

3.2.2 YOLOv2 (2016)

Im Jahr 2016 veröffentlichten Joseph R. et al. [36] Version 2 von YOLO. Der Fokus lag hier auf der Erhöhung der Genauigkeit der Objekt-Lokalisierung. Nennenswert sind dabei die folgenden Strategien:

- **Batch Normalization:** Durch das Normalisieren aller Convolutional-Layers verhindert man Instabilität innerhalb des Netzes. Neben dem zuvor schon normalisierten Input der Daten ist nun auch der Output eines jeden Layers normalisiert und dadurch das Netz stabilisiert. Andere Formen der Regulierung der Daten werden überflüssig. YOLO erzielt dadurch eine Verbesserung der Genauigkeit um 2,4%.
- **High Resolution Classifier:** Die ursprüngliche Version von YOLO trainierte das Klassifizierungsnetzwerk mit Bildern der Größe 224×224 Pixel und erhöhte die Auflösung auf 448×448 Pixel für die Detektion. Mit YOLOv2 wird das Netz anfangs mit Bildern mit einer Auflösung von 448×448 Pixel trainiert. Dadurch fällt es dem Netz leichter die spätere Detektion durchzuführen. Hiermit wurde eine Genauigkeitserhöhung von 3,7% erzielt.
- **Anchor Boxes:** Zuvor wurden für Bounding Boxes die X- und Y-Koordinaten, Breite und Höhe der Bounding Box von zwei Fully-Connected-Layern vorausgesagt. Andere Netze wie zum Beispiel Faster R-CNN besitzen Anchor Boxes, welche zuvor festgelegt wurden und bestimmen Offsets für diese. Offsets anstelle von Koordinaten vorauszusagen vereinfacht das Problem und ist für das Netz einfacher zu lernen. Der Einsatz von Anchor Boxes hat jedoch die allgemeine Genauigkeit des Netzes nicht erhöht, dafür aber eine Verbesserung in der Bestimmung der Bounding Boxes hervorgerufen. Zuvor konnte das Netz nur 98 Bounding Boxes voraussagen, doch mit Anchor Boxes sind es über Tausend bei gleichbleibender Genauigkeit.
- **Dimension Clusters:** Im Gegensatz zu Fast R-CNN, welches zuvor bestimmte Anchor Boxes festlegte, werden bei YOLOv2 mithilfe eines k-means-Algorithmus automatisch Basis Anchor Boxes gefunden (Priors). Für den Algorithmus wurde $k=5$ gewählt, da hier die Modellkomplexität nicht darunter leidet und die Genauigkeit erhöht wurde.
- **Multi-Scale Training:** YOLOv1 wurde mit nur einer Input-Größe von 448×448 Pixel trainiert. Mit dem Einsetzen von Anchor Boxes wurde diese Größe auf

416 × 416 Pixel geändert. Damit YOLOv2 robust gegenüber verschiedenster Input-Größen ist, wird während des Trainings nach allen zehn Iterationen die Input-Größe auf ein Vielfaches von 32 verändert 320,352,...,608. Dadurch kann das Netz im späteren Gebrauch besser mit variierenden Größen umgehen und bessere Voraussagen treffen.

- **Direct location prediction:** Mit dem Einsatz von Anchor boxes traten Instabilitätsprobleme in den anfänglichen Iterationen auf, weswegen man sich dafür entschieden hat, anstelle des Offsets wieder Koordinaten zu bestimmen. Das Netz bestimmt fünf Bounding Boxes pro Zelle und fünf Koordinaten pro Bounding Box.

	YOLO								YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓				
new network?					✓	✓	✓	✓	✓
dimension priors?						✓	✓	✓	✓
location prediction?						✓	✓	✓	✓
passthrough?							✓	✓	✓
multi-scale?								✓	✓
hi-res detector?									✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	78.6

Abbildung 3.12: Übergang von YOLO zu YOLOv2 [35]

Architektur

Die Basis des so entstandenen YOLOv2, Darknet-19, wird als Bild Klassifizierungs-Model benutzt. Es besteht aus 19 Convolutional- und fünf Maxpooling Layers. Siehe Abbildung 3.13

3.2.3 YOLO9000

Mithilfe von WordTree war es möglich mehrere Datensätze zu vereinen. So wurden die Annotations von COCO für die Objekterkennung, und ImageNet für die Bildklassifikation vereint um einen riesigen Trainingsdatensatz zu bilden um einen Large-Scale-Detector zu trainieren. Ein Datensatz mit 9418 Klassen steht zur Verfügung und das Ergebnis des Trainings war YOLO9000. Ein Großteil der Daten von ImageNet hat jedoch keine Informationen über Bounding Boxes. Das Netz lernt also das bestimmen der Bounding

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Abbildung 3.13: Darknet-19 [35]

Boxes durch die von COCO gestellten Daten. Die Klassen werden durch ImageNet gelernt. YOLO9000 wurde mithilfe des ILSVRC-Detection-Task getestet. Die Daten dieses Challenge-Datensatzes teilen nur 44 gemeinsame Klassen mit COCO, was bedeutet, dass YOLO9000 für einen sehr großen Teil der Daten keine Trainingsinformationen bezüglich der Detektion hatte und erreichte trotzdem eine allgemeine Genauigkeit von 19.7% und eine Genauigkeit von 16.0% bei 156 Klassen für welche es noch nie Detektionsdaten gesehen hat. Diese Werte sind besser als die von DPM seiner Zeit.

3.2.4 YOLOv3 (2018)

In den Jahren vor Erscheinen von YOLOv3 gab es einige vielversprechende Objekterkennungs-Modelle, welche fast alle auf ResNet-101 basierten. Die beste Erkennungsrate erzielte unter ihnen RetinaNet. YOLOv3 [36] erreichte im Jahr 2018 jedoch vergleichbare Werte mit geringeren Rechenzeiten.

Mit den folgenden Strategien gelang es weitere Verbesserungen der Genauigkeit zu erzielen:

- **Bounding box predictions:** YOLOv3 benutzt wie YOLOv2 Dimension-Clusters um Anchor Boxes zu generieren. In Version 3 werden jedoch nur vier Koordinaten bestimmt. YOLOv3 bestimmt eine Confidence-Score für jede Bounding Box.

	backbone	AP	AP ₅₀	AP ₇₅	AP _S	AP _M	AP _L
<i>Two-stage methods</i>							
Faster R-CNN+++	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI	Inception-ResNet-v2	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	52.1
<i>One-stage methods</i>							
YOLOv2	DarkNet-19	21.6	44.0	19.2	5.0	22.4	35.5
SSD513	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet	ResNeXt-101-FPN	40.8	61.1	44.1	24.1	44.2	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

Abbildung 3.14: Vergleich von YOLOv3 mit anderen Objektdetektoren [36]

Wenn eine Anchor Box der gegebenen Ground Truth beim Training zu mehr als einem bestimmten Schwellwert überlappt, so wird die Voraussage verworfen und die Anchor Box benutzt.

- **Multi-Class predictions:** Wenn viele verschiedene Klassen trainiert wurden, kann es zu Problemen kommen. Zum Beispiel kann ein Objekt eine Person und eine Frau sein. Vorher wurde nur eine potentielle Klasse bestimmt, nun können mehrere Klassen bestimmt werden.
- **Predictions across scales:** YOLOv3 bestimmt Bounding Boxes in drei verschiedenen Skalierungen der Input-Größe, um die Genauigkeit für kleine Objekte zu erhöhen. Jedoch führt dies zu schlechteren Genauigkeiten bei der Erkennung von größeren Objekten.

Architektur

YOLOv3 benutzt eine neue Architektur für die Feature-Extraction. Es besteht aus 53 Convolutional-Layer und trägt daher den Namen Darknet-53. Außerdem besitzt Darknet-53 Shortcut-Connections. Siehe Abbildung 3.15.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1x	Convolutional	32	1 × 1	128 × 128
	Convolutional	64	3 × 3	
	Residual			
	Convolutional	128	3 × 3 / 2	64 × 64
2x	Convolutional	64	1 × 1	64 × 64
	Convolutional	128	3 × 3	
	Residual			
	Convolutional	256	3 × 3 / 2	32 × 32
8x	Convolutional	128	1 × 1	32 × 32
	Convolutional	256	3 × 3	
	Residual			
	Convolutional	512	3 × 3 / 2	16 × 16
8x	Convolutional	256	1 × 1	16 × 16
	Convolutional	512	3 × 3	
	Residual			
	Convolutional	1024	3 × 3 / 2	8 × 8
4x	Convolutional	512	1 × 1	8 × 8
	Convolutional	1024	3 × 3	
	Residual			
	Avgpool		Global	
	Connected		1000	
	Softmax			

Abbildung 3.15: Architektur von Darknet-53 [36]

3.2.5 YOLOv4 (2020)

Joseph R. hörte mit der Entwicklung von YOLO auf, da er die Verwendung von YOLO im militärischen Bereich nicht weiter vorantreiben wollte. Weitere Fortschritte an YOLO wurden durch Alexey Bochkovskiy et al. [2] erzielt. Alexey B. hat schon vorher durch Optimierungen des Darknet Frameworks zur Entwicklung von YOLO beigetragen.

YOLOv4 ist das Ergebnis einer Reihe an Optimierungen. Im Folgenden werden die Methoden erläutert, welche es in YOLOv4 geschafft haben. Für eine vollständige Übersicht über alle Methoden, welche während der Entwicklung von YOLOv4 angewandt wurden, kann in der Veröffentlichung von Alexey Bochkovskiy et al. [2] nachgelesen werden.

Die Autoren sprechen in ihrer Arbeit über den allgemeinen Aufbau von One- und Two-Stage Detektoren, welcher auch für YOLOv4 zutreffend ist (Siehe Abbildung 3.16), und nennen mögliche Kandidaten für die jeweiligen Module.

Das **Backbone**-Netzwerk eines Objektdetektors ist häufig ein vor-trainiertes Netz, welches für die gewünschte Detektion nur noch angepasst werden muss. Die Autoren nennen dabei drei Netzwerke: ResNext50, Darknet53 und EfficientNet-B3.

Das YOLO eigene Darknet53, sowie ResNext50 basieren auf DenseNet. DenseNet besteht aus mehreren Convolutional-Layern, welche aus einem Batch-Normalization-Layer, einem

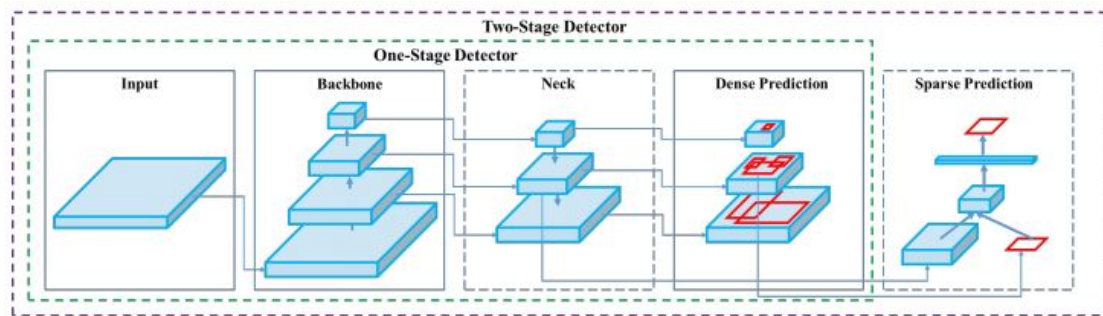


Abbildung 3.16: Aufbau von One- und Two-Stage Detektoren [2]

ReLU-Layer und von einem Convolutional-Layer bestehen. Als Teil der Verbesserungen für YOLOv4 werden die DenseNet-basierenden Netze mithilfe von Cross-Stage-Partial-Connections (CSP) verbessert [46]. Hierbei wird dafür gesorgt, dass alle Layer ihren Input von allen vorherigen Layern bekommen (Siehe Abbildung 3.17). Mit CSP verbesserte Netze tragen oft CSP im Namen.

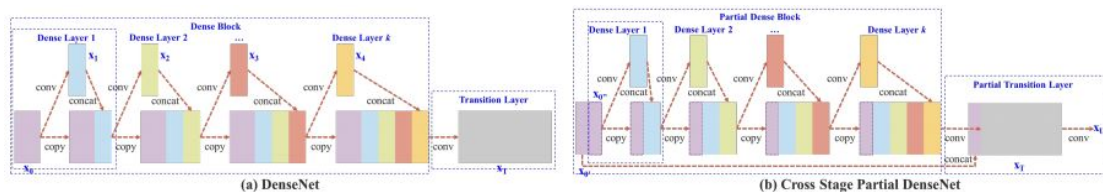


Abbildung 3.17: Cross-Stage-Partial-Connections [46]

EfficientNet war das Ergebnis einer Untersuchung von Skalierungsproblemen von neuronalen Netzen von Mingxing Tan und Quoc V. Le [44]. (Siehe Abbildung 3.18)

Die Autoren Alexey Bochkovskiy et al. beschreiben das optimale Modell für Objekt-Klassifikation wie folgt,

- große Input-Bild-Dimensionen, für besseres Erkennen von kleinen Objekten,
- mehr Layer, um die steigende Größe des Netzes abzudecken
- und mehr Parameter, für mehr Kapazität für die Erkennung mehrerer Objekte verschiedener Größen.

Unter Berücksichtigung der gewünschten Eigenschaften fiel die Wahl auf CSPDarknet53 (Siehe Abbildung 3.19).

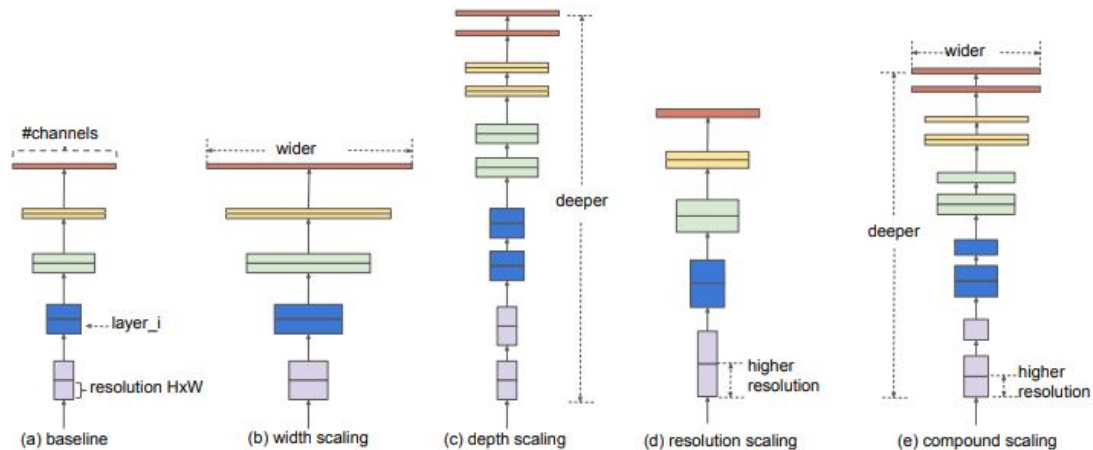


Abbildung 3.18: Verschiedene Skalierungen von neuronalen Netzen [44]

Backbone model	Input network resolution	Receptive field size	Parameters	Average size of layer output (WxHxC)	BFLOPs (512x512 network resolution)	FPS (GPU RTX 2070)
CSPResNext50	512x512	425x425	20.6 M	1058 K	31 (15.5 FMA)	62
CSPDarknet53	512x512	725x725	27.6 M	950 K	52 (26.0 FMA)	66
EfficientNet-B3 (ours)	512x512	1311x1311	12.0 M	668 K	11 (5.5 FMA)	26

Abbildung 3.19: Vergleich von ResNext50, Darknet53 und EfficientNet-B3

Der **Neck**-Teil eines Objektdetektors liegt zwischen dem Backbone und dem Head und ist häufig dafür zuständig, die vom Backbone generierten Features weiterzuverarbeiten und für die folgende Detektion vorzubereiten.

YOLOv4 entschied sich hier für Spatial-Pyramid-Pooling (SPP)[17] und Path-Aggregation-Network (PAN)[28].

Ein SPP-Layer ersetzt das letzte Pooling-Layer des Netzes. Es wendet verschiedene Maßstäbe auf Bilder an, und stellt diese so in verschiedenen Auflösungen dar. Dies sorgt für eine größere Robustheit des Detektors gegenüber Deformation von Bildern.

Ein PAN sorgt für die bessere Verteilung von Informationen innerhalb eines Netzes. Wenn Informationen durch viele Convolutional-Layer verarbeitet wurden, können Informationen verloren gehen. Mit direkten Verbindungen von Layern, mit wichtigen Informationen, können diese, ähnlich wie bei DenseNet, direkt an tiefer liegende Layer weitergegeben werden. Im Falle von PAN jedoch nur an ausgewählte Layer. Dies sorgt für eine erhöhte Genauigkeit des Netzes.

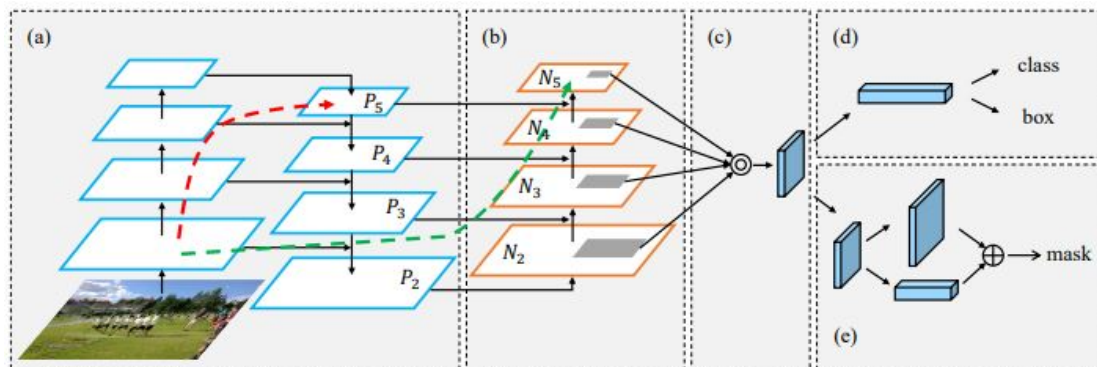


Abbildung 3.20: Aufbau eines Path Aggregation Network (PAN)[28]

Im **Head** werden die Klassen und Bounding Boxes für Objekte im Bild generiert und vorausgesagt. YOLOv4 verwendet hier die bereits in YOLOv3 verwendeten Anker basierte Bounding Box Methode.

Desweiteren wird von Bag-of-Freebies und Bag-of-Specials gesprochen. **Bag of freebies** ist ein Begriff für Methoden, welche die Genauigkeit oder Robustheit eines Detektors erhöhen. Zum Beispiel durch Änderungen an der Art des Trainings oder an den Trainingsdaten. Eine Erhöhung der Kosten des Trainings wird dafür in Kauf genommen. Diese Definition von Bag-of-Freebies trifft beispielsweise auf Data Augmentation zu. Hierbei wird die Variabilität der Input-Bilder durch verschiedene Methoden erhöht, was das Netz im Training robuster macht.

Inhalt des Bags-of-Freebies für YOLOv4 sind für den Backbone unter anderem:

- *CutMix*: Kombination von zwei Bildeausschnitten [49],
- *Mosaic-Data-Augmentation*: Kombination von vier Bildeausschnitten [2],
- *DropBlock-Regularization*: Teile der Bildinformationen fallen lassen, [14]
- *Class Label Smoothing*: Klassenbewertung auflockern [43].

Für den Detektor werden des weiteren folgende Freebies gewählt:

- *CIoU-loss*: IoU der generierten Bounding Box mit der Ground Truth. Bezieht die Entfernung der Mittelpunkte der Boxen mit ein[51],
- *Cross-mini-Batch-Normalization(CmBN)*: Batch Normalisierung für einzelne GPUs [2],

- *Self-Adversarial-Training(SAT)*: Veränderung des Inputs, sodass die Erkennung des Netzes am schlechtesten ist. Sorgt für eine verallgemeinerung des Modells und reduziert Overfitting [52],
- *Cosine-Annealing-Scheduler*: Verändert die Lernrate beim Training. Die Lernrate wird in Intervallen sehr weit minimiert, um einen warmen Neustart des Lernens zu bewirken [29],
- mehrere Anker für eine Ground Truth [2],
- Eliminierung der Raster Sensitivität [2],
- optimale Hyper Parameter [2],
- zufällige Trainingsgrößen [2].

Bag-of-Specials sind Methoden, welche in der Nachbereitung der Daten die Genauigkeit der Objekterkennung erhöhen. Diese Methoden erhöhen jedoch auch die Kosten der Inferenz des Netzes. Hierbei muss abgewägt werden, ob die Verbesserung der Genauigkeit des Netzes einen Verlust an Performance toleriert.

YOLOv4 verwendet für den Backbone folgende Specials:

- *Mish-activation*: Aktivierungsfunktion, welche sich ähnlich wie ReLU verhält, jedoch leicht negative Werte zulässt [30],
- *Cross-Stage-Partial-Connections(CSP)*: Siehe 3.2.5 - Backbone [46],
- *Multi-input-Weighted-Residual-Connections(MiWRC)*: Netze können Layer überspringen [2].

Für den Detektor werden unter anderem folgende Specials verwendet:

- *Spatial-Pyramid-Pooling-Block(SPP)*: Siehe 3.2.5 - Neck [17],
- *Spatial-Attention-Module(SAM)*: Modul zum setzen des Fokus der Objekterkennung, durch minderung von unnützen Informationen im Bild [47],
- *Path-Aggregation-Network(PAN)*: Siehe 3.2.5 - Neck [28],
- *DIoU-NMS*: Inbetrachtung der Distanz des Mittelpunkt eines Objektes zur Bounding Box in Kombination mit Non-Maximum-Suppression [51].

Ergebnisse

Die Autoren führten mit den genannten und diverser anderer Methoden Ablationsstudien durch, um Methoden zu finden, welche eine Verbesserung für die Objekterkennung darstellen. Die oben genannten Methoden sind das Ergebnis dieser Studien (Siehe Abbildung 3.23).

S	M	IT	GA	LS	CBN	CA	DM	OA	loss	AP	AP ₅₀	AP ₇₅
									MSE	38.0%	60.0%	40.8%
✓									MSE	37.7%	59.9%	40.5%
	✓								MSE	39.1%	61.8%	42.0%
		✓							MSE	36.9%	59.7%	39.4%
			✓						MSE	38.9%	61.7%	41.9%
				✓					MSE	33.0%	55.4%	35.4%
					✓				MSE	38.4%	60.7%	41.3%
						✓			MSE	38.7%	60.7%	41.9%
							✓		MSE	35.3%	57.2%	38.0%
✓									GIoU	39.4%	59.4%	42.5%
✓									DIoU	39.1%	58.8%	42.1%
✓									CIoU	39.6%	59.2%	42.6%
✓	✓	✓	✓						CIoU	41.5%	64.0%	44.8%
	✓		✓					✓	CIoU	36.1%	56.5%	38.4%
✓	✓	✓	✓					✓	MSE	40.3%	64.0%	43.1%
✓	✓	✓	✓					✓	GIoU	42.4%	64.4%	45.9%
✓	✓	✓	✓					✓	CIoU	42.4%	64.4%	45.9%

Abbildung 3.21: Ablationsstudie der Bag of Freebies Methoden

Model	AP	AP ₅₀	AP ₇₅
CSPResNeXt50-PANet-SPP	42.4%	64.4%	45.9%
CSPResNeXt50-PANet-SPP-RFB	41.8%	62.7%	45.1%
CSPResNeXt50-PANet-SPP-SAM	42.7%	64.6%	46.3%
CSPResNeXt50-PANet-SPP-SAM-G	41.6%	62.7%	45.0%
CSPResNeXt50-PANet-SPP-ASFF-RFB	41.1%	62.6%	44.4%

Abbildung 3.22: Ablationsstudie der Bag of Specials Methoden

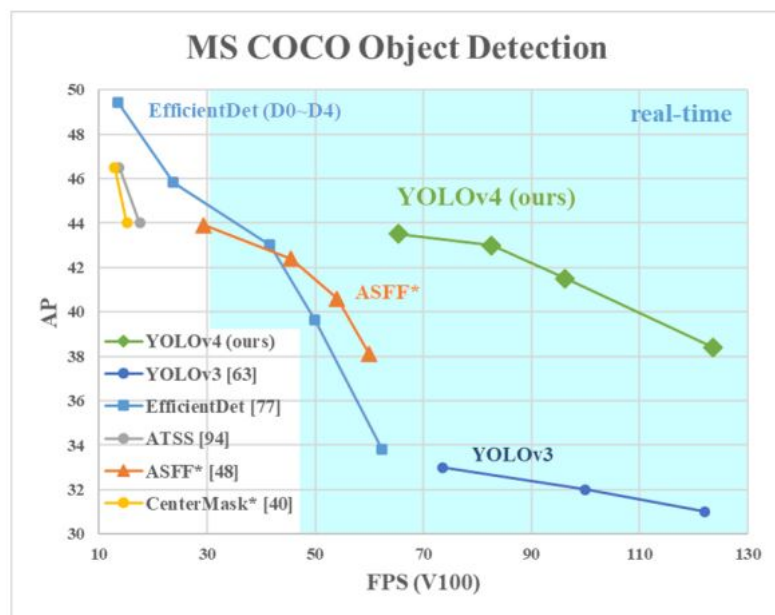


Abbildung 3.23: Vergleich von verschiedenen Objektdetektoren

4 Training des Netzes

Das Training sowie die Ausführung von YOLO wird im Rahmen dieser Arbeit mithilfe des Darknet-Frameworks durchgeführt. Eine Anleitung zum Aufsetzen einer Maschine zum Trainieren und Ausführen von YOLO in Darknet ist im Anhang A.0.1 zu finden.

Es sind bereits vor-trainierte Gewichte für YOLO vorhanden, welche auf verschiedenen Datensätzen trainiert wurden, hauptsächlich auf dem COCO Datensatz. Im Folgenden soll ein vor-trainiertes Netz auf die von uns gewünschten Klassen zugeschnitten werden, um die Genauigkeit des Detektors für diese Klassen zu verbessern.

4.1 Auswahl und Aufbereitung des Datensatzes

Von einem Datensatz wird für das Training ein Bild mit den gewünschten Objekten und einer Annotation benötigt. Die Annotation beinhaltet die Informationen der Bounding Box für die Objekte des dazugehörigen Bildes.

Im Rahmen dieser Arbeit wurden Trainingsdaten aus dem COCO-Datensatz (Siehe Kapitel 3.1.3) und dem Open-Images-Datensatz (Siehe Kapitel 3.1.4) entnommen, da diese besser für die Objekterkennung geeignet sind.

Für die geplante Indoor Anwendung sind die folgenden Klassen von Interesse:

Door, Person, Chair, Table.

Den kompletten Datensatz von COCO kann man sich auf der Website <https://cocodataset.org/#download> herunterladen. Die Annotations befinden sich im .json Format. Wenn man nur Bilder einer bestimmten Klasse benötigt, muss man die gewünschten Bilder über die .json-Datei finden und herunterladen. Ein Tool, welches dabei hilft ist coco-manager. Dieses erstellt aus der .json-Datei mit den Informationen über den gesamten Datensatz eine neue .json-Datei, welche nur die gewünschten Klassen enthält. (<https://>

[//github.com/immersive-limit/coco-manager](https://github.com/immersive-limit/coco-manager)). Um nun die Bilder herunterzuladen und die Annotations zu erstellen, wurde das Script um diese Funktionen erweitert (Siehe Anhang A.0.3). Im COCO-Format sehen die Bounding Box Informationen wie folgt aus:

```
[x, y, width, height]
```

Wobei x und y hier den Ursprung der Bounding Box, die obere Linke Ecke definieren.

Für den Download der Bilder und Annotations des Open-Images-Datensatzes gibt es ein Tool Namens OIDv4ToolKit (https://github.com/EscVM/OIDv4_ToolKit). Hier werden die Annotations aus einer .csv-Datei entnommen und als .txt-Datei gespeichert. Jedoch sind diese Annotations nicht im richtigen Format, weswegen hier mithilfe eines Python Scripts die Annotations umgeschrieben wurden (Siehe Anhang A.0.2). Im Open-Images-Format sehen die Bounding Box Informationen wie folgt aus:

```
[XMin, XMax, YMin, YMax]
```

Annotations Format

Die Annotation-Dateien müssen die Position und die Kategorie der Bounding Box beinhalten. Die Position der Bounding Box entspricht hier dem Mittelpunkt der Box. Die Annotations müssen im Falle von YOLO in Darknet als .txt-Datei vorliegen. Jede .txt-Datei trägt den Namen der .jpg/.png-Datei, für welche sie Daten beinhaltet. Eine Zeile innerhalb dieser .txt-Datei steht für eine Bounding Box und sieht aus wie folgt:

```
<object-class> <x> <y> <width> <height>
```

Legende:

- object-class - INTEGER des Objekts welches Trainiert wird
- x - Gleitkommazahl, relativ zur Breite des Bildes (0,0 bis 1,0)
- y - Gleitkommazahl, relativ zur Höhe des Bildes (0,0 bis 1,0)
- width - Breite der Bounding Box (0,0 bis 1,0)
- height - Höhe der Bounding Box (0,0 bis 1,0)

Sowohl die Annotation-Daten von COCO als auch Open-Images und dem OIDV4ToolKit liegen nicht in diesem Format vor, weswegen sie in allen Fällen umgeschrieben werden müssen.

Die Umrechnung der Bounding Box Informationen wurde mithilfe von Python Scripts durchgeführt. Für die COCO Informationen wurde das coco-manager Script um das Herunterladen der Bilder und der Erstellung umgerechneter Bounding Box Informationen und Annotations erweitert.

Für die Annotations des Open-Images-Datensatzes wurde mithilfe eines Python Scripts die vom Toolkit bereits generierten Annotations umgeschrieben.

4.2 Vorbereitung von Darknet

Bevor das Training gestartet werden kann, werden noch einige Daten benötigt, damit Darknet richtig vorbereitet ist.

Die im vorherigen Schritt erhaltenen Bilder und Annotations werden nun in das Darknet Repository kopiert. Danach muss die .cfg-Datei entsprechend der Daten angepasst werden:

yolov3.cfg

```
# Testing
# batch=1
# subdivisions=1
# Training
batch=64
subdivisions=32
...

learning_rate=0.001
burn_in=1000
max_batches = 8000
policy=steps
steps=6400,7200
scales=.1,.1
...
```

YOLOv3 hat drei YOLO Layer, in welchem die Anzahl der zu trainierenden Klassen angepasst werden muss. Außerdem muss in dem Convolutional-Layer, welches vor jedem

YOLO-Layer ist, die Filter Anzahl nach : $(classes + 5) * 3$ angepasst werden. In diesem Fall sind es vier Klassen und 27 Filter.

yolov3.cfg

```
[convolutional]
size=1
stride=1
pad=1
filters=27
activation=linear

[yolo]
mask = 6,7,8
anchors = 10,13, 16,30, 33,23, 30,61, 62,45, 59,119, 116,90, 156,198, 373,326
classes=4
num=9
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1
```

In einer *obj.names*-Datei müssen die zu trainierenden Klassen zu finden sein. Eine Klasse pro Zeile. Die Reihenfolge der Klassen spielt hierbei eine Rolle. Sie muss in Korrelation zu den Klassennummierungen der Annotations der Trainingsbilder stehen.

obj.names

```
Door
Person
Table
Chair
```

Als nächstes verweist die Datei `obj.names` auf die Orte der benötigten Dateien.

obj.data

```
classes = 4
train = data/train.txt
valid = data/valid.txt
names = data/obj.names
backup = backup
```

Die oben genannten `train.txt`-Datei und `valid.txt`-Datei sind die Annotation-Dateien der Trainingsdaten. Diese wurden mithilfe eines Scripts generiert. Vorher wurden die Trainingsbilder in Train und Validation in einem zwei zu zehn Verhältnis geteilt. Als Testdaten wird später ein selbst aufgenommenes und Annotiertes Video benutzt.

Der Inhalt der `train.txt`-Datei sowie `valid.txt`-Datei sind die Pfade zu den Bildern relativ zum Darknet Root-Directory. Zum Beispiel: `Data/obj/trainimage_1.jpg`

Nun sollte die Ordnerstruktur des Darknet Repositories aussehen wie auf Abbildung 4.1 zu sehen ist.

Zuletzt werden dem Netz noch Startgewichte mitgegeben, diese werden aus dem Github Repository von AlexeyAB heruntergeladen.

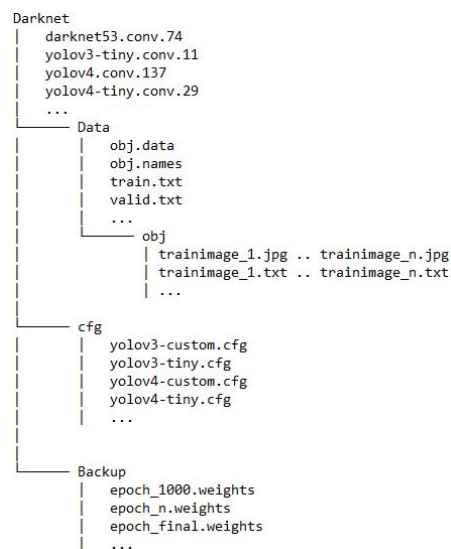


Abbildung 4.1: Ordnerstruktur von Darknet

4.3 Erster Trainingsdurchlauf

Das Training wird gestartet mit:

```
$ ./darknet detector train data/obj.data cfg/yolov3bach.cfg darknet53.conv.74 -map
```

Durch das `-map` Flag wird alle paar Epochen eine Validation durchgeführt und auf einem Diagramm veranschaulicht. Auf diesem Diagramm, welches während des Trainings aktualisiert wird, kann der Loss des Netzes sowie die mean-Average-Precision (mAP) überwacht werden. Nach Iteration 1000 wird erstmals die Validation auf den angegebenen Validationsdaten durchgeführt und der erste mAP-Wert im Diagramm dargestellt. Ab diesem Zeitpunkt wird alle 100 Epochen die Validation durchgeführt.

Das Training läuft auf einer NVIDIA GTX 1660 GPU ungefähr 28 Stunden bei den angegebenen 8000 Epochen für vier zu trainierende Klassen.

Alle 1000 Epochen speichert Darknet die `.weights`-Datei des trainierenden Netzes, sowie die Gewichte, welche die beste mAP erreichten, in einer `_best.weights`-Datei. Es kommt vor, dass das Training des Netzes nicht die angegebene Menge an Epochen erreicht und vorher das Training beendet, wenn zu lange keine nennenswerten Veränderungen im Loss des Netzes aufgetreten sind. Dies soll ein Overfitting auf die Trainingsdaten verhindern. Das Training kann jederzeit von einem der zwischengespeicherten Zeitpunkte weiter trainiert werden, indem die `.weights`-Datei, hier `darknet53.conv.74`, mit der gewünschten `.weights`-Datei ausgetauscht wird. Für diesen Zweck, oder falls das Netz außerplanmäßig abbricht, speichert Darknet alle 1000 Epochen die aktuellen Gewichte unabhängig von mAP oder Loss in einer `_last.weights`-Datei.

Für diese Arbeit wurde das Training für YOLOv3, YOLOv4 sowie die schnelleren, aber auch ungenaueren Versionen YOLOv3-tiny und YOLOv4-tiny durchgeführt.

4.3.1 Auswertung des Trainings

Das Training der Netze zeigt zunächst keine zufriedenstellenden Ergebnisse. Beim Training von YOLOv3 mit vier Klassen wurden für den Großteil der Klassen nur ca. 20% mAP erreicht (Siehe Abbildung 4.2). Dies ist nicht ausreichend für eine Anwendung auf dem Jetson Nano, auf welchem wenig Frames-Per-Second (FPS) zu erwarten sind, aufgrund der Leistungsschwächeren GPU.

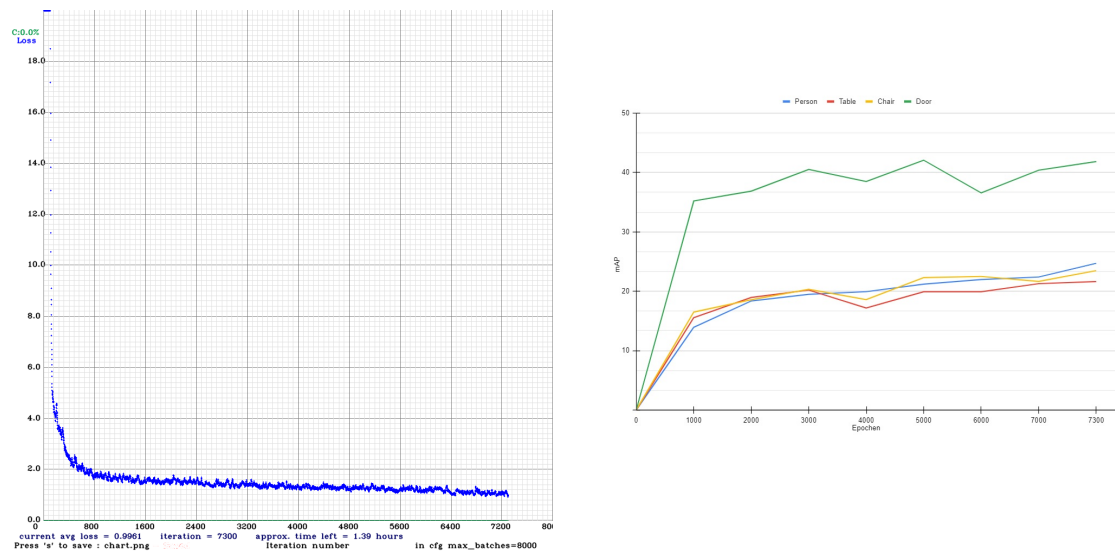


Abbildung 4.2: Loss- und mAP-Graph des Trainings mit vier Klassen

Um eine fehlerhafte Installation des Geräts und Frameworks auszuschließen, wurde YOLOv3 mit einer Änderung erneut trainiert. Es wurde nur eine Klasse trainiert, damit oft zusammen auftretende Objekte im Training nicht zu einer Verwirrung des Netzes führen. Die gewählte Klasse war *Person*. Auch hier waren jedoch niedrige mAP Werte zu sehen (Siehe Abbildung 4.3).

Während des Trainings ist die Lernrate des Netzes zu Anfang sehr groß, weswegen schnell ein Trend zu sehen ist. Da beim erneuten Training des Netzes schnell zu erkennen war, dass auch hier die mAP im Bereich von 40% liegt, wurde das Training abgebrochen und erneut trainiert, aber dieses mal auf einem kleineren Datensatz. Bei einem Training mit wenig Daten ist zu erwarten, dass das Netz schnell auf den wenigen Daten konvergiert, eventuell overfitted und hohe mAP-Werte aufweist. Das erneute Training erzielte jedoch erneut ähnliche Ergebnisse, bei einem Training mit 250 anstatt 2000 Bildern.

Um nun genaueres zu erfahren, wird das Netz einem manuellen Validationstest unterzogen. Hierzu wurde die erhaltene `_best.weights`-Datei auf dem 44 Bild großen Validationsdatensatz, vom Training mit 250 Bildern, getestet (Siehe Abbildung 4.4):

```
$ ./darknet detector map data/obj.data cfg/yolov3bach250.cfg yolov3bach250_best.weights
```

Das Netz zeigt einen sehr hohen Wert an false-negative-Erkennungen (FN) und einen niedrigen Wert and true-positive-Erkennungen (TP). Es erkennt nicht die Menge an

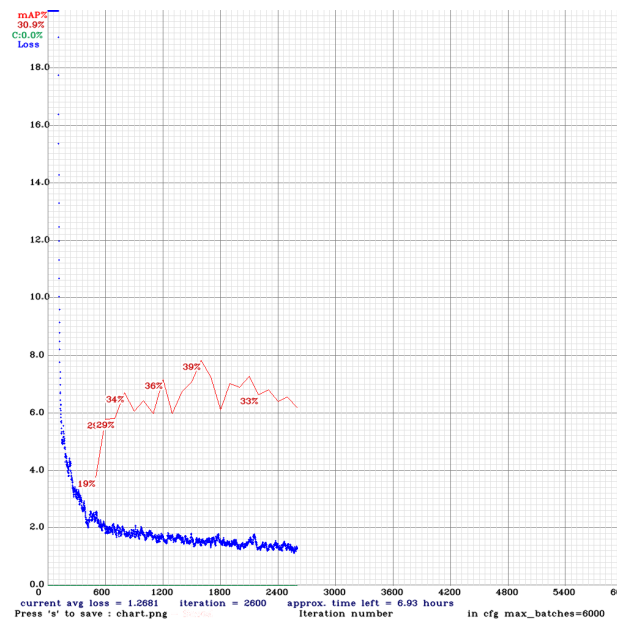


Abbildung 4.3: Loss- und mAP-Graph des Trainings mit einer Klasse

```

calculation mAP (mean average precision)...
Detection layer: 82 - type = 28
Detection layer: 94 - type = 28
Detection layer: 106 - type = 28
44
detections_count = 656, unique_truth_count = 255
class_id = 0, name = Person, ap = 33.18% (TP = 77, FP = 48)
for conf_thresh = 0.25, precision = 0.62, recall = 0.30, F1-score = 0.41
for conf_thresh = 0.25, TP = 77, FP = 48, FN = 178, average IoU = 41.83 %
IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
mean average precision (mAP@0.50) = 0.331759, or 33.18 %
Total Detection Time: 2 Seconds
    
```

Abbildung 4.4: mAP-Berechnung des Trainings mit wenig Trainingsbildern

Objekten, welche es eigentlich erkennen sollte, weswegen auch der Recall-Wert niedrig ausfällt, da oftmals kein Objekt erkannt wird, wo eines sein sollte.

Auf diese Weise wurde auch das, zuvor auf 2000 Bildern trainierte, Netz getestet. Hier sieht man einen deutlich höheren Recall-Wert, sowie mehr TPs und weniger FNs. Die Menge an FPs veränderte sich kaum (Siehe Abbildung 4.5). Der erhöhte Wert an TPs war zu erwarten, da mit einer steigenden Anzahl an Trainingsbildern das Netz robuster wird und somit Objekte als solche identifiziert. Dennoch war dies Anlass, die verwendeten Trainings- und Validationsdaten genauer zu überprüfen.

```
calculation mAP (mean average precision)...
Detection layer: 82 - type = 28
Detection layer: 94 - type = 28
Detection layer: 106 - type = 28
44
detections_count = 1388, unique_truth_count = 255
class_id = 0, name = Person, ap = 66.66% (TP = 148, FP = 45)

for conf_thresh = 0.25, precision = 0.77, recall = 0.58, F1-score = 0.66
for conf_thresh = 0.25, TP = 148, FP = 45, FN = 107, average IoU = 55.59 %

IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
mean average precision (mAP@0.50) = 0.666594, or 66.66 %
Total Detection Time: 2 Seconds
```

Abbildung 4.5: mAP-Berechnung des Trainings mit vielen Trainingsbildern

4.4 Fehlerursache

Es wurden einzelne Bilder an das Netz gegeben und die Ergebnisse der Objekterkennung selbst eingeschätzt. Das Ergebnis waren zufriedenstellende Erkennungen bei den meisten überprüften Testbildern. Die Personen wurden als solche identifiziert, jedoch waren die Confidence-Values nicht hoch. Um die Bewertung bei der Validation des Netzes nachvollziehen zu können, wurde mithilfe von YOLO-Mark, einem Programm zum Annotieren von Bildern im YOLO-Format, die Validationsbilder überprüft.

Es war schnell zu erkennen, dass die Validationsbilder, und damit zwangsläufig auch die Trainingsbilder, nicht sauber annotiert wurden. So wurden zum Beispiel Schuhe und Kopfbedeckungen fälschlicherweise als Personen annotiert, sowie abgeschnittene Personen teilweise überhaupt nicht annotiert (Siehe Abbildung 4.6). Das Netz erkannte hier alle Personen, jedoch wurden hier mehrere, als Personen markierte, Objekte nicht erkannt und somit FNs generiert.

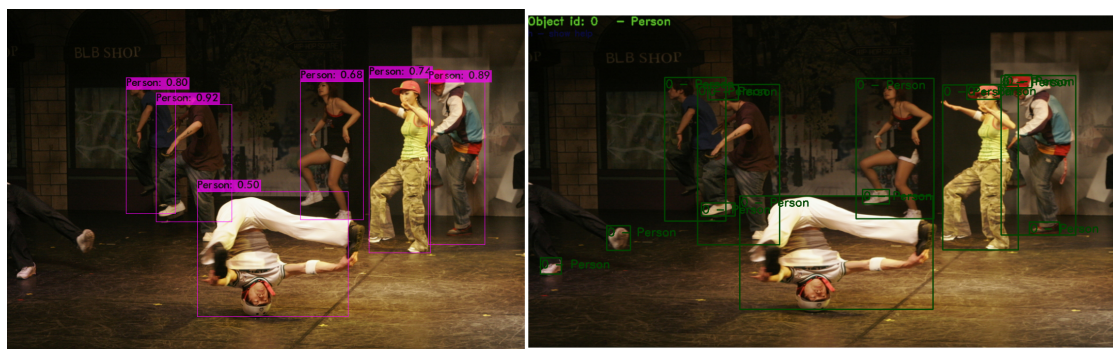


Abbildung 4.6: Beispiel für das Erkennen von false-negatives

Außerdem wurden Gruppen von Menschen in sehr vielen Bildern mit nur einer Bounding Box annotiert. Was dafür sorgt, dass das Netz lernt sowohl Menschen in Gruppen vor-

herzusagen, als auch im einzelnen. Dies steht jedoch im Konflikt miteinander und sorgt im Training für schlechte Ergebnisse. In Abbildung 4.7 kann man erkennen, dass mehr Personen erkannt wurden als überhaupt annotiert wurden. Hier werden FPs generiert.

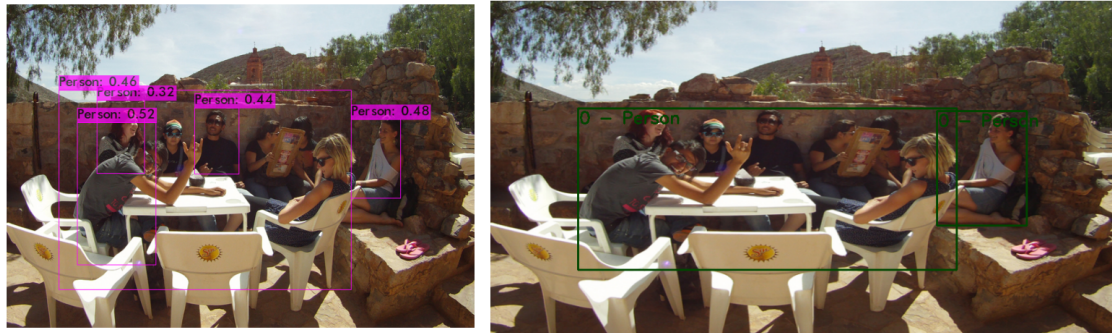


Abbildung 4.7: Beispiel für das Erkennen von false-positives

Aus diesen mäßigen Annotations folgen im Training schlechte Ergebnisse. Die Erkennungen sind bei weiteren Tests mit einzelnen Bildern oftmals nicht zufriedenstellend (Siehe Abbildung 4.8).



Abbildung 4.8: Erkennungsbeispiel

Bei optimalen Bildeigenschaften sind gute Ergebnisse zu erwarten. Jedoch sollte für sehr gute Ergebnisse das Netz mit eigenen, der späteren Anwendung ähnelnden, selbst annotierten Bildern trainiert werden. Deswegen wurden zuerst die Validationdaten korrigiert, indem mittels YOLO-Mark die Annotations angepasst wurden. Ein erneutes validieren des Netzes konnte sofort eine Steigerung in mAP und Recall aufweisen (Siehe Abbildung 4.9). Hierbei wurden lediglich falsche Annotations, sowie Gruppenklassifikationen entfernt (vgl. Abbildung 4.6 und 4.7), als auch fehlende Annotations hinzugefügt.


```

calculation mAP (mean average precision)...
Detection layer: 82 - type = 28
Detection layer: 94 - type = 28
Detection layer: 106 - type = 28
44
detections_count = 1352, unique_truth_count = 204
class_id = 0, name = Person, ap = 73.66% (TP = 148, FP = 42)

for conf_thresh = 0.25, precision = 0.78, recall = 0.73, F1-score = 0.75
for conf_thresh = 0.25, TP = 148, FP = 42, FN = 50, average IoU = 55.82 %

IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
mean average precision (mAP@0.50) = 0.736551, or 73.66 %
Total Detection Time: 2 Seconds
    
```

Abbildung 4.9: mAP nach dem Aufräumen der Annotations

4.5 Training mit verbesserten Daten

Nach erneutem Training eines **YOLOv3** Netzes, bei dem sowohl Trainings- als auch Validationsdaten auf ihre Richtigkeit überprüft wurden, konnten direkt bessere Ergebnisse festgestellt werden. Schon im Training wurden schnell mAP-Werte im Bereich von 60% erreicht (Siehe Abbildung 4.10). Eine Validation im Anschluss zeigte gute Werte in Genauigkeit und Recall. Hierbei wurden, wie auch in den folgenden Trainingsvorgängen, 500 Bilder zum trainieren verwendet.

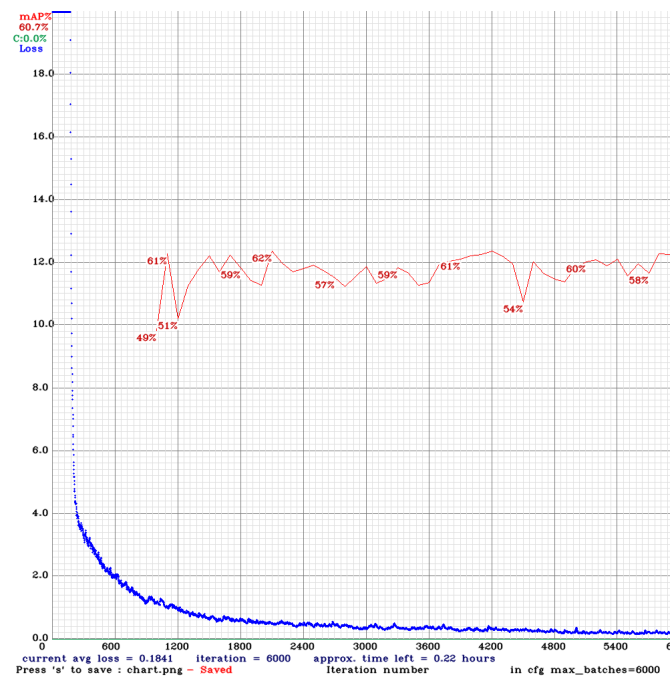


Abbildung 4.10: Training von YOLOv3 auf 500 Bildern

Mit den verbesserten, korrigierten Daten wurde nun auch ein **YOLOv4** Netz trainiert. Durch die in Kapitel 3.2.5 genannten Verbesserungen fürs Training mit YOLOv4 sind bessere mAP Werte zu erwarten. Diese Erwartungen wurden mit dem trainierten Netz erfüllt, welches mAP-Werte um ca 80% (Siehe Abbildung 4.11) erreichte.

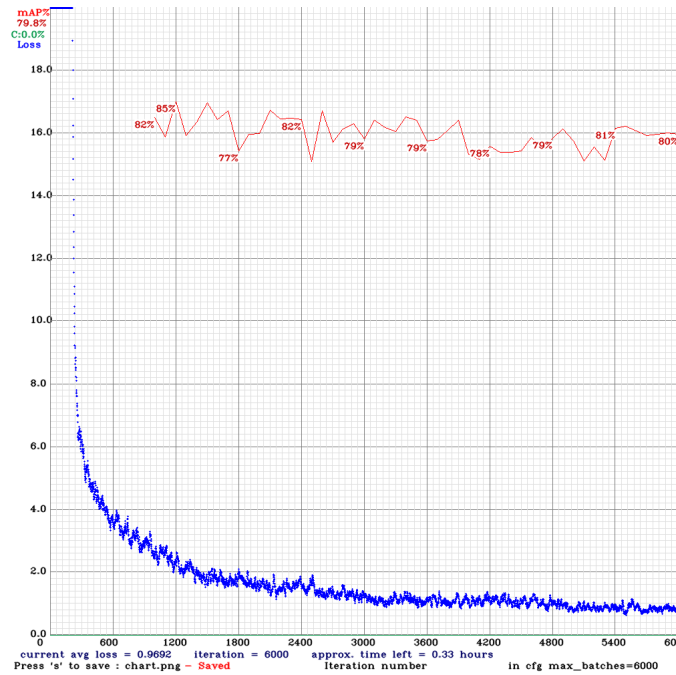


Abbildung 4.11: Training von YOLOv4 auf 500 Bildern

Das Training eines **YOLOv4-tiny** Netzes erreichte mAP-Werte im Bereich von ca. 70% (Siehe Abbildung 4.12).

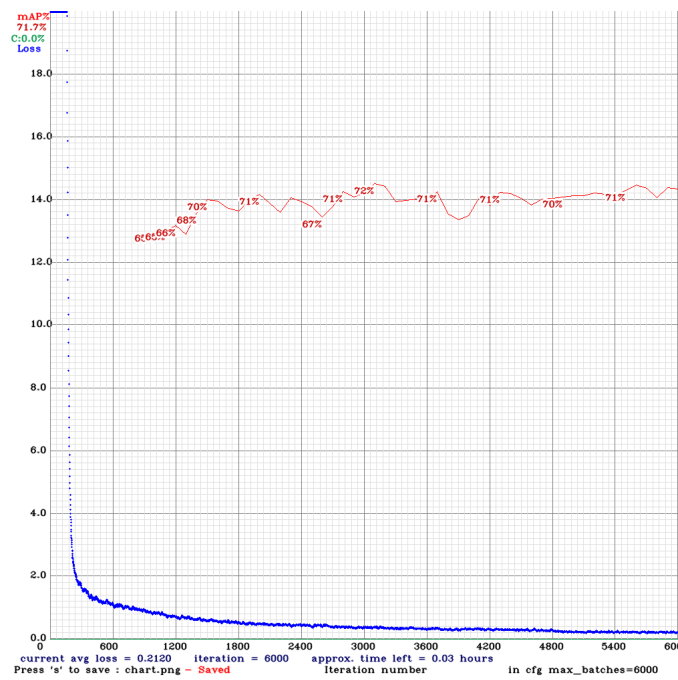


Abbildung 4.12: Training von YOLOv4-tiny auf 500 Bildern

5 Objekterkennung auf dem Jetson Nano

5.1 Vorbereitung des Gerätes

Der Jetson Nano kommt mit einem 3,5V Netzteil, welches für den normalen Gebrauch und den ersten Start des Gerätes ausreicht. Für eine Verwendung über längere Zeiträume unter hoher Last, wie zum Beispiel einer Objekterkennung im Videobetrieb ist ein 5V Netzteil empfohlen. Dieses 5V Netzteil wird erst vom Nano erkannt, sobald der J48-Pin mit einem Jumper überbrückt wurde.

Damit der Jetson Nano im Betrieb nicht zu heiß wird, wurde ein NF-A4x20 5V PWM Lüfter von Noctua angeschlossen. Standardmäßig wird der Lüfter am Jetson Nano nur eingeschaltet, wenn die Temperaturen zu hoch werden. Wird YOLO mittels Videobetrieb für längere Zeit benutzt, so ist ein Lüfter empfohlen, um Schäden an der Hardware zu vermeiden.

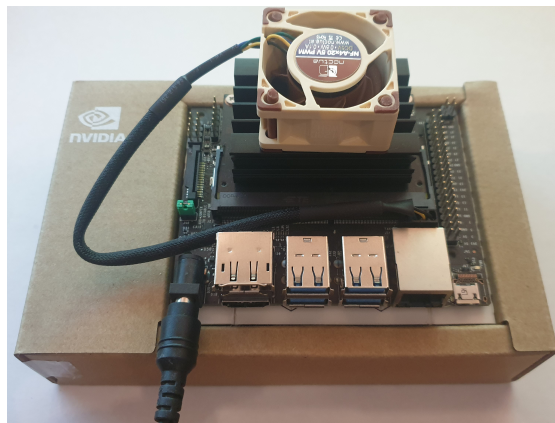


Abbildung 5.1: Aufbau des NVIDIA Jetson Nano

Der Jetson Nano wird mit einer 32GB Micro SD Karte ausgestattet und nach Anleitung von NVIDIA aufgesetzt. Verwendet wird die JetPack Version 4.5.1 und das Aufsetzen von Darknet geschieht analog zur Anleitung auf dem Computer (Siehe Anhang A.0.1).

5.2 Performanceanalyse

Um die Performance der unterschiedlichen, trainierten Netze zu analysieren, wurden diese auf ihre Interferenzgeschwindigkeit im Videobetrieb auf dem Jetson Nano geprüft. Die untersuchten Netze waren YOLOv3, YOLOv3-tiny, YOLOv4 und YOLOv4-tiny. Für die Geschwindigkeitsmessung wurde, mittels detector demo von Darknet, die FPS über einen längeren Zeitraum beobachtet. Dabei werden die durchschnittlichen FPS über die Gesamtdauer eines Videos ausgegeben. Die verwendete Logitech Kamera liefert maximal 30 FPS und eine Input-Größe von 1280×960 Pixel. Anzumerken ist, dass sich die Daten durch den Live-Betrieb von Versuch zu Versuch leicht unterscheiden und sich somit eventuell Abweichungen in den Ergebnissen ergeben können.

Auf dem Gerät, welches zum trainieren benutzt wurde, liefen alle Netze mit 30 FPS, aufgrund der limitierten Bildrate durch die Kamera. Im Nachfolgenden sind die Ergebnisse der Geschwindigkeitsmessung, der verschiedenen YOLO Versionen, auf dem Jetson Nano zu sehen.

CNN	YOLOv3:	YOLOv4:	YOLOv3-tiny:	YOLOv4-tiny:
FPS	1,1	2,8	5,2	7,5

Tabelle 5.1: Performance der YOLO-Netze mit Netzinput von 416×416 Pixel

In Tabelle 5.1 ist zu sehen, dass mit dem YOLOv3 durchschnittlich 1,1 FPS erreicht werden. YOLOv4 hingegen erreicht 2,8 FPS. Wie in Abbildung 3.23 zu erkennen ist, sollten YOLOv4 und -v3 ähnlich schnell sein. Die Messungen können dies nicht bestätigen. Dies könnte unterschiedliche Gründe haben, wie zum Beispiel die leicht unterschiedlichen Videodaten. Für YOLOv3-tiny wurden 5,2 FPS erreicht. YOLOv4-tiny erreicht sogar 7,5 FPS. Auch hier ist wieder zu sehen, dass YOLOv4 schneller ist als YOLOv3.

Mit den in Kapitel 4 erzielten mAP-Werten des YOLOv4-tiny Netzes wäre eine verlässliche Erkennung von Hindernissen, für zum Beispiel einen Segway Loomo Roboter, im Einsatz in Innenräumen von Gebäuden möglich.

5.3 Ausblick

Für eine Steigerung der FPS, kann das Netz auch mit einem kleineren Input betrieben werden. Dies ist aber aufgrund der stark abnehmenden Genauigkeit nicht zu empfehlen.

Der Jetson Nano ist für seinen Preis ein effizientes Gerät. Nachfolgende Jetson Module erzielen, neben höherem Anschaffungskosten, auch höhere FPS Werte und sind mit ihrer fortschreitenden Optimierung für neuronale Netze vielen herkömmlichen GPU's überlegen. Innerhalb der Jetson-Serie wurde die Rechenleistung im Bereich der KI-Performance um ein vielfaches gesteigert. Mit dem **Jetson Xavier NX** wurden Deep-Learning-Accellerator eingeführt, welche mithilfe von auslagern bestimmter Operationen auf dafür optimierte Komponenten die Performance in Bezug auf Deep-Learning und Inferenzzeiten steigern (Siehe Abbildung 5.2).

	Jetson Nano	Jetson TX2 Series	Jetson Xavier NX	Jetson AGX Xavier Series
AI Performance	472 GFLOPs	1.33 TFLOPs*	21 TOPs	32 TOPs*
GPU	128-core NVIDIA Maxwell™ GPU	256-core NVIDIA Pascal™ GPU	384-core NVIDIA Volta™ GPU with 48 Tensor Cores	512-core NVIDIA Volta™ GPU with 64 Tensor Cores
CPU	Quad-Core Arm® Cortex®-A57	Dual-Core Denver 2 64-Bit CPU and Quad-Core Arm® Cortex®-A57	6-core NVIDIA Carmel Arm®v8.2 64-bit CPU 6MB L2 + 4MB L3	8-core NVIDIA Carmel Arm®v8.2 64-bit CPU 8MB L2 + 4MB L3
DL Accelerator	-	-	2x NVDLA	2x NVDLA
Vision Accelerator	-	-	7-Way VLIW Vision Processor	2x 7-Way VLIW Vision Processor
Memory	4 GB 64-bit LPDDR4 (25.6GB/s)	4-8 GB 128-bit LPDDR4 (25.6 - 59.7GB/s)	4-8 GB 128-bit LPDDR4 (59.7GB/s)	32 GB 256-bit LPDDR4x (136.5GB/s)

Tabelle 5.2: Fortschritt der Jetson-Reihe. *Innerhalb der Serie Abweichungen.
(Quelle: <https://developer.nvidia.com/embedded/jetson-modules>)

Eine weitere Option ist die Verwendung von **DeepStream**. Dies ist ein Streaming-Analyse-Toolkit zum erstellen von KI-Applikationen. DeepStream kann mithilfe seiner optimierten Graph-Architektur zu besseren Inferenzgeschwindigkeiten bei der Verwendung von neuronalen Netzen führen. Mittels verschiedener Hardware beschleunigenden Plugins wird der Gebrauch von neuronalen Netzen optimiert. Eine Variante des YOLOv3 Netzes erreicht mittels DeepStream bis zu 11 FPS, der Jetson AGX Xavier erreicht sogar bis zu 223 FPS(Siehe Abbildung 5.3).

Model Arch	Inference Resolution	Precision	Jetson Nano	Jetson Xavier NX		Jetson AGX Xavier			
			GPU (FPS)	GPU (FPS)	DLA1 (FPS)	DLA2 (FPS)	GPU (FPS)	DLA1 (FPS)	DLA2 (FPS)
YoloV3 - ResNet18	906 × 544	INT8	11	78	55	55	223	84	84

Tabelle 5.3: Ausschnitt von Jetson Benchmarks mit Deepstream
(Quelle: <https://docs.nvidia.com/metropolis/deepstream/>)

Desweiteren ist **TensorRT** ein oft benutztes Werkzeug. TensorRT optimiert neuronale Netze, indem es zum Beispiel verschiedene Layer kombiniert oder die Kernel Auswahl optimiert. Nachdem ein Netz trainiert wurde, ermöglicht TensorRT das Netz komprimiert

und optimiert ohne Framework-Overhead auszuführen. Hiermit kann YOLOv3-tiny FPS-Werte von 48 FPS erreichen. Auf dem Jetson AGX Xavier sind über 1000 FPS möglich (Siehe Abbildung 5.4).

Model	Jetson Nano		Jetson TX2 Series		Jetson Xavier NX		Jetson AGX Xavier	
	FPS (limited-Latency)	FPS (max-throughput)	FPS (limited-Latency)	FPS (max-throughput)	FPS (limited-Latency)	FPS (max-throughput)	FPS (limited-Latency)	FPS (max-throughput)
YOLOv3-tiny (416 × 416)	48	49	107	112	607	618	1100	1127

Tabelle 5.4: Ausschnitt von Jetson Benchmarks mit TensorRT
(Quelle: <https://docs.nvidia.com/deeplearning/tensorrt/>)

6 Auswertung im Bezug auf die Aufgabenstellung

6.1 Diskussion der Ergebnisse

Die Trainingsbilder und Annotations die in dieser Arbeit verwendet wurden, wiesen große Mängel auf. Die im Kapitel 3.1 vorgestellten Datensätze bieten einen leichten Zugang zu großen Mengen an Daten aus allen Bereichen und ermöglichen ein Training von robusten neuronalen Netzen. Jedoch ist es sinnvoll die Daten selber zu annotieren, um spezifisch auf die geplante Anwendung zugeschnittene Daten zu erhalten. Zusätzliche Bilder aus der geplanten Einsatzumgebung der Anwendung würden die Genauigkeit der Objekterkennung stark erhöhen.

Die Performance eines CNNs steigt mit der Anzahl der Trainingsdaten und der Trainingszeit. Das Training verlief hinsichtlich der zur Verfügung stehenden Daten erfolgreich und erzielte den Erwartungen entsprechend ausreichende Ergebnisse für die jeweiligen Versionen von YOLO. Speziell für die zukünftig geplante Anwendung mit einem Segway Loomo Robot ist nicht zwangsläufig ein Objekterkennungssystem mit hoher Genauigkeit nötig. Es muss lediglich zuverlässig ein Hindernis erkannt werden und eine Orientierung innerhalb der Räumlichkeiten ermöglicht werden.

Die, aufgrund von Zeitmangel, auf weniger Daten als empfohlen trainierten CNNs, sind mit mAP-Werten von 60% für YOLOv3, 80% für YOLOv4 und 70% für YOLOv4-tiny im Bereich des erwarteten und zufriedenstellend. Mit 7,5 FPS auf dem Jetson Nano ist das YOLOv4-tiny Netz theoretisch schon verwendbar.

Die Geschwindigkeit kann nichtsdestotrotz mit den in Kapitel 5 genannten Optimierungen verbessert werden.

6.2 Vergleich mit anderen Lösungen

Im Bereich der Unterstützung für Bewegungs- und Sichteingeschränkten Personen ist Interesse vorhanden. Mittels KI und Objekterkennung ist es wünschenswert zum Beispiel Rollstühlen zu ermöglichen, Personen selbstständig an gewünschte Orte innerhalb von Gebäuden zu führen.

So zeigt zum Beispiel die Arbeit von Grewal, H et. al [16] die Wichtigkeit von Systemen auf, welche in unbekanntem Indoor-Umgebungen als Unterstützung für Bewegungs- und Sichteingeschränkte Personen dienen.

In einer Veröffentlichung von Yang, X et. al [48] wird ein Indoor-Objekterkennungssystem als Unterstützung für Blinde Personen vorgestellt. Es werden unter anderem nötige Erkennungsklassen vorgestellt und eine relative Positionsbestimmung anhand der Bilddaten diskutiert.

Im Bereich der selbst-orientierenden und selbst-fahrenden Fahrzeuge sind diverse Projekte zu finden. So werden zum Beispiel beim jährlichen Carolo-Cup der Technischen Universität Braunschweig, Miniaturautos mit Systemen ausgestattet, welche eine Erkennung von Straßenschildern und Hindernissen ermöglichen, damit das Miniaturauto gefahrlos selbstständig sein Ziel erreichen kann.

6.3 Future Work

In Zukunft kann ein Fortschritt im Bereich der autonomen Fortbewegung mittels günstiger Hardware das Leben von Bewegungs- und Sichteingeschränkten Personen drastisch erleichtern. Für den Verbraucher ohne Vorkenntnisse in diesem Bereich, sind eine einfache Zugänglichkeit und bessere Anpassbarkeit solcher Anwendungen von Interesse.

Speziell aufbauend auf dieser Arbeit wäre ein Auslagern der Objekterkennung auf ein zentrales System in Betracht zu ziehen. Dies könnte die Objekterkennung und Orientierung eines Loomo Segway Robots stark verbessern, würde jedoch den autonomen Aspekt einschränken. Dies sollte in Zukunft in Betracht gezogen werden.

Literaturverzeichnis

- [1] ALEXEY, Gordeev ; KLYACHIN, Vladimir ; ELDAR, Kurbanov ; DRIABA, Aleksandr: Autonomous Mobile Robot with AI Based on Jetson Nano. In: *Proceedings of the Future Technologies Conference* Springer (Veranst.), 2020, S. 190–204
- [2] BOCHKOVSKIY, Alexey ; WANG, Chien-Yao ; LIAO, Hong-Yuan M.: YOLOv4: Optimal Speed and Accuracy of Object Detection. In: *CoRR* abs/2004.10934 (2020). – URL <https://arxiv.org/abs/2004.10934>
- [3] BÖRCS, Attila ; NAGY, Balázs ; BENEDEK, Csaba: Instant Object Detection in Lidar Point Clouds. In: *IEEE Geoscience and Remote Sensing Letters* 14 (2017), Nr. 7, S. 992–996
- [4] CHOLLET, Francois: *Deep learning with Python*. Simon and Schuster, 2017
- [5] DALAL, N. ; TRIGGS, B.: Histograms of oriented gradients for human detection. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* Bd. 1, 2005, S. 886–893 vol. 1
- [6] DENG, Jia ; DONG, Wei ; SOCHER, Richard ; LI, Li-Jia ; LI, Kai ; FEI-FEI, Li: ImageNet: A large-scale hierarchical image database. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition* IEEE Computer Society (Veranst.), 2009, S. 248–255
- [7] DIONISI, Alessandro ; SARDINI, Emilio ; SERPELLONI, Mauro: Wearable object detection system for the blind. In: *2012 IEEE International Instrumentation and Measurement Technology Conference Proceedings*, 2012, S. 1255–1258
- [8] ERHAN, Dumitru ; SZEGEDY, Christian ; TOSHEV, Alexander ; ANGUELOV, Dragomir: Scalable Object Detection using Deep Neural Networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014

- [9] EVERINGHAM, Mark ; ESLAMI, SM A. ; VAN GOOL, Luc ; WILLIAMS, Christopher K. ; WINN, John ; ZISSERMAN, Andrew: The Pascal Visual Object Classes Challenge—a Retrospective. (2014)
- [10] EVERINGHAM, Mark ; VAN GOOL, Luc ; WILLIAMS, Christopher ; WINN, John ; ZISSERMAN, Andrew: The Pascal Visual Object Classes (VOC) challenge. In: *International Journal of Computer Vision* 88 (2010), 06, S. 303–338
- [11] EVERINGHAM, Mark ; WINN, John: The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Development Kit. (2012)
- [12] EVERINGHAM, Mark ; ZISSERMAN, Andrew ; WILLIAMS, Christopher K. I. ; VAN GOOL, Luc ; ALLAN, Moray ; BISHOP, Christopher M. ; CHAPELLE, Olivier ; DALAL, Navneet ; DESELAERS, Thomas ; DORKÓ, Gyuri ; DUFFNER, Stefan ; EICHORN, Jan ; FARQUHAR, Jason D. R. ; FRITZ, Mario ; GARCIA, Christophe ; GRIFFITHS, Tom ; JURIE, Frederic ; KEYSERS, Daniel ; KOSKELA, Markus ; LAAKSONEN, Jorma ; LARLUS, Diane ; LEIBE, Bastian ; MENG, Hongying ; NEY, Hermann ; SCHIELE, Bernt ; SCHMID, Cordelia ; SEEMANN, Edgar ; SHAWE-TAYLOR, John ; STORKEY, Amos ; SZEDMAK, Sandor ; TRIGGS, Bill ; ULUSOY, Ilkay ; VIITANIEMI, Ville ; ZHANG, Jianguo: The 2005 PASCAL Visual Object Classes Challenge. In: QUIÑONERO-CANDELA, Joaquin (Hrsg.) ; DAGAN, Ido (Hrsg.) ; MAGNINI, Bernardo (Hrsg.) ; BUC, Florence d’Alché (Hrsg.): *Machine Learning Challenges. Evaluating Predictive Uncertainty, Visual Object Classification, and Recognising Tectual Entailment*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, S. 117–176. – ISBN 978-3-540-33428-6
- [13] FELZENSZWALB, P. F. ; GIRSHICK, R. B. ; MCALLESTER, D. ; RAMANAN, D.: Object Detection with Discriminatively Trained Part-Based Models. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32 (2010), Nr. 9, S. 1627–1645
- [14] GHIASI, Golnaz ; LIN, Tsung-Yi ; LE, Quoc V.: DropBlock: A regularization method for convolutional networks. In: *CoRR* abs/1810.12890 (2018). – URL <http://arxiv.org/abs/1810.12890>
- [15] GIRSHICK, Ross ; DONAHUE, Jeff ; DARRELL, Trevor ; MALIK, Jitendra: Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014

- [16] GREWAL, Harkishan S. ; JAYAPRAKASH, Neha T. ; MATTHEWS, Aaron ; SHRIVASTAV, Chinmay ; GEORGE, Kiran: Autonomous wheelchair navigation in unmapped indoor environments. In: *2018 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)* IEEE (Veranst.), 2018, S. 1–6
- [17] HE, Kaiming ; ZHANG, Xiangyu ; REN, Shaoqing ; SUN, Jian: Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. In: *CoRR* abs/1406.4729 (2014). – URL <http://arxiv.org/abs/1406.4729>
- [18] HE, Kaiming ; ZHANG, Xiangyu ; REN, Shaoqing ; SUN, Jian: Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, S. 770–778
- [19] HUSSIN, R. ; JUHARI, M. R. ; KANG, Ng W. ; ISMAIL, R.C. ; KAMARUDIN, A.: Digital Image Processing Techniques for Object Detection From Complex Background Image. In: *Procedia Engineering* 41 (2012), S. 340–344. – URL <https://www.sciencedirect.com/science/article/pii/S1877705812025684>. – International Symposium on Robotics and Intelligent Sensors 2012 (IRIS 2012). – ISSN 1877-7058
- [20] JALLED, Fares ; VORONKOV, Iliia: Object Detection using Image Processing. In: *CoRR* abs/1611.07791 (2016). – URL <http://arxiv.org/abs/1611.07791>
- [21] JIAO, Licheng ; ZHANG, Fan ; LIU, Fang ; YANG, Shuyuan ; LI, Lingling ; FENG, Zhixi ; QU, Rong: A Survey of Deep Learning-Based Object Detection. In: *IEEE Access* 7 (2019), S. 128837–128868. – URL <http://dx.doi.org/10.1109/ACCESS.2019.2939201>. – ISSN 2169-3536
- [22] KRASIN, Ivan ; DUERIG, Tom ; ALLDRIN, Neil ; FERRARI, Vittorio ; ABU-EL-HAJA, Sami ; KUZNETSOVA, Alina ; ROM, Hassan ; UIJLINGS, Jasper ; POPOV, Stefan ; KAMALI, Shahab ; MALLOCI, Matteo ; PONT-TUSET, Jordi ; VEIT, Andreas ; BELONGIE, Serge ; GOMES, Victor ; GUPTA, Abhinav ; SUN, Chen ; CHECHIK, Gal ; CAI, David ; FENG, Zheyun ; NARAYANAN, Dhyanesh ; MURPHY, Kevin: *OpenImages: A public dataset for large-scale multi-label and multi-class image classification*. 2017. – URL <https://storage.googleapis.com/openimages/web/index.html>. – Zugriffsdatum: 2021-11-07
- [23] KRIZHEVSKY, Alex ; SUTSKEVER, Ilya ; HINTON, Geoffrey E.: Imagenet classification with deep convolutional neural networks. In: *Advances in neural information processing systems* 25 (2012), S. 1097–1105

- [24] KUZNETSOVA, Alina ; ROM, Hassan ; ALLDRIN, Neil ; UIJLINGS, Jasper ; KRASIN, Ivan ; PONT-TUSET, Jordi ; KAMALI, Shahab ; POPOV, Stefan ; MALLOCI, Matteo ; KOLESNIKOV, Alexander ; AL. et: The Open Images Dataset V4. In: *International Journal of Computer Vision* 128 (2020), Mar, Nr. 7, S. 1956–1981. – URL <http://dx.doi.org/10.1007/s11263-020-01316-z>. – ISSN 1573-1405
- [25] LIN, Tsung-Yi ; MAIRE, Michael ; BELONGIE, Serge ; HAYS, James ; PERONA, Pietro ; RAMANAN, Deva ; DOLLÁR, Piotr ; ZITNICK, C L.: Microsoft coco: Common objects in context. In: *European conference on computer vision* Springer (Veranst.), 2014, S. 740–755
- [26] LIN, Tsung-Yi ; MAIRE, Michael ; BELONGIE, Serge ; HAYS, James ; PERONA, Pietro ; RAMANAN, Deva ; DOLLÁR, Piotr ; ZITNICK, C L.: *COCO - Common Objects in Context*. 2021. – URL <https://cocodataset.org/#explore>. – Zugriffsdatum: 2021-06-04
- [27] LIU, Danqing: *A Practical Guide to ReLU*. 2017. – URL <https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7>. – Zugriffsdatum: 2021-19-08
- [28] LIU, Shu ; QI, Lu ; QIN, Haifang ; SHI, Jianping ; JIA, Jiaya: Path Aggregation Network for Instance Segmentation. In: *CoRR* abs/1803.01534 (2018). – URL <http://arxiv.org/abs/1803.01534>
- [29] LOSHCHILOV, Ilya ; HUTTER, Frank: SGDR: Stochastic Gradient Descent with Restarts. In: *CoRR* abs/1608.03983 (2016). – URL <http://arxiv.org/abs/1608.03983>
- [30] MISRA, Diganta: Mish: A Self Regularized Non-Monotonic Neural Activation Function. In: *CoRR* abs/1908.08681 (2019). – URL <http://arxiv.org/abs/1908.08681>
- [31] MNIH, Volodymyr ; KAVUKCUOGLU, Koray ; SILVER, David ; GRAVES, Alex ; ANTONOPOULOU, Ioannis ; WIERSTRA, Daan ; RIEDMILLER, Martin A.: Playing Atari with Deep Reinforcement Learning. In: *CoRR* abs/1312.5602 (2013). – URL <http://arxiv.org/abs/1312.5602>
- [32] OZTURK, Tulin ; TALO, Muhammed ; YILDIRIM, Eylul A. ; BALOGLU, Ulas B. ; YILDIRIM, Ozal ; RAJENDRA ACHARYA, U.: Automated detection of COVID-19 cases using deep neural networks with X-ray images. In: *Computers in Biology and*

- Medicine* 121 (2020), S. 103792. – URL <https://www.sciencedirect.com/science/article/pii/S0010482520301621>. – ISSN 0010-4825
- [33] PATEL, Chirag ; SHAH, Dipti ; PATEL, Atul: Automatic number plate recognition system (anpr): A survey. In: *International Journal of Computer Applications* 69 (2013), Nr. 9
- [34] REDMON, Joseph ; DIVVALA, Santosh ; GIRSHICK, Ross ; FARHADI, Ali: You Only Look Once: Unified, Real-Time Object Detection. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016
- [35] REDMON, Joseph ; FARHADI, Ali: YOLO9000: Better, Faster, Stronger. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017
- [36] REDMON, Joseph ; FARHADI, Ali: YOLOv3: An Incremental Improvement. In: *CoRR* abs/1804.02767 (2018). – URL <http://arxiv.org/abs/1804.02767>
- [37] RUMELHART, David E. ; HINTON, Geoffrey E. ; WILLIAMS, Ronald J.: Learning representations by back-propagating errors. In: *nature* 323 (1986), Nr. 6088, S. 533–536
- [38] RUSSAKOVSKY, Olga ; DENG, Jia ; SU, Hao ; KRAUSE, Jonathan ; SATHEESH, Sanjeev ; MA, Sean ; HUANG, Zhiheng ; KARPATHY, Andrej ; KHOSLA, Aditya ; BERNSTEIN, Michael u. a.: ImageNet Large Scale Visual Recognition Challenge. (2014)
- [39] RUSSAKOVSKY, Olga ; DENG, Jia ; SU, Hao ; KRAUSE, Jonathan ; SATHEESH, Sanjeev ; MA, Sean ; HUANG, Zhiheng ; KARPATHY, Andrej ; KHOSLA, Aditya ; BERNSTEIN, Michael ; BERG, Alexander C. ; FEI-FEI, Li: ImageNet Large Scale Visual Recognition Challenge. In: *International Journal of Computer Vision (IJCV)* 115 (2015), Nr. 3, S. 211–252
- [40] SHARMA, Sagar ; SHARMA, Simone: Activation functions in neural networks. In: *Towards Data Science* 6 (2017), Nr. 12, S. 310–316
- [41] SIMONYAN, Karen ; ZISSERMAN, Andrew: Very deep convolutional networks for large-scale image recognition. In: *arXiv preprint arXiv:1409.1556* (2014)
- [42] SZEGEDY, Christian ; LIU, Wei ; JIA, Yangqing ; SERMANET, Pierre ; REED, Scott ; ANGUELOV, Dragomir ; ERHAN, Dumitru ; VANHOUCKE, Vincent ; RABINOVICH,

- Andrew: Going deeper with convolutions. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, S. 1–9
- [43] SZEGEDY, Christian ; VANHOUCKE, Vincent ; IOFFE, Sergey ; SHLENS, Jonathon ; WOJNA, Zbigniew: Rethinking the Inception Architecture for Computer Vision. In: *CoRR* abs/1512.00567 (2015). – URL <http://arxiv.org/abs/1512.00567>
- [44] TAN, Mingxing ; LE, Quoc V.: EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In: *CoRR* abs/1905.11946 (2019). – URL <http://arxiv.org/abs/1905.11946>
- [45] UIJLINGS, Jasper ; SANDE, K. ; GEVERS, T. ; SMEULDERS, A.W.M.: Selective Search for Object Recognition. In: *International Journal of Computer Vision* 104 (2013), 09, S. 154–171
- [46] WANG, Chien-Yao ; LIAO, Hong-Yuan M. ; YEH, I-Hau ; WU, Yueh-Hua ; CHEN, Ping-Yang ; HSIEH, Jun-Wei: CSPNet: A New Backbone that can Enhance Learning Capability of CNN. In: *CoRR* abs/1911.11929 (2019). – URL <http://arxiv.org/abs/1911.11929>
- [47] WOO, Sanghyun ; PARK, Jongchan ; LEE, Joon-Young ; KWEON, In S.: CBAM: Convolutional Block Attention Module. In: *CoRR* abs/1807.06521 (2018). – URL <http://arxiv.org/abs/1807.06521>
- [48] YANG, Xiaodong ; TIAN, YingLi ; YI, Chucai ; ARDITI, Aries: Context-Based Indoor Object Detection as an Aid to Blind Persons Accessing Unfamiliar Environments. In: *Proceedings of the 18th ACM International Conference on Multimedia*. New York, NY, USA : Association for Computing Machinery, 2010 (MM '10), S. 1087–1090. – URL <https://doi.org/10.1145/1873951.1874156>. – ISBN 9781605589336
- [49] YUN, Sangdoon ; HAN, Dongyoon ; OH, Seong J. ; CHUN, Sanghyuk ; CHOE, Junsuk ; YOO, Youngjoon: CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features. In: *CoRR* abs/1905.04899 (2019). – URL <http://arxiv.org/abs/1905.04899>
- [50] ZHAN, Chaohui ; DUAN, Xiaohui ; XU, Shuoyu ; SONG, Zheng ; LUO, Min: An Improved Moving Object Detection Algorithm Based on Frame Difference and Edge Detection. In: *Fourth International Conference on Image and Graphics (ICIG 2007)*, 2007, S. 519–523

- [51] ZHENG, Zhaohui ; WANG, Ping ; LIU, Wei ; LI, Jinze ; YE, Rongguang ; REN, Dongwei: Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34 (2020), Apr., Nr. 07, S. 12993–13000. – URL <https://ojs.aaai.org/index.php/AAAI/article/view/6999>
- [52] ZHOU, Wangchunshu ; GE, Tao ; XU, Ke ; WEI, Furu ; ZHOU, Ming: Self-Adversarial Learning with Comparative Discrimination for Text Generation. In: *CoRR* abs/2001.11691 (2020). – URL <https://arxiv.org/abs/2001.11691>
- [53] ZHU, Yingying ; ZHANG, Chengquan ; ZHOU, Duoyou ; WANG, Xinggang ; BAI, Xiang ; LIU, Wenyu: Traffic sign detection and recognition using fully convolutional network guided proposals. In: *Neurocomputing* 214 (2016), S. 758–766. – URL <https://www.sciencedirect.com/science/article/pii/S092523121630741X>. – ISSN 0925-2312
- [54] ZOU, Zhengxia ; SHI, Zhenwei ; GUO, Yuhong ; YE, Jieping: Object Detection in 20 Years: A Survey. In: *CoRR* abs/1905.05055 (2019). – URL <http://arxiv.org/abs/1905.05055>

A Anhang

A.0.1 Aufsetzen einer Maschine für YOLO

0. System auf den neusten Stand bringen

1. GCC installieren

```
~$ sudo apt install build-essential
```

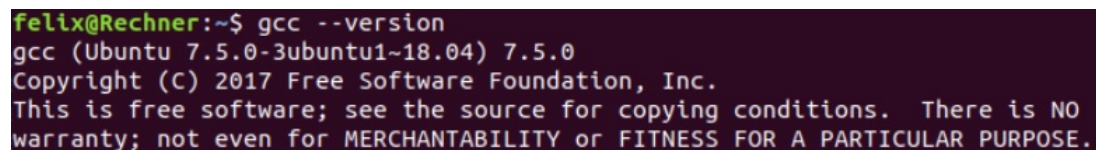
A terminal window with a dark background and light text. The prompt is 'felix@Rechner:~\$'. The command 'gcc --version' has been executed, resulting in the following output: 'gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0', 'Copyright (C) 2017 Free Software Foundation, Inc.', and a disclaimer: 'This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.'

Abbildung A.1: GCC Version

2. CUDA installieren

2.1. Version passend zur GPU auswählen auf folgender Website:

https://de.wikipedia.org/wiki/CUDA#Unterst%C3%BCTzte_GPUs

2.2. Passende Nvidia Treiber installieren:

Anwendungen und Aktualisierungen -> Zusätzliche Treiber -> Neuste

2.3. Computer neu starten

2.4. erfolgreiche Installation prüfen mit:

```
~$ nvidia-smi
```

3. Installiere CUDA version (CUDA-Desktop (GTX1660))

<https://developer.nvidia.com/CUDA-toolkit-archive>

```
~$ wget https://developer.download.nvidia.com/
compute/cuda/11.2.0/local_installers/cuda_11.2.0_460.27.04_linux.run

~$ sudo sh cuda_11.2.0_460.27.04_linux.run
```



```
felix@Rechner:~$ wget https://developer.download.nvidia.com/compute/cuda/11.2.0/local_installers/cuda_11.2.0_460.27.04_linux.run
--2021-01-08 11:51:03-- https://developer.download.nvidia.com/compute/cuda/11.2.0/local_installers/cuda_11.2.0_460.27.04_linux.run
Auflösen des Hostnamens developer.download.nvidia.com (developer.download.nvidia.com) _ 152.199.20.126
Verbindungsaufbau zu developer.download.nvidia.com (developer.download.nvidia.com)[152.199.20.126]:443 _ verbunden.
HTTP-Anforderung gesendet, auf Antwort wird gewartet _ 200 OK
Länge: 3046790184 (2,8G) [application/octet-stream]
Wird in »cuda_11.2.0_460.27.04_linux.run« gespeichert.

cuda_11.2.0_460.27.04_linux.run 100%[=====] 2,84G 25,1MB/s in 1m 56s
2021-01-08 11:52:59 (25,1 MB/s) - »cuda_11.2.0_460.27.04_linux.run« gespeichert [3046790184/3046790184]
```

Abbildung A.2: Download von CUDA



```
felix@Rechner:~$ sudo sh cuda_11.2.0_460.27.04_linux.run
=====
= Summary =
=====
Driver: Not Selected
Toolkit: Installed in /usr/local/cuda-11.2/
Samples: Not Selected

Please make sure that
- PATH includes /usr/local/cuda-11.2/bin
- LD_LIBRARY_PATH includes /usr/local/cuda-11.2/lib64, or, add /usr/local/cuda-11.2/lib64 to /etc/ld.so.conf and run ldconfig as root

To uninstall the CUDA Toolkit, run cuda-uninstaller in /usr/local/cuda-11.2/bin
***WARNING: Incomplete installation! This installation did not install the CUDA Driver. A driver of version at least 460.00 is required for CUDA 11.2 functionality to work.
To install the driver using this installer, run the following command, replacing <cudaInstaller> with the name of this run file:
sudo <cudaInstaller>.run --silent --driver

Logfile is /var/log/cuda-installer.log
```

Abbildung A.3: CUDA Installation. Kein Driver, keine Samples.

4. cuDNN installieren

4.1. Download der richtigen Version

<https://developer.nvidia.com/cudnn> (Account wird benötigt.)

4.2. Datei entpacken:

```
~$ tar -xzvf cudnn-11.1-linux-x64-v8.0.5.39.tgz
```

4.3. An richtige Stelle kopieren:

```
~$ sudo cp cuda/include/cudnn*.h /usr/local/cuda/include
~$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
```

4.4. chmod anpassen:

```
~$ sudo chmod a+r /usr/local/cuda/include/cudnn*.h
~$ /usr/local/cuda/lib64/libcudnn*
```

5. OpenCV installieren

5.0. Vorbereitung:

```
~$ sudo apt install libgtk2.0-dev and pkg-config
~$ sudo apt install git
```

5.1. Repository klonen:

```
~$ mkdir ~/opencv_build && cd ~/opencv_build
~$ git clone https://github.com/opencv/opencv.git
~$ git clone https://github.com/opencv/opencv_contrib.git
```

5.2. Version auswählen:

```
~$ git checkout 3.4
(in beiden Repositories)
```

5.3. Build:

```
~$ cd ~/opencv_build/opencv
~$ mkdir build && cd build
```

5.4. Make:

```
~$ sudo apt install cmake
```

```
~$ cmake -D CMAKE_BUILD_TYPE=RELEASE \  
~$ -D CMAKE_INSTALL_PREFIX=/usr/local \  
~$ -D INSTALL_C_EXAMPLES=ON \  
~$ -D INSTALL_PYTHON_EXAMPLES=ON \  
~$ -D OPENCV_GENERATE_PKGCONFIG=ON \  
~$ -D OPENCV_EXTRA_MODULES_PATH=~/.opencv_build/opencv_contrib/modules \  
~$ -D BUILD_EXAMPLES=ON ..
```

5.5. Compilieren:

```
~$ make -j4
```

(4 ist die Anzahl der Kerne, "nproc" für Anzahl an Kernen)

5.6. Installieren:

```
~$ sudo make install
```

5.7. Verifizieren:

```
~$ pkg-config --modversion opencv
```

6. Darknet installieren:

6.1. Repository klonen

```
~$ git clone https://github.com/AlexeyAB/darknet.git
```

6.1. Vortrainierte Gewichte downloaden:

yolov3 weights <https://pjreddie.com/media/files/yolov3.weights>

yolov3tiny weights <https://pjreddie.com/media/files/yolov3-tiny.weights>

yolov4 weights https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.weights

yolov4tiny weights https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_pre/yolov4-tiny.weights

A.0.2 OIDv4 ToolKit

1.1. Repository klonen

```
~$ git clone https://github.com/EscVM/OIDv4_ToolKit.git
```

1.2. Installation der benötigten Pakete

```
~$ pip3 install -r requirements.txt
```

1.3. Test der Installation

```
~$ python3 main.py
```

1.4. Download der Bilder

```
~$ python3 main.py downloader --classes Door Person Table Chair -  
-type_csv train
```

Das Toolkit erstellt in einem Ordner `./OID/Dataset/train/` einen Ordner Pro gewünschte Klasse und in Diesem Ordner einen weiteren mit den Labels zu den Bildern. Diese Annotationen sind nicht im Darknet Format und müssen noch umgeschrieben werden.

Sie liegen vor im Format : `<object-class> <xMin> <yMin> <xMax> <yMax>`

Und sollen vorliegen als : `<object-class> <x> <y> <width> <height>`

Mithilfe eines Scriptes aus einer Suggestion des Repositories `convert_ annotations.py` werden die bereits erstellten Annotationen gelesen, umgeformt und in den Ordner der Bilder gespeichert.

`convert_ annotations.py:`

```
..  
def convert(filename_str, coords):  
    os.chdir("../")  
    image = cv2.imread(filename_str + ".jpg")  
    coords[2] -= coords[0]  
    coords[3] -= coords[1]  
    x_diff = int(coords[2]/2)  
    y_diff = int(coords[3]/2)  
    coords[0] = coords[0]+x_diff
```

```
coords[1] = coords[1]+y_diff
coords[0] /= int(image.shape[1])
coords[1] /= int(image.shape[0])
coords[2] /= int(image.shape[1])
coords[3] /= int(image.shape[0])
os.chdir("Label")
return coords
..
```

1.5 Download des converter Scripts

https://github.com/EscVM/OIDv4_ToolKit/pull/97/files#diff-23665c637730cd2d0c4

1.6 classes.txt

Hier müssen die Klassen stehen, welche von YOLO trainiert werden sollen. Dies ist der gleiche Inhalt wie obj.names aus Darknet. Es muss die selbe Reihenfolge haben wie später obj.names.

classes.txt

```
Door
Person
Table
Chair
```

1.7 Ausführen des Scripts:

```
~$ python3 convert_annotations.py
```

A.0.3 coco manager

1.1 Download des originalen Scripts.

<https://github.com/immersive-limit/coco-manager/blob/master/filter.py>

1.1 Script mit Erweiterung für Download von Annotationen und erstellen der train.txt

filter.py

```
import json
import urllib.request
import os
#original at https://github.com/immersive-limit/coco-manager

from pathlib import Path

class CocoFilter():
    """ Filters the COCO dataset
    """

    def _process_info(self):
        self.info = self.coco['info']

    def _process_licenses(self):
        self.licenses = self.coco['licenses']

    def _process_categories(self):
        self.categories = dict()
        self.super_categories = dict()
        self.category_set = set()

        for category in self.coco['categories']:
            cat_id = category['id']
            super_category = category['supercategory']

            # Add category to categories dict
            if cat_id not in self.categories:
                self.categories[cat_id] = category
                self.category_set.add(category['name'])
            else:
                print('ERROR: Skipping duplicate category id: {category}')

            # Add category id to the super_categories dict
            if super_category not in self.super_categories:
                self.super_categories[super_category] = {cat_id}
            else:
                self.super_categories[super_category] |= {cat_id} # e.g. {1, 2, 3} |= {4}
                => {1, 2, 3, 4}

    def _process_images(self):
        self.images = dict()
        for image in self.coco['images']:
            image_id = image['id']
            if image_id not in self.images:
```

```
        self.images[image_id] = image
    else:
        print('ERROR: Skipping duplicate image id: {image}')

def _process_segmentations(self):
    self.segmentations = dict()
    for segmentation in self.coco['annotations']:
        image_id = segmentation['image_id']
        if image_id not in self.segmentations:
            self.segmentations[image_id] = []
        self.segmentations[image_id].append(segmentation)

def _filter_categories(self):
    """ Find category ids matching args
        Create mapping from original category id to new category id
        Create new collection of categories
    """
    missing_categories = set(self.filter_categories) - self.category_set
    if len(missing_categories) > 0:
        print('Did not find categories: {missing_categories}')
        should_continue = input('Continue? (y/n) ').lower()
        if should_continue != 'y' and should_continue != 'yes':
            print('Quitting early.')
            quit()

    self.new_category_map = dict()
    new_id = 1
    for key, item in self.categories.items():
        if item['name'] in self.filter_categories:
            self.new_category_map[key] = new_id
            new_id += 1

    self.new_categories = []
    for original_cat_id, new_id in self.new_category_map.items():
        new_category = dict(self.categories[original_cat_id])
        new_category['id'] = new_id
        self.new_categories.append(new_category)

def _filter_annotations(self):
    """ Create new collection of annotations matching category ids
        Keep track of image ids matching annotations
    """
    self.new_segmentations = []
    self.new_image_ids = set()
    for image_id, segmentation_list in self.segmentations.items():
        for segmentation in segmentation_list:
            original_seg_cat = segmentation['category_id']
            if original_seg_cat in self.new_category_map.keys():
                new_segmentation = dict(segmentation)
                new_segmentation['category_id'] =
                    self.new_category_map[original_seg_cat]
```



```
        self.new_segmentations.append(new_segmentation)
        self.new_image_ids.add(image_id)

def _filter_images(self):
    """ Create new collection of images
    """
    self.new_images = []
    for image_id in self.new_image_ids:
        self.new_images.append(self.images[image_id])

def main(self, args):
    # Open
    self.input_json_path = args.input_json
    self.output_json_path = args.output_json
    self.filter_categories = args.categories
    self.outputdir = args.outputdir
    self.traintxt = args.traintxt
    self.amount = args.amount

    if self.amount==0 or None or len(self.amount)!=1:
        self.amount = int(999999)
    else:
        self.amount = int(self.amount[0])

    # Verify input path exists
    if not Path(self.input_json_path).exists():
        print('Input json path not found.')
        print('Quitting early.')
        quit()

    # Verify output path does not already exist
    if Path(self.output_json_path).exists():
        should_continue = input('Output path already exists. Overwrite? (y/n)
        ').lower()
        if should_continue != 'y' and should_continue != 'yes':
            print('Quitting early.')
            quit()

    # Load the json
    print('Loading json file...')
    with open(self.input_json_path) as json_file:
        self.coco = json.load(json_file)

    # Process the json
    print('Processing input json...')
    self._process_info()
    self._process_licenses()
    self._process_categories()
    self._process_images()
    self._process_segmentations()
```

```
# Filter to specific categories
print('Filtering...')
self._filter_categories()
self._filter_annotations()
self._filter_images()

# Build new JSON
new_master_json = {
    'info': self.info,
    'licenses': self.licenses,
    'images': self.new_images,
    'annotations': self.new_segmentations,
    'categories': self.new_categories
}

# Write the JSON to a file
print('Saving new json file...')
with open(self.output_json_path, 'w+') as output_file:
    json.dump(new_master_json, output_file)
print('Filtered json saved.')

# Write annotation and images
print('copy images and writing annotations...')

cat_counter = {}
for category in self.new_categories:
    cat_counter[category['id']] = 0

with open(os.path.join(self.traintxt[0], "train.txt"), 'a') as traintxt_file:
    for idx, image in enumerate(self.new_images):

        #search allocation for image
        image_height = image['height']
        image_width = image['width']
        image_id = image["id"]

        image_categories = []

        for annotation in self.new_segmentations:
            with open(os.path.join(self.outputdir[0], ("coco"+str(idx)+".txt")),
                'a') as annotation_file:
                if str(image_id) == str(annotation['image_id']):
                    ctr_x =
                        ((annotation['bbox'][0]+(annotation['bbox'][2])/2))/image_width
                    ctr_y =
                        ((annotation['bbox'][1]+(annotation['bbox'][3])/2))/image_height
                    height = annotation['bbox'][3]/image_height
                    width = annotation['bbox'][2]/image_width
```

```
        annotation_file.write(str(annotation['category_id']-1) + ' ' +
                               str(ctr_x) + ' ' + str(ctr_y) + ' ' + str(width) + ' ' +
                               str(height) + '\n')

    # add category for image
    if annotation['category_id'] not in image_categories:
        image_categories.append(annotation['category_id'])

# check if all categories would fit in max count
isValid = True

for category in image_categories:
    if cat_counter[category] + 1 > self.amount:
        isValid = False

if isValid:
    # if it fits increment overall counter and add/download image
    for category in image_categories:
        cat_counter[category] += 1

    urllib.request.urlretrieve(image["coco_url"],
                                os.path.join(self.outputdir[0], ("coco" + str(idx)
                                                                    + ".jpg")))
    traintxt_file.write(os.path.join(self.outputdir[0], ("coco" + str(idx)
                                                            + ".jpg\n")))
    traintxt_file.flush()

else:
    # if it does not fit remove annotation
    print("one category reached the limit")
    os.remove(os.path.join(self.outputdir[0], ("coco"+str(idx)+".txt")))

print(cat_counter)

isFull = True
for category in cat_counter:
    if category < self.amount:
        isFull = False
if isFull:
    break

print('done.')
```

```
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Filter COCO JSON: "
                                               "Filters a COCO Instances JSON file to only "
                                               "include specified categories. "
                                               "This includes images, and annotations. Does not "
                                               "modify 'info' or 'licenses'.")
```

```
parser.add_argument("-i", "--input_json", dest="input_json",
                    help="path to a json file in coco format")
parser.add_argument("-o", "--output_json", dest="output_json",
                    help="path to save the output json")
parser.add_argument("-c", "--categories", nargs='+', dest="categories",
                    help="List of category names separated by spaces, e.g. -c person dog
                           bicycle")
parser.add_argument("-d", "--outputdir", nargs='+', dest="outputdir",
                    help="name of output directory")
parser.add_argument("-t", "--traintxt", nargs='+', dest="traintxt",
                    help="location of train.txt")
parser.add_argument("-a", "--amount", nargs='+', dest="amount",
                    help="limits the amount of images downloaded")
args = parser.parse_args()

cf = CocoFilter()
cf.main(args)
```

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Neuronale Indoor-Objekterkennung für mobile Roboter

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum Unterschrift im Original