Diplomarbeit

„Entwicklung eines Systems zur Artikelerkennung mit Hilfe von Künstlicher Intelligenz zur Überprüfung kommissionierter Waren“

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines

Diplom-Ingenieurs

unter der Leitung von

Univ.-Prof. Dipl.-Ing. Dr.tech. Georg Kartnig
(E307 Institut für Konstruktionswissenschaften und Produktenwicklung)

eingereicht an der Technischen Universität Wien

Fakultät für Maschinenwesen und Betriebswissenschaften

von

Urban Hanspeter
Matr.Nr. 01127617

Wien, November 2021

_____

Urban Hanspeter

Ich habe zur Kenntnis genommen, dass ich zur Drucklegung meiner Arbeit unter der Bezeichnung

Diplomarbeit

nur mit Bewilligung der Prüfungskommission berechtigt bin.

Ich erkläre weiters Eides statt, dass ich meine Diplomarbeit nach den anerkannten Grundsätzen für wissenschaftliche Abhandlungen selbstständig ausgeführt habe und alle verwendeten Hilfsmittel, insbesondere die zugrunde gelegte Literatur, genannt habe.

Weiters erkläre ich, dass ich dieses Diplomarbeitsthema bisher weder im In- noch Ausland (einer Beurteilerin/einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe und dass diese Arbeit mit der vom Begutachter beurteilten Arbeit übereinstimmt.

Wien, November 2021

_____

Urban Hanspeter

Genderklausel

Für die vorliegende Arbeit wurde wegen der besseren Lesbarkeit auf die gleichzeitige Verwendung weiblicher und männlicher Personenbegriffe verzichtet. Gemeint und angesprochen sind natürlich immer beide Geschlechter.

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die mich während des Erstellens dieser Diplomarbeit motiviert und unterstützt haben.

Ein besonderer Dank gilt dem Herrn Univ.-Prof. Dipl.-Ing. Dr.tech. Georg Kartnig, der in seiner Funktion als Betreuer und Ansprechpartner mich fachlich unterstützt hat. Weiters bedanke ich mich dafür, es mir ermöglicht zu haben, diese Diplomarbeit am Institut für Konstruktionswissenschaften und Produktentwicklung verfassen zu dürfen.

Zu guter Letzt möchte ich mich bei meinen Eltern, Brigitte und Stephan, für die bedingungslose Unterstützung während des ganzen Studiums bedanken. Sie waren stehts für mich da und waren in vielerlei Hinsicht eine große Stütze. Weiterer Dank gilt meinem Bruder Tobias, der immer wieder als Ansprechpartner ein offenes Ohr für mich hatte. Letzter Dank gebührt meiner Partnerin Magdalena, die mich mit ihrer positiven Einstellung und Motivation stehts zur Seite gestanden ist.

Die vorliegende Arbeit widme ich meinen Eltern Brigitte und Stephan.

Kurzfassung

Diese Arbeit befasst sich mit der Erkennung von Objekten zur Überprüfung von kommissionierten Artikeln bzw. Objekten mittels Künstlicher Intelligenz. Dabei wird ein Überblick über die theoretischen Hintergründe und Funktionsweisen von Neuronalen Netzen aufgezeigt. Dieser Teil der vorliegenden Arbeit fokussiert sich auf das Einsatzgebiet der Objekterkennung und die dafür verwendeten *Convolutional Neural Networks,* auf denen die Erkennungsalgorithmen basieren. Dabei werden die einzelnen Bausteine und Elemente desselben beschrieben.

Der zweite Teil der Arbeit bezieht sich auf die praktische Umsetzung eines Kommissionierassistenzsystemes. Dabei wurden zwei OD-Modelle (YOLOv3, YOLOv4) für die Erkennung von drei unterschiedlichen Artikeln mittels Google *Colab* trainiert. Diese sind für die Erkennung der Artikel zuständig, wobei YOLOv3 eine Erkennungswahrscheinlichkeit von über 82% und YOLOv4 eine Erkennungswahrscheinlichkeit von über 90% aufweist. Der dargestellte Kommissionierprozess beginnt mit der Erstellung einer Auftragsliste, in welcher die zu kommissionierenden Artikel aufgelistet sind. Anhand dieser Liste werden die Artikel bereitgestellt und mit Hilfe der OD-Modelle auf ihre Richtigkeit überprüft. Dabei wird eine Liste der erkannten Objekte erstellt und mittels eines Python-Skripts mit der Auftragsliste verglichen. Dadurch werden mögliche Differenzen erkannt. Sollte ein Fehler auftreten, müssen die falsch kommissionierten Artikel korrigiert werden und anschließend kann das OD-Modell die Artikel erneut prüfen. Sobald alle Artikel richtig kommissioniert sind, kann der nächste Auftrag bearbeitet werden.

Die Umsetzung der Objekterkennung mit den OD-Modellen YOLOv3 und YOLOv4 und der Einsatz als Assistenzsystems bei der Kommissionierung hat zufriedenstellende Ergebnisse geliefert und könnte zukünftig in der Industrie angewendet werden.

Abstract

This thesis discusses the recognition of objects for the verification of picked items or objects by using Artificial Intelligence. Thereby an overview of the theoretical background and functionality of neural networks is given. This part of the thesis focuses on the application field of object recognition and the Convolutional Neural Networks used for this purpose, on which the recognition algorithms are based on. The individual building blocks and elements of the same will be described.

The second part of the thesis refers to the practical implementation of a picking assistance system. Therefore two OD models (YOLOv3, YOLOv4) for the recognition of three different items were trained by Google Colab. These are responsible for item recognition, with YOLOv3 having a recognition probability of over 82% and YOLOv4 having a recognition probability of over 90%. The illustrated picking task starts with the creation of an order list, in which the items for picking are listed. Based on this list, the items are provided and checked by correctness using the OD models. A list of recognized objects is created and compared with the order list by a Python script. With this approach, possible differences can be detected. If an error occurs, the incorrectly picked items must be corrected and afterwards the OD model can check the items again. As soon as all articles are picked correctly, the next order can be processed.

The implementation of object recognition with the OD models YOLOv3 and YOLOv4 and the use as an assistance system during picking has delivered satisfying results and could be applied in the industry in the future.

Inhaltsverzeichnis

Teil A: Theorieteil

Teil B: Praxisteil, Zusammenfassung und Ausblick

Teil A

Theorieteil

1 Einleitung

Die vorliegende Arbeit setzt sich mit dem Einsatz von Künstlicher Intelligenz im Bereich der Logistik zur Überprüfung bzw. Erkennung von Artikeln im Bereich der Kommissionierung auseinander. Aufbauend auf den theoretischen Grundlagen und dem Stand der Technik soll ein Erkennungssystem konzeptioniert werden, welches in weiterer Folge eine Möglichkeit für den Einsatz in der Industrie darstellen kann.

Im folgenden Kapitel wird im ersten Teil eine allgemeine Einführung in das Themengebiet gegeben, um einen ersten Überblick zu geben. Weiters werden die behandelten Problemstellungen und Forschungsfragen aufgezeigt, die sich beim Erstellen dieser Arbeit ergeben haben. Anschließend wird der erarbeitete Lösungsansatz vorgestellt und die Struktur der Arbeit aufgezeigt.

1.1 Allgemeine Einführung in das Themenfeld

Objekterkennung wird heute in den verschiedensten Bereichen in unserem Alltag eingesetzt. Häufige Einsatzfelder sind z.B. die Videoüberwachung, die Fußgängerüberwachung, die Verleumdungserkennung, die selbstfahrenden Autos und die Erscheinungsbild-Erkennung. [1]

Grund für den häufigen Einsatz der Objekterkennung in den letzten Jahren ist die rapide Entwicklung von Computer Vision (CV). Die Objekterkennung (*Object detection*, OD) ist ein Teilgebiet von CV [2]. In der OD werden Merkmale und Eigenschaften von Bildern im ersten Schritt extrahiert. Auf dieser Grundlage aufbauend werden anschließend die erhobenen Objektinformationen analysiert und kategorisiert. Um die Kategorisierungen vornehmen zu können, werden verschiedene Machine Learning-Methoden (ML) eingesetzt. Eine besondere Art von ML ist *Deep Learning* (DL). Dieser Algorithmus wird aufgrund seiner großen Fortschritte in den letzten Jahren für CV eingesetzt. DL hat die Fähigkeit, viele Merkmale zu verarbeiten und zu analysieren. Grund dafür ist, dass der Algorithmus, ähnlich wie das menschliche Gehirn, eigenständig Dinge erlernen und imitieren kann. Im Speziellen bedeutet dies, dass der Algorithmus im Stande ist, mit Trainingsdaten zu lernen, Merkmale bzw. Muster eigenständig zu erkennen und in Folge anhand dieser Erkenntnisse, Entscheidungen zu treffen.

Ein großes Anwendungsgebiet, in welchem OD-Modelle zur Leistungssteigerung beitragen können, ist die Logistik. Ein großer Vorteil beim Einsatz von OD ist, dass Arbeitsschritte, wie z.B. die Kommissionierung schneller, weniger fehlerhaft und somit kostengünstiger durchgeführt werden können. Dies war auch u.a. ein Anlass dafür, im

Rahmen dieser Diplomarbeit ein Erkennungssystem für kommissionierte Ware zu entwickeln.

1.2 Problemstellung / Forschungsfragen

Von 2009 bis 2018 stieg in Deutschland der Umsatz im Versandhandel von 30 Mrd.€ auf 72,2 Mrd.€. Dies entspricht einem Anstieg von rund 140,7%. Dieses große Wachstum bedeutet, dass die Logistikzentren mehr als je zuvor von effizienter Arbeit abhängig sind. Einer der Prozesse, der in der Logistik von großer Bedeutung ist, ist das Kommissionieren der Waren bzw. Artikel. Da es vorkommt, dass die Kommissionierer und Konsolidierer mehrere Kundenaufträge (Bestellungen) gleichzeitig bearbeiten, kann es hierbei häufig zu Fehlern und hohem Zeitaufwand kommen. Durch das Benutzen eines Assistenzsystems im Kommissionierprozess wurde nachgewiesen, dass sich die Dauer für jenen Prozess auf bis zu 30% senken lässt. Weiters wird nicht nur die Bearbeitungszeit reduziert, sondern es können durch das Überprüfen der Richtigkeit der Artikel mittels OD, Fehler vermieden werden. [3]

Aus diesem Grund wurde im Rahmen dieser Diplomarbeit ein OD-Modell entwickelt, welches in der Lage ist, sowohl auf Bild- als auch auf Videoaufnahmen vordefinierte Objekte zu erkennen. Dies soll in weiterer Folge als Überwachungsinstrument dienen, um den Kommissionierprozess so effizient wie möglich zu gestalten.

Aus der Problemstellung ergeben sich folgende Forschungsfragen:

- Welche Möglichkeiten zum Erkennen von Objekten mit KI im Kommissionierprozess und welche signifikanten Unterschiede gibt es zwischen diesen?
- Wie können die kommissionierten Objekte im Kommissionierprozess durch Bild- oder Videoaufnahmen mittels KI mit einer hohen Wahrscheinlichkeit erkannt werden?
- Wie könnte ein Prozess für die Umsetzung bzw. in der Anwendung im Unternehmen aussehen?

Nach einer Grundlagenrecherche der theoretischen Hintergründe und einem Überblick über den State-of-the-Art soll im Praxisteil der Diplomarbeit eine Methode aufgezeigt werden, um die Forschungsfragen beantworten zu können. Somit ergeben sich folgende Ziele für die vorliegende Arbeit:

- Recherche der theoretischen Grundlagen
- Aufzeigen der verschiedenen Algorithmen/Modelle, die für OD eingesetzt werden können
- State-of-the-Art-Analyse für die Anwendungen von OD

- Erstellen eines OD-Modelles auf Basis der erwähnten Punkte
- Auswertung der Ergebnisse

1.3 Lösungsansatz / Arbeitspakete

Um die genannten Forschungsfragen beantworten zu können, wird zunächst auf die theoretischen Hintergründe von OD und dessen Funktionsweise eingegangen. Eine umfangreiche Grundlagenrecherche soll als Fundament für die Umsetzung eines OD-Modells dienen. Nach dem Aufzeigen verschiedener OD-Algorithmen, wird der für diese Arbeit am besten geeignete ausgewählt.

Im praktischen Teil dieser Diplomarbeit wird ein Datensatz erstellt, der Bilder mit den zu erkennenden Objekten enthält. Dieser Datensatz wird anschließend verwendet, um den OD-Algorithmus zu trainieren, so dass dieser im Stande ist, die Objekte auf noch nie gesehenen Bild- bzw. Videoaufnahmen erkennen zu können. Das Trainieren des Modelles wird auf der Online-Plattform *Colab* (Google Colaboratory) vorgenommen. Nach diesem Prozess wird der trainierte Algorithmus getestet. Dazu wird in der IDE (*integrated development environment*) PyCharm mit Hilfe der Python Library OpenCV eine Erkennungssoftware erstellt, um das Testen lokal vornehmen zu können. Zusätzlich werden auch Tests direkt in *Colab* vorgenommen.

Um die Objekte überprüfen zu können, wird im Python-Skript eine Funktion integriert, welche eine Liste mit den erkannten Objekten erstellt. Diese wird mit einer zweiten Liste, welche eine Auftragsliste darstellen soll, verglichen. Mittels dieser Methode soll ein möglichst realitätsnahes Szenario für einen Kommissionierprozess dargestellt werden.

1.4 Aufbau und Struktur der Arbeit

Als Basis und Einführung wird in Kapitel 1 die Ausgangslage dieser Arbeit beschrieben. Zu Beginn wird allgemein das Themenfeld vorgestellt. Darauf folgt die Darlegung der Problemstellung und die daraus abgeleiteten Forschungsfragen. Weiters wird noch auf den Lösungsansatz und die verwendeten Arbeitspakete überblicksartig eingegangen.

Kapitel 2 dient als theoretische Grundlage, um die Funktionsweisen der Thematik kennenzulernen. Hierbei wird auf die OD und deren einzelne Elemente eingegangen. Zu Beginn werden Neuronale Netze (NN) und der Trainingsprozess dieser aufgezeigt. Das Hauptaugenmerk wird hierbei auf die *Convolutional Neural Networks* (CNN) gelegt, da im Praxisteil ein solches für den Erkennungsvorgang verwendet wird.

Abschließend werden noch verschiedene OD-Algorithmen und deren Unterschiede aufgezeigt.

Die Einsatzgebiete von OD-Algorithmen (OD-Modelle) werden in Kapitel 3 durch eine State-of-the-Art Analyse aufgezeigt. Im Besonderen wird hierbei auf den Einsatz in der Logistik und im Straßenverkehr eingegangen.

In Kapitel 4 wird ein Konzept für die Umsetzung und Implementierung erarbeitet.

Die Umsetzung und Implementierung der OD-Modelle werden in Kapitel 5 aufgezeigt. Dabei wird jeder erforderliche Schritt erklärt und auf die Unterschiede zwischen YOLOv3 und YOLOv4 eingegangen. Weiters wird die Performance der OD-Modelle gemessen und einander gegenübergestellt. Zum Schluss wird eine Möglichkeit vorgestellt, wie die OD-Modelle durch einen Listenvergleich für das Überprüfen von Waren im Kommissionierprozess eingesetzt werden können.

In Kapitel 6 werden Herausforderungen bzw. Probleme, die sich bei der Umsetzung ergeben haben, und die Behebung dieser aufgezeigt. Ebenso werden die für diese Arbeit formulierten Forschungsfragen beantwortet.

Abschließend werden im Kapitel 7 die Kernaussagen dieser Arbeit zusammengefasst und im Ausblick mögliche Verbesserungen für die Zukunft angesprochen.

Die vorliegende Arbeit wird in Teil A, welcher den Theorieteil umfasst (Kapitel 1 bis Kapitel 3) und Teil B, der den Praxisteil und das Resümee beinhaltet (Kapitel 4 bis Kapitel 7), aufgeteilt. Der Aufbau der vorliegenden Arbeit ist in Abbildung 1 dargestellt.

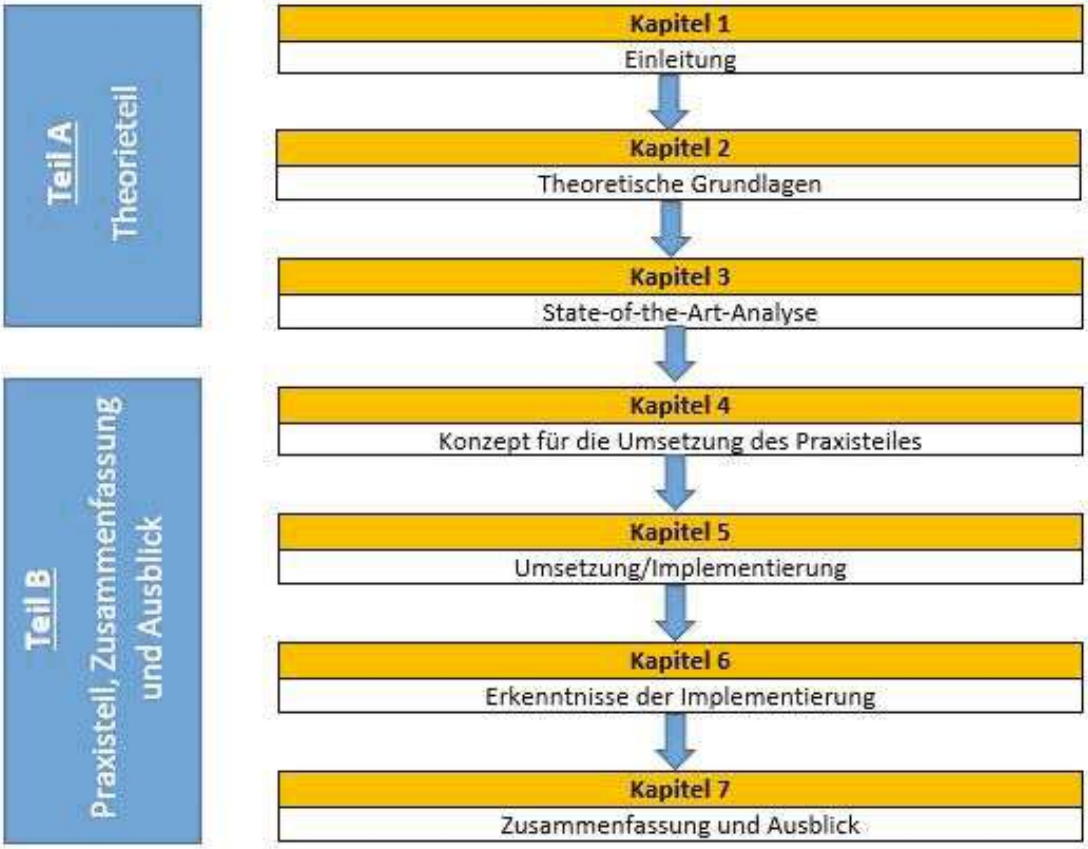


Abbildung 1: Überblick über den Aufbau der Diplomarbeit

2 Theoretische Grundlagen

Im Kapitel Theoretische Grundlagen werden wichtige Begriffe und Funktionsweisen der OD erläutert. Ausgehend von der Begriffsdefinition der Künstlichen Intelligenz wird der Aufbau neuronaler und künstlich neuronaler Netze aufgezeigt. In weiterer Folge wird der Trainingsprozess von künstlichen neuronalen Netzen (KNN) beschrieben und dabei auf die wichtigsten Schritte eingegangen. Danach wird die Funktionsweise von *Convolutional Neural Networks* (CNN) beschrieben. *Transfer Learning*, eine Technik, die für das effiziente Training von KNN verwendet wird, wird in der Folge behandelt. Zum Schluss dieses Kapitels wird allgemein auf OD eingegangen und welche Daten dafür verwendet werden können. Des Weiteren werden verschiedene OD-Modelle beschrieben.

2.1 Künstliche Intelligenz (KI)

„Artifcial Intelligence is the study of how to make computers do things at which, at the moment, people are better.“ (def. Rich 1983) [4]

Mit dieser allgemeinen Definition wurde schon 1983 deutlich gemacht, dass die Möglichkeiten und Grenzen neuer Technologien immer neu definiert werden müssen, da sie sich in kurzer Zeit maßgeblich verändern können. Künstliche Intelligenz (KI) wird heutzutage als die Fähigkeit einer Maschine bezeichnet, menschliche Aufgaben ausführen zu können. Im Speziellen besitzt die Maschine die Fähigkeit, kognitive Aufgaben wie Beschreibungen, Vorhersagen und Empfehlungen durch eigenständiges Lernen bewältigen zu können. [4]

Bei der Bezeichnung KI handelt es sich um einen Oberbegriff, der ein großes Gebiet abdeckt. In Abbildung 2 ist die Eingrenzung der verschiedenen Begrifflichkeiten zu sehen.

Abbildung 2: Überblick über die Teilbereiche von KI [4]

Die NN dienen zur Modellbildung von KI-Systemen und durch das ML werden die gewünschten Konzepte bzw. das Wissen durch das Verwenden von Algorithmen generiert. DL stellt spezielle Algorithmen dar, die das ML unterstützen. Das Wort „Deep“ bezieht sich dabei auf die Anzahl der Schichten des NN. [4]

2.2 Neuronale Netze (NN)

Ein wichtiger Teil der KI sind künstliche Neuronale Netze (KNN). Dieser Begriff kommt ursprünglich aus der Neurologie. Neuronale Netze bezeichnen in der Neurologie die Verbindungen von Neuronen im menschlichen Gehirn [4]. Diese Verknüpfungen und Funktionsweisen gelten als Inspiration für das Schaffen von KNN. KNN wiederum können jedoch nicht als realistische Modelle des Gehirns verwendet werden. Allerdings erlauben uns KNN, ganz vereinfachte Modelle des Gehirns zu bilden und dessen Verhaltensweisen nachzuahmen [5]. KNN bestehen aus verschiedenen Schichten, sogenannten *Layers*. Der erste *Layer* wird *Input-Layer* genannt. Dieser wird mit den erforderlichen Rohdaten (Bildern etc.) gefüttert und ist nur für die Aufnahme dieser zuständig. Im nächsten Schritt werden die aufgenommenen Daten an den *Hidden-Layer* weitergegeben und verarbeitet. Es können mehrere *Hidden-Layer* hintereinander angeordnet sein, wobei die verarbeiteten Daten immer vom vorherigen *Layer* weitergegeben werden. In manchen Modellen kann ein KNN aus 10.000 solcher *Layer* bestehen. Zum Schluss befindet sich der *Output-Layer*. Dieser gibt das Ergebnis des KI-Systems aus (Abbildung 3).

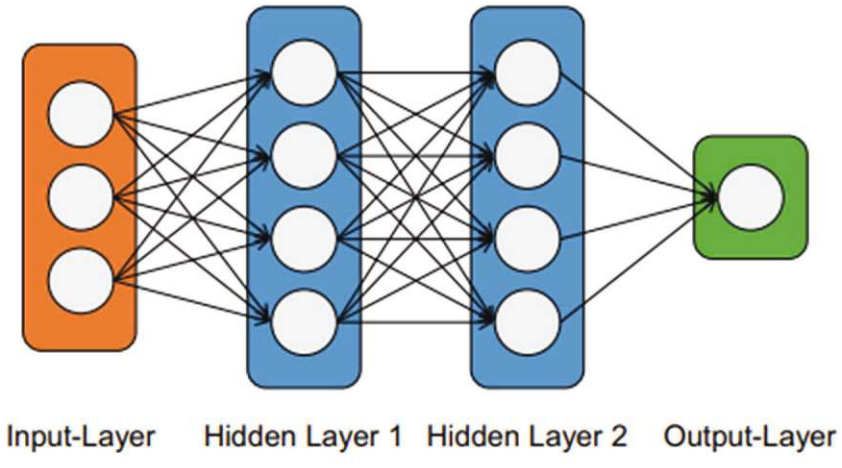


Abbildung 3: Aufbau eines künstlichen Neuronalen Netzes (KNN) [4]

Jeder Verarbeitungsknoten wird anhand vorhandener Regeln programmiert. Eine solche Verarbeitung verfügt nicht nur über deren programmierte Regeln (Algorithmus),

sondern ist im Stande durch Maschinelles Lernen (*Machine Learning*, ML) eigenständig zu lernen, d.h. Verarbeitungsschritte zu ergänzen, zu korrigieren und somit zu verbessern. Im Allgemeinen können Algorithmen auf drei verschiedene Arten lernen [4]:

- ***Supervised Learning* (Beaufsichtigtes Lernen)**

  Beim *Supervised Learning* erhält der ML-Algorithmus den *Input* und den dazugehörigen *Output*. Anhand dieser Informationen versucht der Algorithmus Muster zu erkennen, die erklären, wie es zum dazugehörigen *Output* gekommen ist. Häufig verwendete Methoden, die bei dieser Lernmethode angewendet werden, sind Lineare Regressionen, Lineare Diskriminanzanalysen und das Entscheidungsbaumverfahren. Mittels dieser lernt der Algorithmus mit den bereits bekannten Trainingsdaten und Ergebnissen und kann, sobald die gewünschte Genauigkeit erreicht ist, für neue Daten verwendet werden.

  Anwendungsbereich: Bilderkennung, Gefahrerkennung bei selbstfahrenden Autos (siehe Kapitel 3) etc.

- ***Unsupervised Learning* (Nicht-überwachtes Lernen)**

  Beim *Unsupervised Learning* erhält der ML-Algorithmus keine vordefinierten Inputs und/oder Outputs, sondern muss auftretende Muster und Merkmale selbst erkennen. Der Algorithmus erhält unbearbeitete Daten, identifiziert Ähnlichkeiten und gruppiert diese in der Folge. Um dieses sogenannte *Clustering* vornehmen zu können, werden unter anderem hierarchische und *K-Means-Clusterung* angewendet. Diese Lernmethode ist im Vergleich zum *Supervised Learning* weniger erfolgreich.

  Anwendungsbereich: Auswertung des Nutzungsverhaltens von Personen in Sozialen Medien (Wann ist man online? Welche Seite wird oft aufgerufen? Welche Likes werden gesetzt? etc.)

- ***Reinforcement Learning* (Verstärkendes Lernen)**

  Beim *Reinforcement Learning* ist der Algorithmus im Stande, mit einem Belohnungsprinzip selbständig Lösungen zu erarbeiten. Das System probiert durch einen iterativen Prozess verschiedene Lösungsansätze aus. Führt diese Methode zu einem akzeptablen Ziel, wird das durch eine „Belohnung“ gewertet und sobald sich ein Ansatz vom Ziel entfernt, wird dieser „bestraft“. Mit dieser Vorgehensweise korrigiert und optimiert sich der Algorithmus selbstständig. Ein Beispiel für diese Lernvariante ist das Duell des Go-Weltmeisters (Go:

strategisches Brettspiel) mit einem für *AlphaGO* programmierten KI-System. Nach einigen Partien gegeneinander konnte sich der Algorithmus durch die gesammelten Erfahrungen bei den Siegen und Niederlagen kontinuierlich verbessern und war in der Lage den Weltmeister zu besiegen.

Anwendungsbereich: Szenarien, in denen das Ergebnis nicht klar definiert ist; neue Erkenntnisse durch Interaktion mit der Umwelt

Eine besondere Art von KNN und Teilgebiet von ML ist das *Deep Learning* (DL) bezeichnet (siehe Abbildung 2). DL unterscheidet sich von herkömmlichen ML-Methoden aufgrund der Tiefe des Modells. Die „Tiefe“ bezieht sich dabei auf die Anzahl der vorhandenen Schichten (*Layer*) des Modells. Wegen dieser Besonderheit ist ein DL-Netzwerk in der Lage, mehr Eingabedaten aufzunehmen und eine größere Anzahl von Daten zu verarbeiten. Bei einfachen KNN werden manuell Regeln für Algorithmen definiert, um Merkmale bzw. Unterschiede zu filtern. Bei DL-Modellen hingegen werden diese Regeln im Laufe des Trainingsprozesses automatisch erlernt. [4]

2.2.1 Activation Function

Die KNN dienen als Baublöcke für die DL-Modelle. Um ein System als NN (*Neural Network)* bezeichnen zu können, muss dieses eine Graphenstruktur enthalten (siehe Abbildung 3). Dabei führt, wie bereits erwähnt, jeder Knoten eine Berechnung durch. Das Ergebnis der Berechnung (Signal) wird anschließend mit einer Gewichtung (*Weight*) zum nächsten Knoten weitergeschickt. Die Gewichtung gibt dabei an, ob das Signal abgeschwächt oder verstärkt wird. Somit geben „positive“ Gewichtungen eines Signales an, dass diese für die Klassifizierung sehr wichtig sind, wohingegen „negative“ Gewichtungen genau das Gegenteil bewirken. Bei einem KNN sind die Gewichtungen durch einen Lernalgorithmus modifizierbar und können somit für verschiedene Anwendungen in der OD trainiert werden. [5]

Dies geschieht im menschlichen Gehirn auf eine sehr ähnliche Art und Weise. Das Gehirn besteht aus Neuronen, welche mit anderen Neuronen verbunden sind. Der Zellkörper (*Cell Body*) eines Neurons wird Soma genannt und ist in der Lage, durch die Dendriten (*Dendrite*) *Inputs* zu erhalten und nach Verarbeitung dieser die *Outputs* (*Axon*) weiterzugeben. Dies erfolgt aber nur dann, wenn das Inputsignal stark genug ist, um das Neuron zu aktivieren. Dieser Prozess kann dann ein Signal entlang der

verknüpften Neuronen weitergeben. Eine Darstellung eines menschlichen Neurons ist in Abbildung 4 (links) zu sehen.

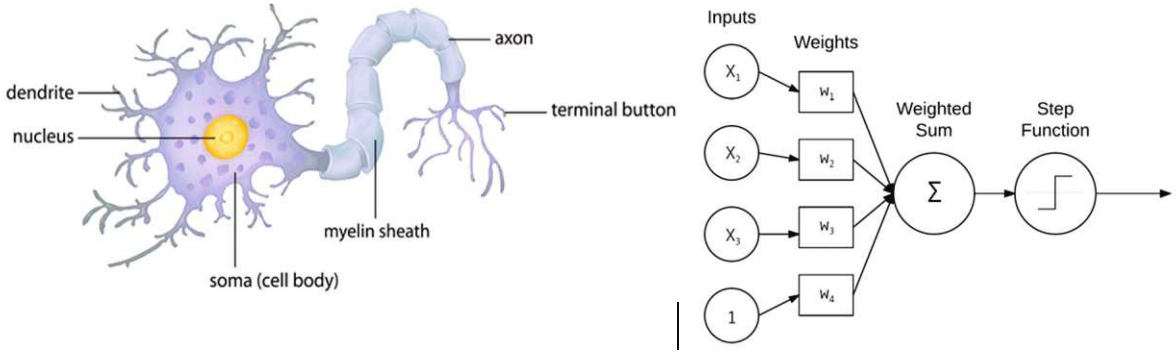


Abbildung 4: links: Darstellung eines Neurons des menschlichen Gehirns; rechts: Darstellung eines KNN [5]

Dem menschlichen Neuron wird in Abbildung 4 (rechts) ein einfaches KNN gegenübergestellt. Die Variablen x1, x2 und x3 sind die Inputs für das NN, die z.B. der Pixelintensität eines Bildes entsprechen. Der Wert 1 (Abbildung 4 rechts) stellt den *Bias* b (gewollte Verzerrung) dar. Diese Werte entsprechen den Werten des *Feature-Vektors* (Merkmale-Vektor), welche die Intensität der Pixel in einem Bild darstellen. Für jeden *Input* x gibt es eine zugehörige Gewichtung (*Weights, w*), die mit dem Neuron verbunden ist. Anschließend wird das Produkt mit der Formel (1) aufsummiert und zu einer *Activation Function* (z.B. *Step Function*) weitergeleitet. Sobald die *Activation Function* aktiviert wird, wird das Signal dem nächsten Neuron als *Input* weitergeleitet.

$$f(x_i, w,) = \sum_{i=1}^{n} w_i * x_i \qquad (1)$$

In Abbildung 5 sind die *Activation Functions* abgebildet. Dabei ist zu erwähnen, dass die *ReLU*- und *Leaky ReLU*-Funktion, die häufigsten *Activation Functions* in DL sind. [5]

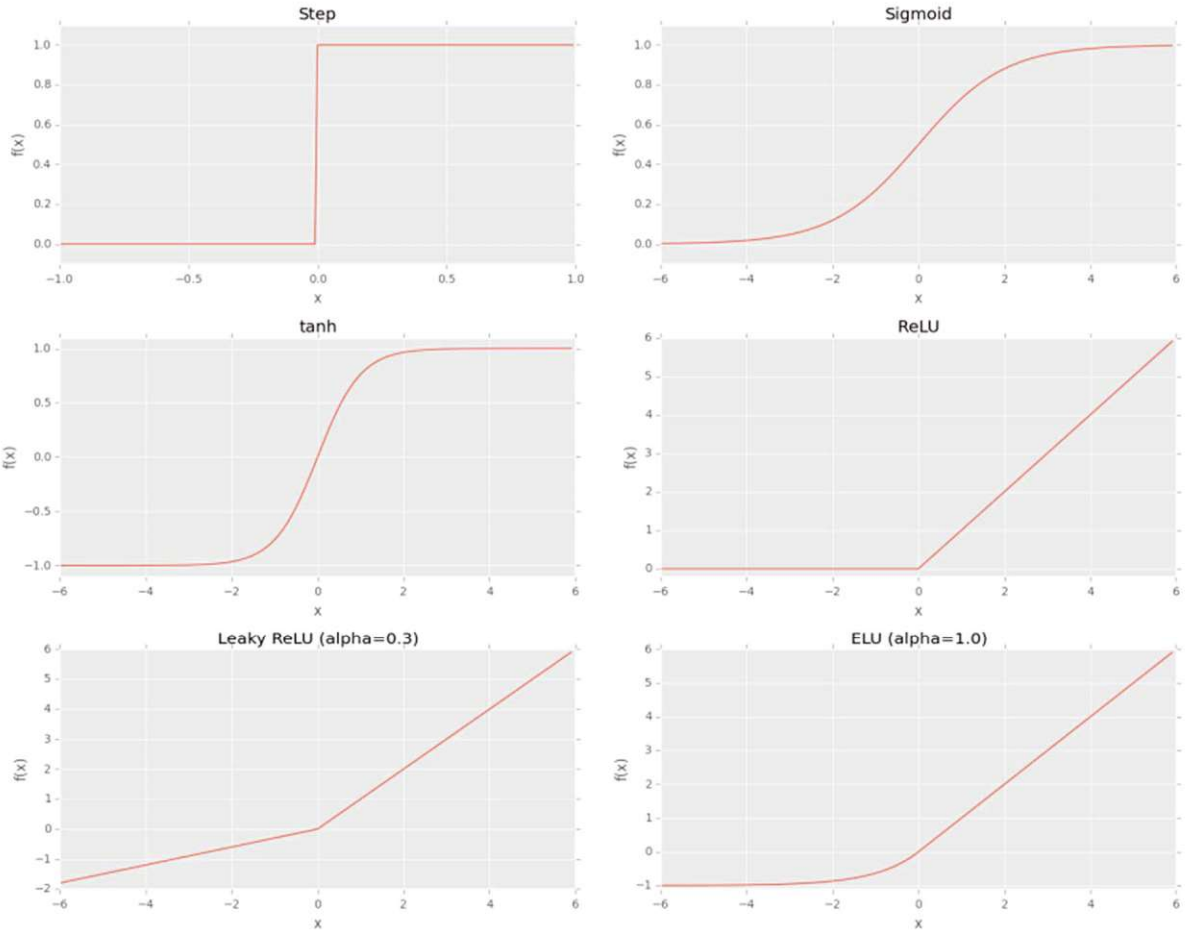


Abbildung 5: Verschiedene *Activation Functions* die bei NN angewendet werden. Von oben links: *Step Function; Sigmoid Activation Function*; tanh; ReLU (in NN am häufigsten verwendet); Leaky ReLU (verarbeitet auch negative Werte); ELU. [5]

Für weitere Erklärungen und die Anwendung der Formeln wird auf die Fachliteratur verwiesen. [5]

2.2.2 Feedforward Neural Network

Es gibt sehr viele unterschiedliche NN-Modelle. In dieser Arbeit wird nur auf das *Feedforward* NN eingegangen, da der Algorithmus (YOLOv3/v4), der im Praxisteil verwendet wird, auf diesem basiert.

Feedforward NN (Abbildung 6) zeichnen sich dadurch aus, dass bei der Verbindung zwischen den Knoten das Signal von *Layer* i nur zum *Layer* i+1 weitergeleitet wird. Es gibt keine „Rückwärts“ Verbindung oder Verbindungen zwischen den Knoten im *Layer* selbst. Jedoch kann *Feedforward* NN sogenannte *Feedback Connections* aufweisen, bei welchen der *Output* wiederum als *Input* verwendet werden kann.

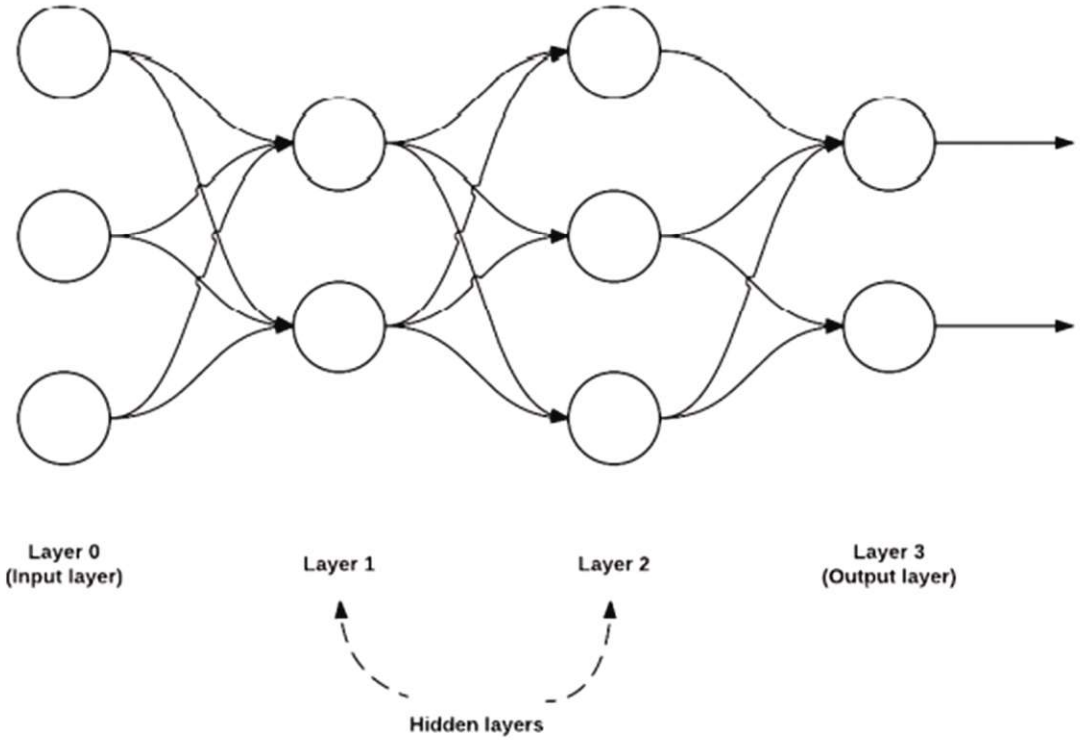


Abbildung 6: Darstellung eines *Feedforward* NN`s [5]

2.3 Training von NN

Das Trainieren eines NN entspricht einem Optimierungsproblem, bei dem der Fehler zwischen *Output* und Zielwert minimiert werden soll. Im Feld der OD wird vor allem *Supervised Learning* angewandt, bei dem ein gelabelter Datensatz (z.B. mit Bildern, siehe Kapitel 4.3) verwendet wird, damit dieser mit dem Output verglichen werden kann. Durch das *Encoding* (Kapitel 2.3.1) werden die Daten im Datensatz in ein computertaugliches Format gebracht, damit diese verwendet werden können. Durch das Verwenden einer *Loss-Function* (Kapitel 2.3.2) ist es möglich, das Optimierungsproblem (Kapitel 2.3.3) zu bewerten und mittels *Backpropagation* (Kapitel 2.3.4) durch iterative Prozesse die Werte für ein gegebenes OD-Problem im Trainingsprozess anzupassen. Um optimale Trainingsresultate zu erhalten und den Trainingsprozess steuern zu können, werden die Hyperparameter *Learning Rate*, *Batch Size*, *Epochs* und *Iterations* (Kapitel 2.3.5) benötigt.

2.3.1 Encoding

Der Datensatz wird gelabelt, indem man Objekten Codes bzw. Nummern zuordnet. Hier geht es darum, die Daten in ein Format zu bringen, dass ein Computer diese

verarbeiten kann. Es gibt im Allgemeinen viele unterschiedliche Prinzipien um zu *Labeln*. Die gängigsten Methoden, die in DL eingesetzt werden, sind *Label Encoding* und *One Hot Encoding*. [6]

Beim *Label Encoding* werden den verschiedenen Kategorien im Datensatz Nummern beginnend bei 0 zugeteilt. Ein Beispiel dafür ist in Abbildung 7 (links) zu sehen. Der Datensatz besteht aus 3 Kategorien: Katze, Hund und Zebra. Diesen drei Kategorien werden die Nummern 0-2 zugeordnet, damit der Computer diese Information verarbeiten kann. Beim *One Hot Encoding* werden hingegen jeder Kategorie in Vektorform Nullen und Einsen zugeordnet. Wie in Abbildung 7 rechts zu sehen ist, werden für jede Kategorie Vektoren erstellt. Dabei steht bei der richtigen Klasse eine Eins und alle anderen Einträge sind mit einer Null befüllt. So wird dem Computer verdeutlicht, welcher Kategorie welches Bild angehört.

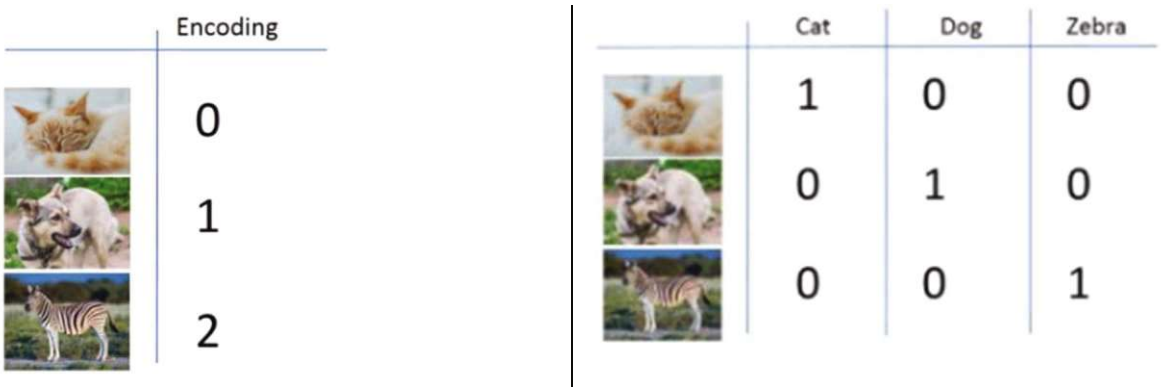


Abbildung 7: links: *Label Encoding*; rechts: *One Hot Encoding* [7]

2.3.2 Loss-Function

Mit Hilfe der *Loss-Function* wird geprüft, wie gut das Ergebnis des NN bei der Erkennung war, indem der *Output* mit dem *gelabelten Input* abgeglichen wird. Hier gilt, je höher die Übereinstimmung dieser Werte, desto genauer ist das NN bei der Erkennung. Deshalb ist das Ziel, das Ergebnis der *Loss-Function* zu minimieren. [5]

Es gibt viele verschiedene *Loss-Functions,* die für das Bestimmen der Performance eines NN eingesetzt werden können. Die am meisten verwendete ist der *Mean Square Error* $L(y, \hat{y})$ (MSE, in Formel (2)). In Formel (3) werden mit y die Werte des *Inputs* und mit $\hat{y}$ die Werte des *Outputs* bezeichnet. m gibt dabei die Anzahl der verarbeiteten *Input*-Signale an. [8]

$$L(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 \qquad (2)$$

Angenommen, es wird ein NN für das Erkennen von drei Kategorien, welche in Abbildung 7 zu sehen sind, trainiert. Als *Input* wird ein Bild verwendet, auf welchen eine Katze abgebildet ist. Der *Input*-Vektor y sieht dann wie in Formel (3) aus und entspricht gleichzeitig dem idealen *Output* des NN. Der tatsächliche *Output* ist der Vektor $\hat{y}$. Anhand dieser 2 Vektoren wird der *Loss*-Wert berechnet, den es zu optimieren gilt.

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \hat{y} = \begin{bmatrix} 0.82 \\ 0.01 \\ 0.12 \end{bmatrix} \tag{3}$$

Die Werte des *Output*-Vektors können auch Werte über 1 annehmen. Aus diesem Grund wird eine *Softmax-function* angewendet, die die Werte normalisiert, also zwischen 0 und 1 bringt. [9]

2.3.3 Optimierung durch *Gradient Descent*

Um das Ergebnis der *Loss- Function* zu verbessern bzw. zu optimieren, gilt es die Parameter w (siehe Formel (1)) bzw. den Bias b zu verändern (siehe Kapitel 2.2.1). Die in DL meistverwendete Optimierungsmethode ist der *Gradient Descent* (Gradientenverfahren). *Gradient Descent* ist ein Algorithmus, der durch iterative Schritte versucht, die optimale Lösung für ein gegebenes Problem zu finden. Ein Beispiel wird in Abbildung 8 dargestellt, wo auf der x-Achse die Gewichte und auf der y-Achse das dazugehörige Ergebnis der *Loss-Function* aufgetragen wird. Es ist erkennbar, dass bei den unterschiedlichen *Weights* die *Loss-Function* Höhen und Tiefen *Global-* und *Local Minimum* als Ergebnis liefert. Das Ziel ist es idealerweise, das Globale Minimum zu finden, sodass das Ergebnis der *Loss-Function* minimiert wird.

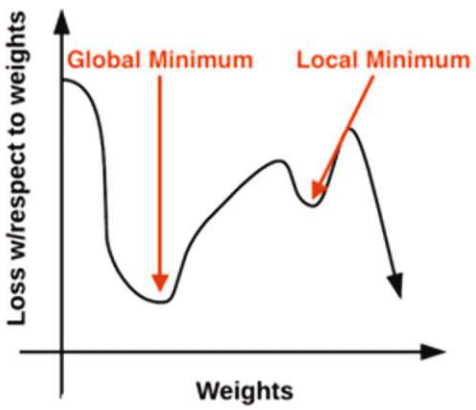


Abbildung 8: y-Achse: Ergebnisse einer *Loss-Function*; x-Achse: *Weights* [5]

Die am meisten verwendete *Gradient Descent* Methode ist der *Stochastic Gradient Descent* (SGD). In diesem Verfahren werden die *Weights* berechnet und in kleinen *Batches* (siehe Kapitel 2.3.5) von Trainingsdaten (*Inputs*) aktualisiert. Dies liefert im Vergleich zum normalen *Gradient Descent*, bei welchem der ganze Datensatz auf einmal betrachtet wird, schnellere Ergebnisse. [5]

2.3.4 Backpropagation

Der wichtigste Baustein eines NN, der das Lernen überhaupt ermöglicht, ist die *Backpropagation*. Diese ist dafür zuständig, dass in dieser Lernphase die *Weights* aktualisiert werden.

Backpropagation lässt sich in zwei Phasen einteilen. Die erste Phase wird *Forward Pass* und die zweite *Backward Pass* genannt. In der *Forward Pass* Phase werden alle *Inputs* durch das NN geschickt und die *Outputs* dafür erzeugt. In der zweiten Phase werden die berechneten *Weights* von dem SGD mittels der Anwendung der Kettenregel im NN aktualisiert. Damit *Backpropagation* funktioniert, bzw. angewendet werden kann, müssen die *Activation Functions* differenzierbar sein (siehe Kapitel 2.2.1).

Die *Backpropagation* Methode bildet das Kernelement für das Trainieren von NN und wird bei einfachen, bis hin zu sehr komplexen NN Modellen angewendet. [5]

2.3.5 Hyperparameter

Um den Lernprozess eines NN-Modelles zu kontrollieren, werden sogenannte Hyperparameter benötigt. Solche Parameter sind *Learning Rate*, *Batch Size*, *Epochs* und *Iterations*.

Wie in Kapitel 2.3.3 beschrieben, handelt es sich bei dem SGD um ein iteratives Verfahren zur Optimierung eines Problems. Das heißt, durch das Ergebnis mehrerer Durchläufe, wird dies Schritt für Schritt angepasst bzw. optimiert. Aus diesem Vorgang resultiert, dass der Wert der *Loss-Function* immer geringer wird, was als positives Anzeichen gedeutet werden kann. Zu Beginn sind die Schritte dieser Annäherung größer, werden aber laufend kleiner, bis die *Loss-Function* das gewünschte Ergebnis liefert (je kleiner, desto besser). Dieses Verhalten wird als *Learning Rate* bezeichnet. Die *Learning Rate* kann durch die Parameter *Batch Size*, *Epochs* und *Iterations* gesteuert werden.

Die *Epochs* geben an, wie oft der Datensatz als Input benutzt wird. Das bedeutet, dass bei einem *Epoch* = 1 der *Input* nur ein einziges Mal durch das NN läuft. Da die *Weights*

immer wieder aktualisiert werden müssen, wird dies zu keinen guten Ergebnissen führen. Das NN kann durch die Anzahl der *Epochs* zu *Overfitting* bzw. *Underfitting* führen. Durch eine hohe Zahl an *Epochs* tendiert das NN zu *Overfitting*. Das bedeutet, dass die Ergebnisse für den Trainingsdatensatz sehr gut ausfallen, aber auf andere *Inputs* schlechte Ergebnisse geliefert werden. *Underfitting* hingegen kann das Ergebnis von zu wenigen *Epochs* sein. Hier liefert das NN schlechte Ergebnisse, sowohl am Trainigs-*Input* als auch an generellen *Inputs*. Es gilt, eine Summe an *Epochs* zu wählen, bei welcher keiner dieser Fälle auftreten kann. In Abbildung 9 sind die Szenarien dargestellt.

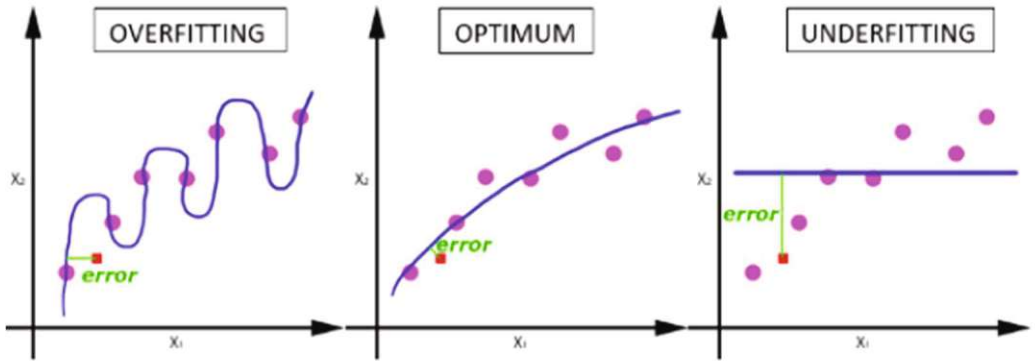


Abbildung 9: Darstellung von *Overfitting* vs. Optimum vs. *Underfitting* [10]

Da das Verarbeiten des gesamten *Inputs* in einem Schritt zu schlechten Lernergebnissen führen würde, wird dieser in Pakete geteilt und anschließend in das NN geschickt. Die Zahl der Pakete des *Inputs* heißt *Batch Size*.

Um die dazugehörigen *Iterations* zu erhalten wird die Anzahl der *Inputs* durch die *Batch Size* dividiert. Das heißt, die *Iteration* bedeutet, wie oft die Pakete durch das NN eingespeist werden müssen, damit ein bestimmter *Epoch* erreicht wird. Zum Beispiel enthält ein Datensatz 5000 Daten und die *Batch Size* ist 100. Somit ist die *Iteration* gleich 50. [10]

2.4 Convolutional Neural Networks (CNN)

Convolutional Neural Networks, auch bekannt als CNN, sind eine besondere Art von KNN, um Daten zu verarbeiten. Wie der Namen bereits sagt, handelt es sich um ein NN das eine mathematische Operation, eine sogenannte *Convolution* ausführt. Ein Merkmal eines CNN ist dessen gitterartige Topologie (siehe Abbildung 11). [11]

In einem regulären KNN wird ein Input-Vektor in den *Hidden-Layers* (können auch mehrere sein) bearbeitet, während der *Output*-Vektor als Maß für die OD gilt. Dabei

sind die Knoten nur mit den Knoten im nächsten *Layer* verbunden und nicht im *Layer* selbst. Ein solches NN ist aber nicht mehr ideal, sobald es um das Erkennen von Objekten auf Bildern geht. Bei einem Bild handelt es sich um einen *Input,* der drei Dimensionen aufweist, Breite (*Width*), Höhe (*Height*) und Tiefe (*Depth*). Die Tiefe beschreibt die Anzahl der Farb-*Channels* R = Red, G = Green und B = Blue und ergibt deshalb den Wert drei. So hat ein Bild z.B. die Form 32x32x3. Würde nun ein Bild mit dieser Größe als *Input* für ein reguläres NN verwendet werden, ergäben sich im ersten *Layer* 32x32x3=3072 *Weights*. Dies ergibt eine sehr hohe Anzahl an Parametern, welche sich auf die Performance des NN negativ auswirken. [12]

Im Gegensatz zu KNN werden bei CNN die Knoten des Netzwerkes dreidimensional angeordnet. Bei der Verarbeitung des Inputs sind die Knoten der *Layer* nicht mehr vollständig mit allen Knoten des jeweiligen darauffolgenden *Layers* verknüpft (*Fully Connected*), sondern die Knoten sind jeweils nur mehr mit einigen der darauffolgenden verbunden. Dabei werden nach Durchlaufen des CNN dreidimensionale Input-Vektoren in einem einzigen Vektor entlang seiner Tiefe angeordnet. In Abbildung 10 sieht man den Unterschied der beiden Netzwerke.

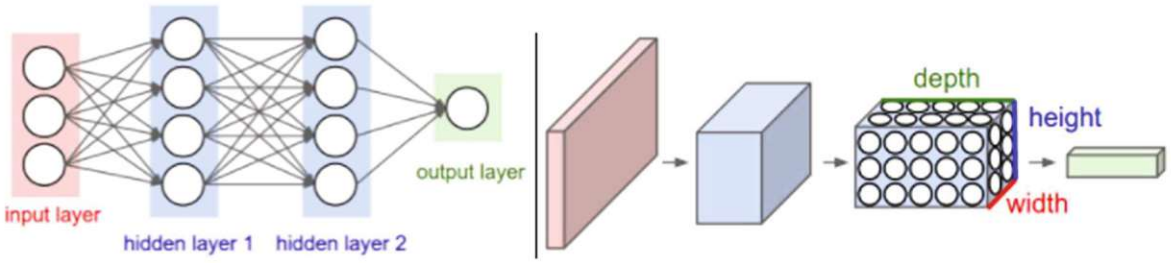


Abbildung 10: links: reguläres KNN; rechts: CNN [12]

CNN benutzt bei der Verarbeitung des *Inputs* eine Sequenz an verschiedenen *Layern*: *Convolutional Layer*, *Pooling Layer*, *Activation Layer* und Fully *Connected Layer* [12]. Auf die verschiedenen *Layers* wird in der Folge eingegangen.

2.4.1 Convolutional Layer

Der *Convolutional Layer* (*ConvLayer*) bildet den Kern des CNN. Die Hauptaufgabe dieses *Layers* ist es, die relevanten bzw. wichtigen Merkmale eines *Inputs* zu erkennen, damit nur diese zur Erkennung von Objekten verwendet werden. Hierbei wird ein *Input*-Bild mit einem Filter, eine mxm Matrix, Schritt für Schritt gescannt und ein kleinerer *Output* erzeugt. Der verwendete Filter wird auch *Kernel* genannt und dessen Größe kann beliebig gewählt werden (immer eine ungerade Anzahl an Reihen und Spalten) [13]. Die Einträge im Kernel entsprechen den *weights*, welche durch

Backpropagation (siehe Kapitel 2.3.4) aktualisiert werden [14]. Ein Beispiel für die Funktionsweise eines *ConvLayers* ist in Abbildung 11 zu sehen. Für jeden Farb-*Channel* (RGB) wird eine *Convolution* mit einem Filter der Größe 3x3 und mit dem *Stride* 1 vorgenommen. Der *Stride* gibt an, um wie viele Positionen der Filter nach einer Operation horizontal bzw. vertikal weiterrückt. Der *Input Channel* #1 (Red) ist rot hinterlegt, und das Feld, welches in diesem Moment durch den Filter bearbeitet wird, ist ebenso rot hinterlegt. Das *Output*-Ergebnis dieses 3x3 Feldes ergibt sich durch die Summe der Multiplikation der einzelnen Felder (Matrixmultiplikation). Dies wird analog für alle *Channels* durchgeführt und davon die Summe gebildet. Zusätzlich kann noch ein *Bias* Term hinzugefügt werden. Diese Summe entspricht dem *Output* dieser *Convolution.* Anschließend rückt der Filter, je nach *Stride*, eine Spalte weiter nach rechts. Dieser Prozess wird so lange wiederholt, bis der gesamte *Input* gefiltert ist. Rechts in der Abbildung 10 ist der *Output* mit den Ergebnissen der *Convolution* zu sehen. [13]

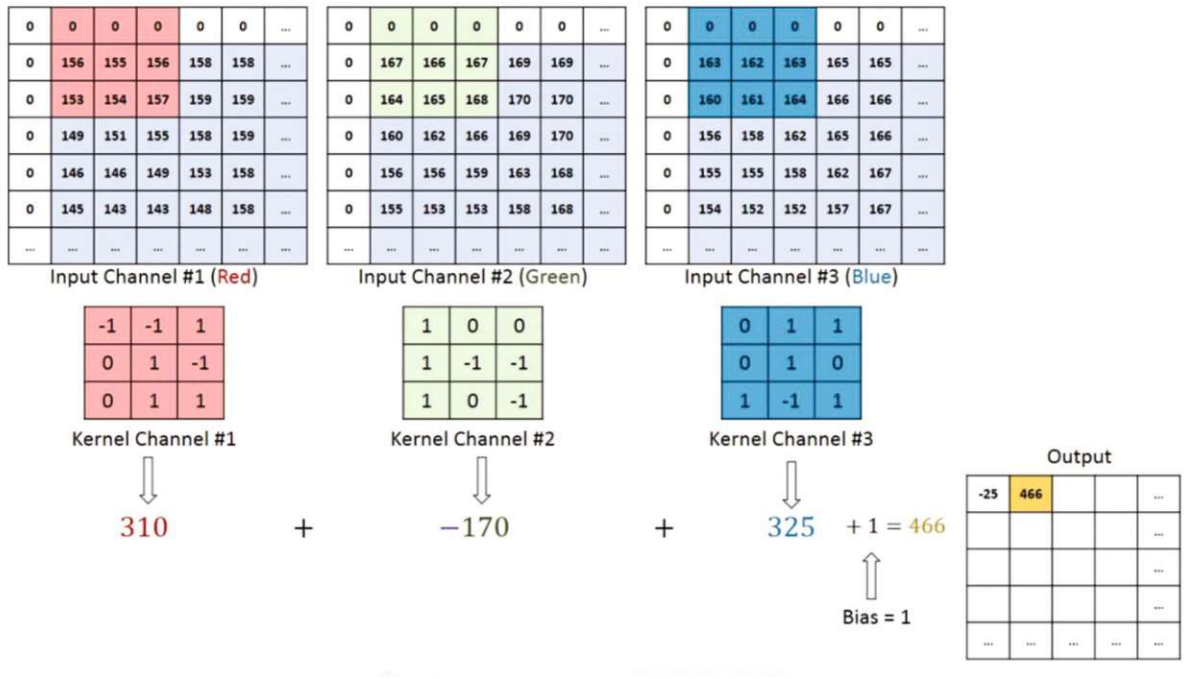


Abbildung 11: Funktionsweise eines *ConvLayer* [13]

Das Ergebnis eines *ConvLayers* wird auch als *Feature-Map* bezeichnet. Als *Features* werden die Merkmale verstanden, die am Bild erkannt werden. Das Erkennen dieser Merkmale ist die Hauptaufgabe von *ConvLayers*. Im ersten *ConvLayer* werden „grobe" Merkmale, wie z.B. Ecken oder Farben, erkannt. Sobald mehrere *ConvLayers* hintereinander arbeiten, werden mehrere Details erkannt. Somit wird dem Netzwerk die Fähigkeit gegeben, die Bilder in einem Datensatz zu verstehen. [13]

Nach einem *ConvLayer* wird eine *Activation Function* (Kapitel 2.2.1) verwendet, um nur bestimmte Signale weiterzuleiten. [13]

2.4.2 Pooling Layer

Es ist üblich, zwischen den einzelnen *ConvLayer* zusätzlich *Pooling Layer* zu verwenden. *Pooling Layer* funktionieren ähnlich wie *ConvLayer* und sind dafür zuständig, die Größe der *Feature-Map* zu reduzieren. Grund dafür ist, dass durch die Minimierung der Dimensionen (Verringerung der Signale) die erforderliche Rechenleistung verringert wird. Weiters ist das *Pooling* ein Instrument, um die wichtigsten Merkmale zu extrahieren. Es gibt zwei verschiedene Arten von *Pooling Layers*, den *Max Pooling Layer* und den *Average Pooling Layer*.

Der *Max Pooling Layer* benutzt wie der *ConvLayer,* einen Filter und filtert von der betrachteten Stelle den größten Wert heraus. In Abbildung 12 (links) ist ein Beispiel zu sehen, in welchem ein Filter der Größe 2x2 auf einen *Input* von 4x4 verwendet wird. Gleich wie beim *ConvLayer*, kann ein *Stride* gewählt werden. In diesem Beispiel wurde der *Stride* = 2 gewählt. In jeden 2x2 Bereich wird während des Filterns nur der maximale Wert übernommen. Im oberen linken Bereich (rotes Quadrat) wird somit nur der Wert 20 weiter übernommen.

Im Gegensatz zum *Max Pooling Layer* wird beim *Average Pooling Layer* der Durchschnittswert des betrachteten Bereiches übernommen. Im oberen linken Bereich (rotes Quadrat) wird somit der Durchschnittswert von 13 übernommen.

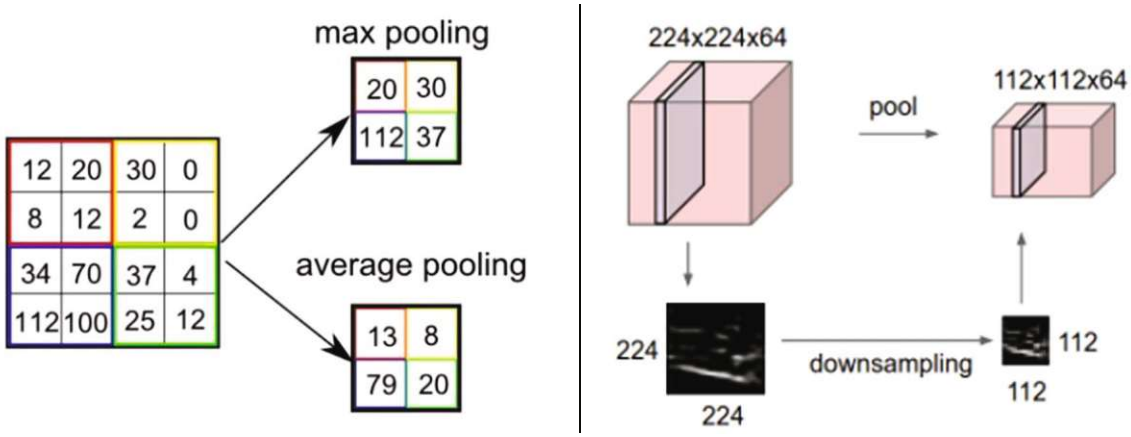


Abbildung 12: Funktionsweisen der verschiedenen *Pooling Layers* [13] [12]

Auf der rechten Seite der Abbildung 12 ist ein Volumen dargestellt, welches durch den Filter von 2x2 und einen *Stride* von 2 komprimiert wurde. Daraus ergibt sich bei einem *Input* von 224x224x64 ein *Output* von 112x112x64. Für genaue Berechnungsschritte wird auf die Fachliteratur verwiesen. [12] [13]

2.4.3 Fully Connected Layer

Ein CNN besteht im Regelfall aus einer Aneinanderreihung von jeweils einem *ConvLayer* und einem darauffolgenden *Pooling Layer.* Am Ende dieser Operationen wird noch ein *Fully Connected Layer* angehängt, welcher für die Klassifikation der Objekte zuständig ist. Kurz vor diesem *Layer* gilt es noch, die gefilterten Merkmale aus dem *Input*-Bild zu *Flatten (*siehe Abbildung 13 blaue Punkte*)*, damit diese für den *Fully Connected Layer* brauchbar sind [12]. *Flatten* bedeutet, dass der dreidimensionale Output eines *Layers* in Vektorform umgewandelt wird [13]. Der Output des *Fully Connected Layers* entspricht keiner Wahrscheinlichkeit, sondern irgendwelchen Zahlen. Diese werden mittels einer *Softmax-function* zwischen Null und Eins gebracht (normalisiert) und entsprechen dann den Wahrscheinlichkeiten. [9]

Ein CNN besteht daher aus den oben genannten *Layers* und kann einen Aufbau wie in Abbildung 13 besitzen. Es sei hier angemerkt, dass ein CNN in der Praxis normalerweise deutlich mehr *Layer* besitzt.

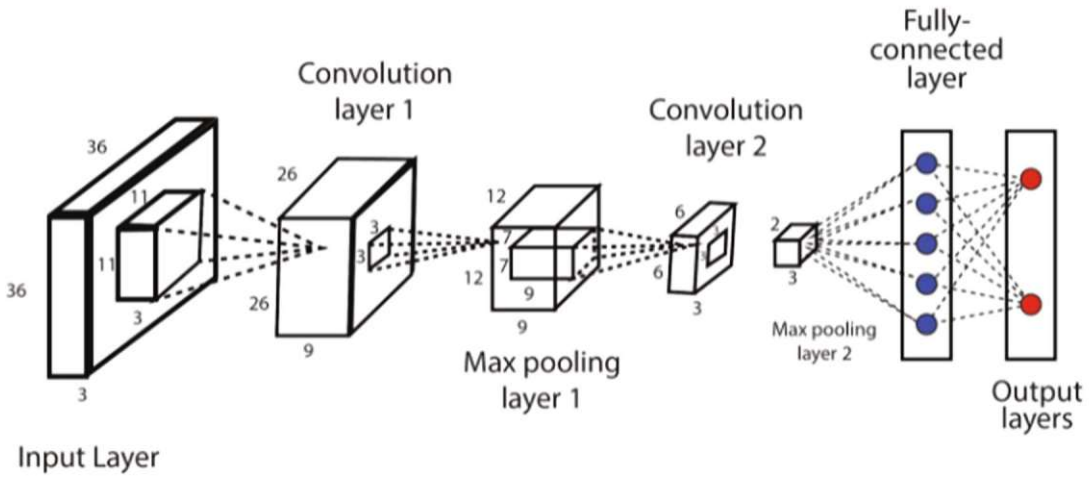


Abbildung 13: Aufbau eines CNN mit den verschiedenen *Layers* [15]

2.5 Transfer Learning

Ein OD-Modell benötigt, um gute Leistungen zu erbringen im Trainingsprozess einen großen Datensatz an Inputs. Der Prozess, Daten zu sammeln und anschließend zu *Labeln,* kann sehr aufwändig sein. Zusätzlich erhöht sich der Zeitaufwand beim Trainingsprozess, wenn das OD-Modell von Grund auf ohne Vorkenntnisse trainiert werden muss. Um diese Schritte zu beschleunigen, gibt es eine Lernmethode, die als *Transfer Learning* bezeichnet wird. Beim *Transfer Learning* wird ein bereits trainiertes NN (A) dazu verwendet, um bei einem neu zu trainierenden NN (B) Lernschritte

überspringen zu können. Dies bedeutet, dass bereits erlernte *Weights,* vom NN (A) benutzt werden, damit das NN (B) diese nicht mehr selbständig erlernen muss. [11]

In Abbildung 14 wird der *Transfer Learning*- Prozess dargestellt. Dabei wird ein NN mit Hilfe des *ImageNet* Datensatzes (z.B. *Tiny ImageNet*: 100.000 Bilder von 200 verschiedenen Objekten) trainiert. Dieses NN wird als *pre-trained* NN verwendet und bildet die Grundlage des tatsächlichen NN. Diese Grundlage entspricht dem ersten *Layer* des tatsächlichen NN und ist durch das Vortrainieren in der Lage, bereits Formen und Größen zu erkennen. Auf dieser Grundlage aufbauend werden anschließend mit dem eigenen Datensatz die weiteren *Layer* trainiert. Dieser Schritt wird auch als *Fine Tuning* bezeichnet. [16]

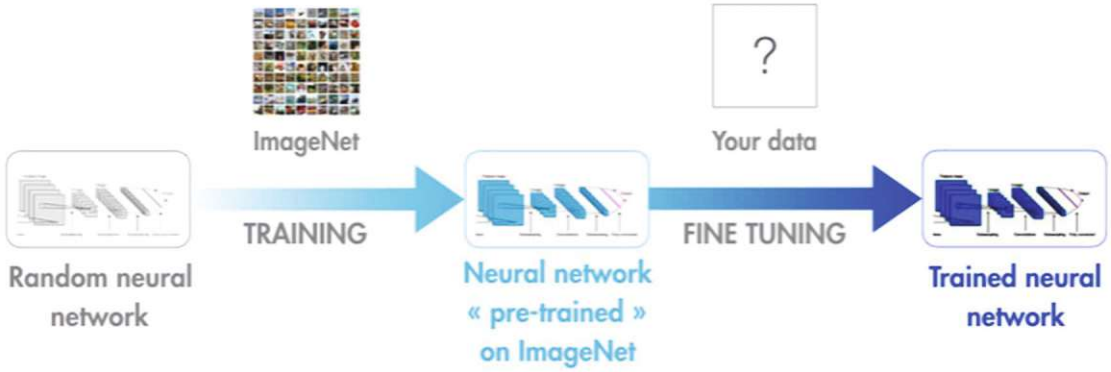


Abbildung 14: Anwendungsbeispiel für den *Transfer Learning*- Prozess [16]

2.6 Objekterkennung

Die Objekterkennung (OD) ist eine *Computer Vision* (CV) Technik, die angewendet wird, um Objekte in Bildern, Videos oder Livestreams zu erkennen. Dabei wird das erkannte Objekt mittels sogenannten *Bounding Boxes* (BB) umrahmt und so die Lokalisierung im Bildrahmen ermöglicht. OD wird häufig mit Bilderkennung (*Image Recognition* IR) verwechselt, obwohl die Art der Erkennung unterschiedlich ist. Bei der Bilderkennung wird einem Bild ein „Label“ zugeordnet. In Abbildung 15 ist der Unterschied zwischen den beiden Erkennungen ersichtlich. Dabei wird einem Bild, auf dem ein Hund abgebildet ist, bei der IR dem gesamten Bild das *Label* „Hund“ zugeschrieben. Bei der OD hingegen wird das Objekt im Bild selbst erkannt (in diesem Fall Hund) und mit einer BB und dem *Label* gekennzeichnet. [17]

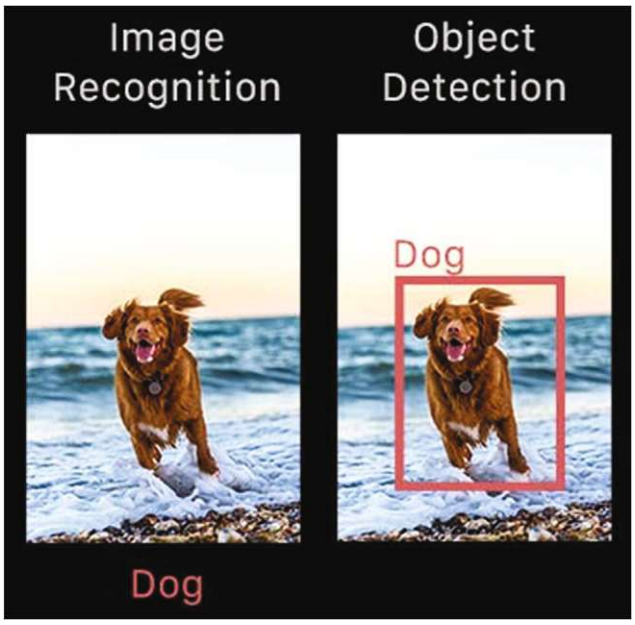


Abbildung 15: Unterschied *Image Recognition* (links) und *Object Detection* (rechts) [17]

Bei der OD trifft man auf zwei grundlegende Herausforderungen. Die Erste ist das Lokalisieren des Objektes auf dem gegebenen *Input* und die Zweite ist dessen Klassifizierung, d.h. um welches Objekt es sich handelt. Um diese Probleme lösen zu können, wird die Vorgehensweise eines OD-Modelles in drei Teile gegliedert [18]:

1. Informative Region Selection

Da verschiedene Objekte an verschiedenen Stellen des Inputs (Bild, Video, etc.) auftreten können, ist es wichtig, die sogenannten *Region Proposals* bzw. die *Region of Interests* (ROI) herauszufiltern, an denen sich die Objekte mit einer hohen Wahrscheinlichkeit befinden. Aufgrund der unterschiedlichen Größe und Form der Objekte, gestaltet sich dies als schwierige Aufgabe und nimmt viel Rechenleistung in Anspruch.

2. Feature Extraction

Damit Objekte erkannt werden können, müssen visuelle Merkmale aus dem *Input* extrahiert werden. Ein Beispiel für einen *Feature Extractor* wären *Convolutional Neural Networks* (CNN), welche im YOLO-Modell (siehe Kapitel 2.8.5) verwendet werden. Herausforderungen ergeben sich bei der Findung von Merkmalen durch unterschiedliche Beleuchtungsbedingungen und Hintergründe.

3. Classification

Zum Schluss gilt es noch, das erkannte Objekt zu klassifizieren. Eine dafür oft eingesetzte Methode ist die *Suport Vector Machine* (SVM). SVMs werden in dieser Arbeit nicht weiter erläutert.

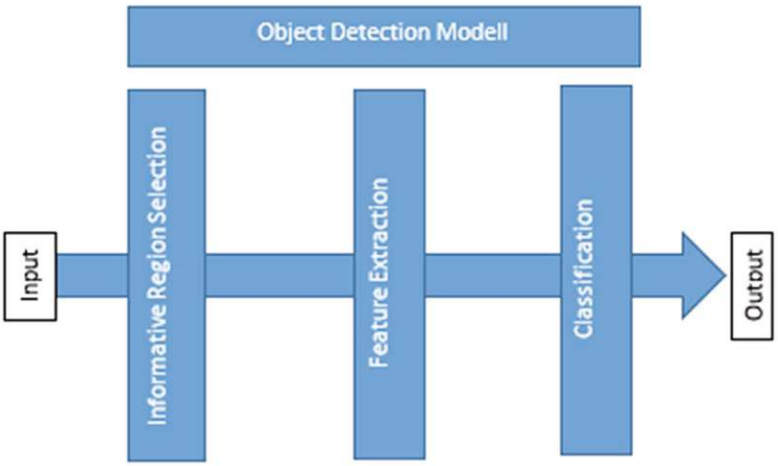


Abbildung 16: Grundlegender Aufbau eines OD-Modelles (in Anlehnung an [18])

2.7 Daten für OD

Die Daten, die als Input für ein CNN verwendet werden, besitzen im Regelfall mehrere *Channels* und können zusätzlich unterschiedliche Dimensionen besitzen. Die Datentypen unterscheiden sich bei der Einteilung in der Dimensionalität (1-D, 2-D und 3-D) und in deren Anzahl an *Channels*. Einige Beispiele sind in Tabelle 1 aufgelistet.

Tabelle 1: Beispiel verschiedener Datenformen (in Anlehnung an [11])

	Single-Channel	***Multi-Channel***
1D	Audiosignale	Skelett-Animationsdaten: Animation von Computer erzeugten Wesen
2D	Audiosignale die durch Fouriertransformation in einen 2D Tensor transformiert wurden	Farbbilder: bestehen aus den C*hannels* *red*, *green* und *blue* (RGB); Bilder einer Kamera
3D	Volumenförmige Daten: in diese Kategorie fallen z.B. CT-Scans	Farbvideos

Für diese Arbeit sind 2D- und 3D-Multi-Channel Daten von großer Bedeutung, da sich die Aufgabenstellung mit Objekterkennung auf Bildern und Videos beschäftigt.

2.8 OD-Modelle

In diesem Kapitel wird auf die verschiedenen Algorithmen eingegangen, die für OD eingesetzt werden. Dabei werden *Region-based* CNN und YOLO beschrieben, wobei das OD-Modell YOLO ausführlicher betrachtet wird, da dieses im Praxisteil der vorliegenden Arbeit in der Umsetzung verwendet wird.

Neben den oben genannten OD-Modellen gibt es noch weitere Modelle wie Single Shot Detector (SSD) [19], RetinaNet [20] und Region-based Fully Convolutional Network (R-FCN) [21], auf welche in der vorliegenden Arbeit nicht eingegangen wird.

2.8.1 *Region-based* CNN

Im folgenden Kapitel wird auf *Region-based* CNN für die OD eingegangen. Dabei werden die im Laufe der letzten Jahre entwickelten Varianten und dessen Verbesserungen aufgezeigt. Bei den *Region-based* CNN handelt es sich um sogenannte *two-stage detectors* oder *multi-stage detectors*, da diese im ersten Schritt die *Region of interest* bzw. *Region Proposals* definieren und in einem zweiten Schritt die Klassifizierung vornehmen. *Region Proposals* sind vorgeschlagenen BB, in der sich möglicherweise ein Objekt befinden kann. [22]

2.8.1.1 R-CNN

In der OD ist eine große Herausforderung, die Stellen im Bild erkennen zu können, sowie zu bestimmen an welcher Stelle sich das Objekt befindet, welches erkannt werden soll. Ein einfacher Ansatz wäre, die sogenannte *Region of Interest* (ROI) zu definieren und anschließend ein CNN für dessen Klassifikation zu verwenden. Das Problem dabei ist, dass die Objekte viele verschiedene Positionen und Formen haben können und die Zahl der ROI deshalb hoch sein kann. Aus diesem Grund haben Ross Girshick et al. [23] eine Methode entwickelt, die hilft, dieses Problem zu vermeiden. Da in diesem Modell *Region Proposals* mit CNN kombiniert werden, trägt es den Namen R-CNN.

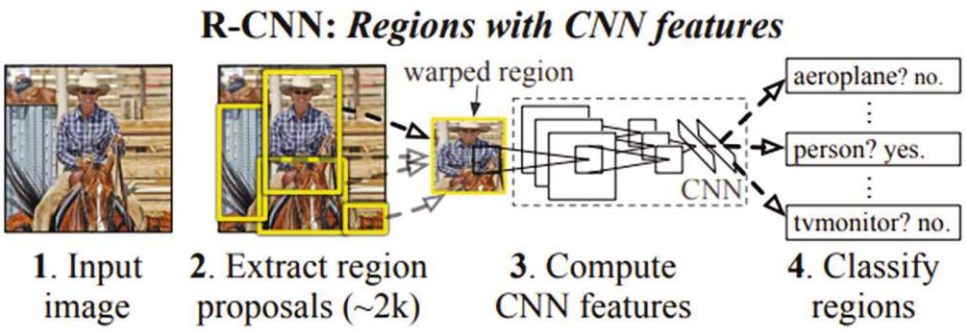


Abbildung 17: R-CNN Modellbeschreibung [23]

In Abbildung 17 ist die Funktionsweise des R-CNN Modelles in vier Schritten dargestellt. R-CNN benötigt ein *Input Image*. Im zweiten Schritt wird eine selektive Suche durchgeführt, um bis zu 2000 *Regions* vom *Input Image* zu extrahieren, die sogenannten *Regions Proposals*. Im Anschluss werden diese zu einem Format mit einer bestimmten Größe verarbeitet und in ein CNN eingespeist. Das CNN verarbeitet die weitergeleiteten Informationen und erstellt einen Vektor als dessen Output. Das CNN dient als *Feature Extractor* und gibt diese extrahierten Merkmale einer SVM (siehe Kapitel 2.6) weiter, die für das Klassifizieren der Objekte in den *Region Proposals* verantwortlich sind. Neben der Klassifizierung des Objektes berechnet der Algorithmus mögliche Abweichungen der BB und passt diese an das erkannte Objekt an [24]. R-CNN besteht somit aus drei separaten Modulen (Punkt 2-4 in Abbildung 17) und wird deshalb als *multi-stage model* bezeichnet. [25]

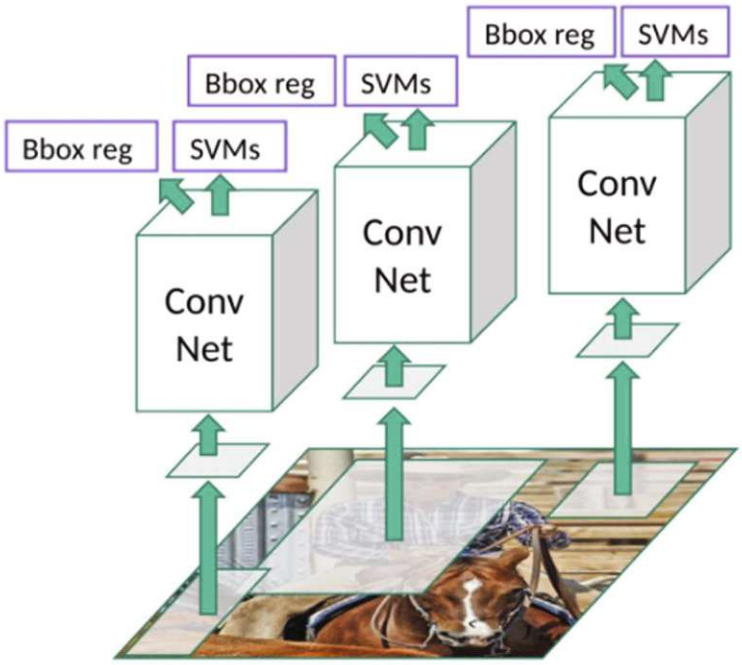


Abbildung 18: Schematischer Ablauf von R-CNN [24]

In Abbildung 18 wird die soeben beschriebene Vorgangsweise durch R-CNN aufgezeigt. Dabei folgt aus den verschiedenen *Region Proposals* die Verarbeitung der *Features* bis hin zur SVM. Neben der Klassifizierung werden noch BB Adjustierungen vorgenommen (Bbox reg). [24]

R-CNN bringt einige Probleme mit sich. Der größte Nachteil ist die geringe Geschwindigkeit des Modelles, da das Netzwerk immer insgesamt 2000 *Region Proposals* klassifizieren muss. Eine Anwendung in Echtzeit ist nicht erstrebenswert, da das Netzwerk rund 47 Sekunden für die Bearbeitung eines einzelnen Eingabebildes benötigt. [24]

2.8.1.2 *Fast* R-CNN

Um die Probleme von R-CNN zu beheben, präsentieren dieselben Autoren, Ross Girshick et al. [26], *Fast* R-CNN. In Abbildung 19 ist das Modell von *Fast* R-CNN dargestellt. Im Gegensatz zu R-CNN, bei welchem es drei separate Module (siehe Abbildung 17: 2, 3 und 4) gibt, werden bei *Fast* R-CNN alle in einem Schritt vollzogen. [25]

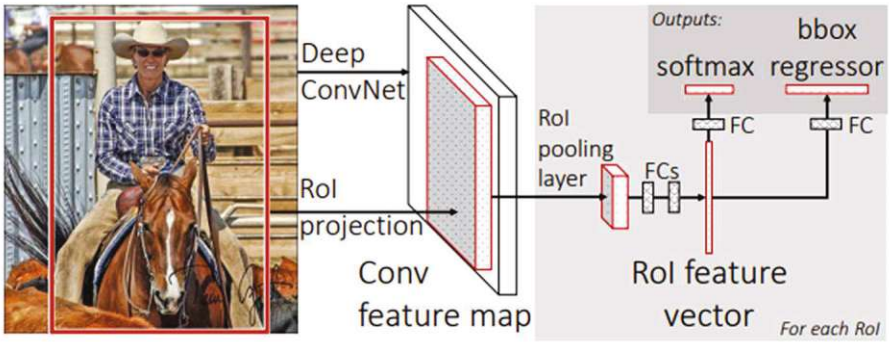


Abbildung 19: *Fast* R-CNN Modellbeschreibung [26]

Wie in Abbildung 19 zu sehen ist, ist der Aufbau von Fast R-CNN ähnlich wie der von R-CNN. Der Hauptunterschied zu R-CNN ist, dass die *Region Proposals* durch eine selektive Suche nicht vor dem CNN aus dem Bild extrahiert werden, sondern dass das Bild durch ein CNN geschickt wird und in weiterer Folge eine *Feature Map* generiert. Anhand dieser *Feature Map* werden mit Hilfe von *Pooling Layers* die *Regions Proposals* indentifiziert. Mittels *Fully Connected Layers* wird für jede *Region Proposal* die Wahrscheinlichkeit vorhergesagt und, wie bei R-CNN, die BB Adjustierung berechnet. Für die Normalisierung der Wahrscheinlichkeiten wird eine *softmax-function* verwendet (siehe Kapitel 2.4.3). [26] [24]

Da *Fast* R-CNN nicht immer 2000 *Region Proposals* ins CNN einspielen muss, ist dieser im Vergleich zu R-CNN bei der Objekterkennung erheblich schneller. [24]

2.8.1.3 *Faster* R-CNN

Die beiden genannten R-CNN Modelle verwenden ein selektives Suchen der *Region Proposals*, welches ein sehr zeitintensives Verfahren ist und die Leistung des Netzwerkes beeinträchtigt. Aus diesem Grund haben Shaoqing Ren et al. [27] einen OD-Algorithmus entwickelt, der dem Netzwerk ermöglicht, die *Region Proposals* zu erlernen. Wie in Abbildung 20 zu sehen, gibt es nur wenige Unterschiede zu Fast R-CNN. Das Netzwerk benutzt als Input ein Bild, welches durch das CNN verarbeitet wird. Dieses erstellt die *Feature Map*. Im Gegensatz zu den oben genannten Netzwerken werden nicht die *Region Proposals* durch eine selektive Suche ermittelt, sondern ein separates Netzwerk wird verwendet. Die vom *Region Proposal Network* RPN definierten *Region Proposals* werden durch einen *Pooling Layer* erneut verarbeitet und in weiterer Folge für die Objektidentifizierung und die Berechnung der *Offset*-Werte (Adjustierung) der BB verwendet. [24]

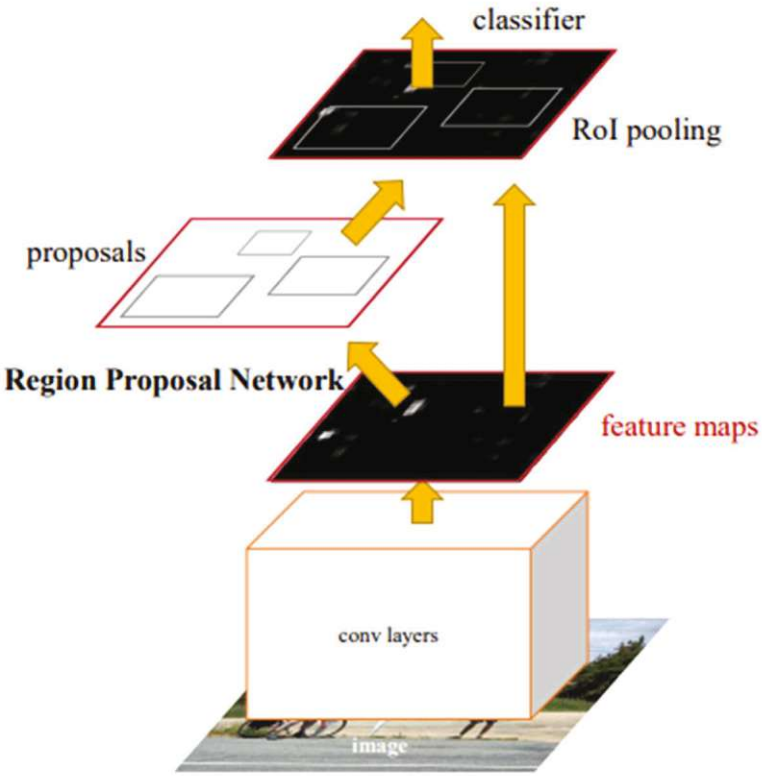


Abbildung 20: Schematischer Ablauf von *Faster* R-CNN [27]

Faster R-CNN besteht so gesehen aus zwei Teilen (*two-stage detector*): dem RPN Netzwerk und dem *Fast* R-CNN für die Klassifizierung bzw. die Definition der BB. Mit dieser Methode werden, hinsichtlich der Geschwindigkeit beim Erkennen von Objekten sehr gute Ergebnisse erzielt [27]. In Abbildung 21 werden die klaren Vorteile in Bezug auf die Testgeschwindigkeit aufgezeigt, wobei *SPP-Net* ein weiteres Modell ist, welches in dieser Arbeit nicht berücksichtigt wird.

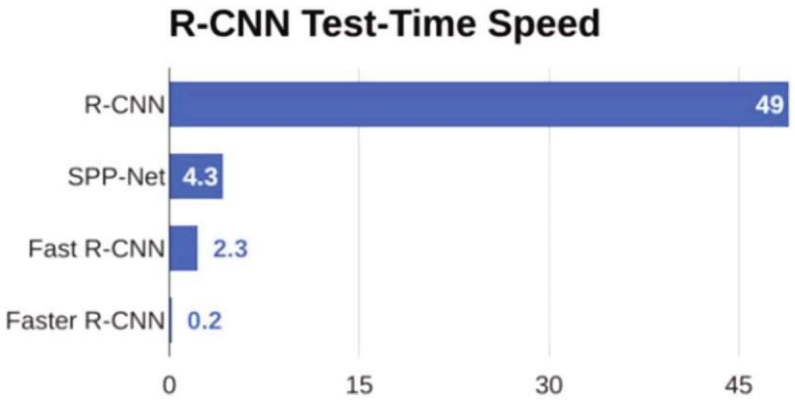


Abbildung 21: Gegenüberstellung der Testzeit in Sekunden aller R-CNN Modelle [24]

2.8.2 You Only Look Once (YOLO)

Zu den populärsten Algorithmen für OD weltweit zählt heutzutage YOLO. Laut den Entwicklern *Facebook AI* wird dieser, aufgrund seiner einheitlichen Architektur, als sehr schnell in seiner Art beschrieben [28]. 2016 stellte Joseph Redmon die erste Version von YOLO in seinem Paper vor [29]. Aufbauend auf der ersten Version von YOLO, folgten in den darauffolgenden Jahren weitere Versionen, bei denen erhebliche Verbesserungen in Bezug auf Genauigkeit und Schnelligkeit erzielt wurden. Joseph Redmon entwickelte die Nachfolgeversionen YOLOv2 und YOLOv3. Die letzte und aktuellste Version war zum Zeitpunkt des Verfassens dieser Arbeit YOLOv4. Diese wurde von Alexey Bochkovskiy et al., im Paper „YOLOv4: Optimal Speed and Accuracy of Object Detection“, vorgestellt [30]. Der Name YOLO, *You Only Look Once* ergibt sich aus der Funktionsweise des Modelles, da das Eingabebild nur einmal betrachtet wird [28]. YOLO zählt zu den *one-stage detectors*, da das Definieren der BB und die Objektklassifizierung in einem Schritt vorgenommen wird. [22]

In weiterer Folge wird auf die einzelnen Versionen von YOLO eingegangen.

2.8.2.1 YOLOv1

Die Funktionsweise von YOLO ist in drei Phasen unterteilt. In der ersten Phase wird das Eingabebild skaliert (448x448), in der zweiten Phase durchläuft das Bild ein CNN und in Phase drei werden die daraus resultierenden Erkenntnisse mit Wahrscheinlichkeiten bewertet (siehe Abbildung 22).

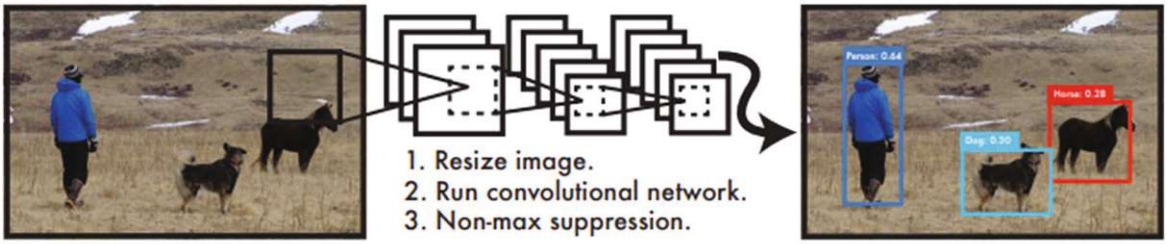


Abbildung 22: Phasen von YOLO [29]

In YOLO werden alle einzelnen Komponenten der OD in einem einzigen Netzwerk zusammengeführt, d.h. es werden, nicht wie bei den *Region-based* OD-Modellen, ROI's separat erstellt. Das Netzwerk verwendet dabei Merkmale aus dem gesamten Bild, um BB vorherzusagen. Diese Vorhersage ist nicht nur für ein Objekt möglich, sondern es werden für alle erkennbaren Objekte (Klassen) auf dem gesamten Bild BB vorhergesagt. Kurz gesagt, das Netzwerk betrachtet das Bild global, wobei alle Objekte beurteilt werden [29]. Um Objekte vorhersagen zu können, teilt YOLO das *Input*-Bild in SxS Teile auf (siehe Abbildung 23). Diese werden auch *Grid Cells* genannt.

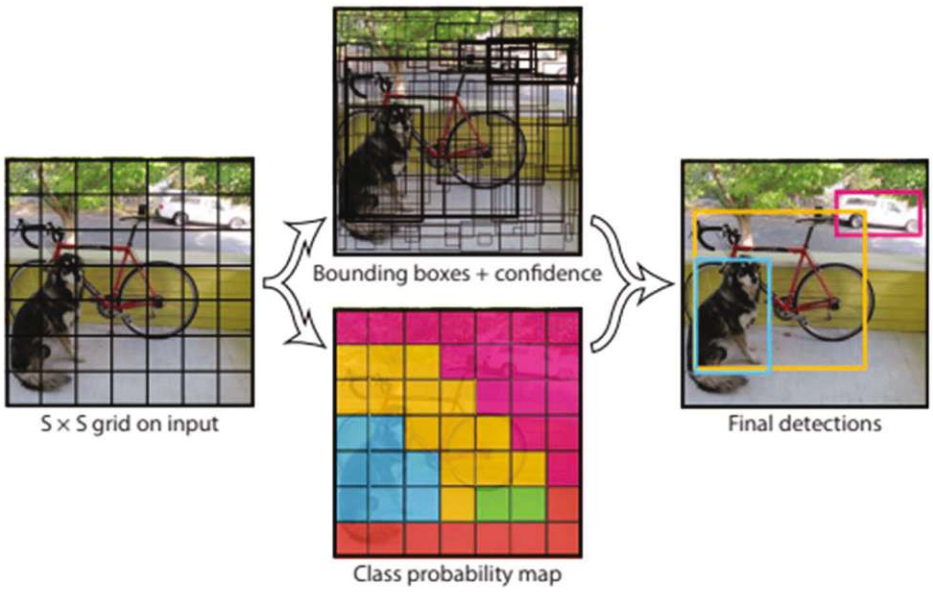


Abbildung 23: Darstellung des Vorganges für das Vorhersagen der *Bounding Boxes* [29]

In jeder *Grid Cell* werden B (Anzahl der BB) *Bounding Boxes,* dessen Koordinaten x,y,w,*h* und die Wahrscheinlichkeiten (*Confidence Scores*) vorhergesagt. Die Koordinaten x und y entsprechen dem Mittelpunkt und w (*with*) und h (*height*) entsprechen der Breite bzw. der Höhe der BB. Die *Confidence Score* Pr(Object) gibt an, wie hoch die Wahrscheinlichkeit ist, dass sich ein Objekt in dieser befindet (siehe abbildung 23). Redmon et al. [29] definieren in ihrem Paper die Wahrscheinlichkeit, ob sich ein Objekt in einer BB befindet und wie hoch diese ist, aus dem Produkt von Pr(Object) und IOU (*Intersection Over Union*)(siehe Formel 4). Aus dieser Gleichung ergibt sich, wenn kein Objekt erkannt wurde (*Pr(Object)=0),* dass die Wahrscheinlichkeit, dass ein Objekt (Klasse) enthalten ist, gleich Null sein muss. [29]

$$\Pr(Object) * IOU \tag{4}$$

Definition IOU:

Die *Intersection Over Union* (IOU) ist eine Methode, um die OD während des Trainings zu bewerten, indem die Genauigkeit der erstellten BB im Vergleich zu der GTB errechnet wird. Hierbei wird die überlappende Fläche beider Boxen zur Bewertung herangezogen. In Abbildung 24 wird dargestellt, wie die IOU zwischen den beiden Boxen errechnet wird. Die überlappende Fläche wird durch die gesamte Fläche beider Boxen dividiert. B1 entspricht dabei der GTB und B2 der BB die das OD-Modell erstellt hat. Es kann vorkommen, dass das OD-Modell während des Erkennens (nach dem Training) mehrere BB für ein Objekt erstellt. Durch die Angabe eines Schwellenwertes,

der IOU zwischen den erkannten BB, kann angegeben werden, ab was für einem Wert eine BB als „richtige“ Vorhersage interpretiert wird. Somit kann verhindert werden, dass für ein Objekt mehrere BB am Bild angezeigt werden (siehe Abbildung 23). Dieses Filtern der „besten“ BB für die Vorhersage wird als *Non Maximum Suppression* NMS bezeichnet. [31]

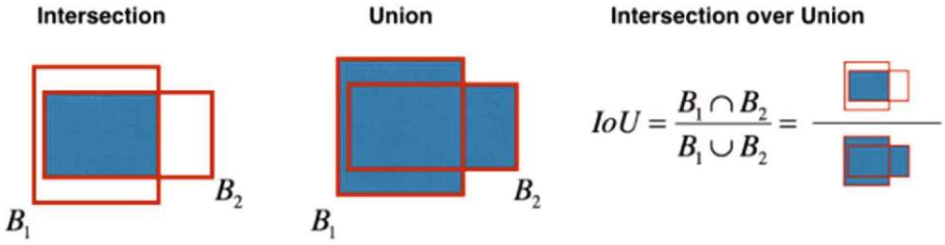


Abbildung 24: Erklärung der *Intersection Over Union* IOU [32]

In jeder *Grid Cell* wird auch die *Class Probability* Pr(Classi/Object) angegeben. Diese Wahrscheinlichkeit gibt an, um welche Art von Objekt (z.B. Katze, Hund etc.) es sich in der BB handelt (siehe Abbildung 23). Diese Wahrscheinlichkeit ist davon abhängig, ob sich ein Objekt in der *Grid Cell* befindet und wird nur einmal pro *Cell* angegeben. In Abbildung 23 ist zu sehen, wie die prognostizierten BB am Bild vorausgesagt werden und durch NMS nur die „Besten“ davon verwendet und am Ausgabebild abgebildet werden. Um die endgültige Erkennungswahrscheinlichkeit zu erhalten, werden die Wahrscheinlichkeiten Pr(Classi/Object), Pr(Object) mit der IOU multipliziert (siehe Formel 9). [29]

$$\Pr(Class_i/Object) * \Pr(Object) * IOU = \Pr(Class_i) * IOU \quad (5)$$

Der genaue Aufbau von YOLO ist in Abbildung 25 zu sehen.

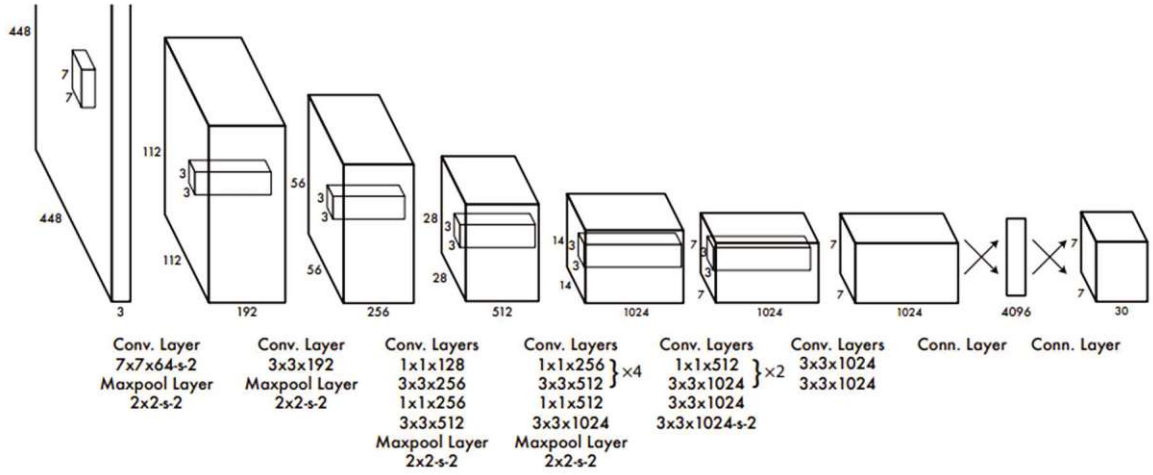


Abbildung 25: Architektur von YOLO [29]

Das Netzwerk besteht aus 24 *Convolutional Layers* die das CNN (Definition siehe Kapitel 2.4) darstellen. Am Ende jeder *Convolution* (als Blöcke in Abbildung 25 dargestellt) befindet sich ein *Maxpool Layer*, der die wichtigsten Informationen filtert und nur diese weitergibt. Am Ende des Netzwerkes werden 2 *Fully Connected Layer* eingesetzt, die für die endgültige Klassifizierung zuständig sind. YOLO wurde am Datensatz PASCAL VOC evaluiert. Hierbei wurden die Bilder in 7x7 *Grid Cells* unterteilt und pro *Cell* B=2 BB verwendet. Da der Datensatz von *PASCAL VOC* C=20 Klassen enthält, ergibt sich die finale Voraussagung als ein 7x7x30 Tensor (siehe Abbildung 25 letzter Block). [29]

Im Allgemeinen ergibt sich bei dem *Output* von YOLO ein Tensor der Form (siehe Abbildung 25):

$$SxSx(B * (5 + C)) \tag{6}$$

Hier entspricht S der Anzahl der *Grid Cells*, in die das Bild unterteilt wurde, B der verwendeten BB und C der Anzahl der Klassen, die im Datensatz enthalten sind. [33]

Während des Trainings von YOLO wird bei jeder *Iteration* mit Hilfe einer *Loss- Function* die Aussagekraft bewertet. Am Anfang des Trainings ist der Ausgabewert der Funktion hoch. Das Ziel ist es, einen möglichst kleinen Wert zu erzielen. Die *Loss-Function* sieht in YOLO wie folgt aus:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \tag{7}$$

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \right] + \tag{8}$$

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{obj} \left(C_i - \hat{C}_i\right)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{noobj} \left(C_i - \hat{C}_i\right)^2 + \tag{9}$$

$$\sum_{i=0}^{S^2} 1_i^{obj} \sum_{c \, \epsilon \, classes} (p_i(c) - \hat{p}_i(c))^2 \tag{10}$$

Die Funktion setzt sich aus den verschiedenen Summen der quadratischen Fehler (*Sum Squared Error*,SSE) zusammen (siehe Formel (7)-(10)). Der erste Teil der *Loss-Function* betrachtet die Koordinaten x, y und die Maße w, h der BB. Der zweite Teil beinhaltet die Wahrscheinlichkeit, dass sich ein Objekt in der BB befindet. Im letzten Teil wird die Klassenzugehörigkeit betrachtet, falls ein Objekt in der BB enthalten ist. Die Variable $1_{i,j}^{obj}$ bzw. 1_i^{obj} nimmt den Wert Eins an, wenn ein Objekt enthalten ist, ansonsten liegt dieser bei Null. $1_{i,j}^{noobj}$ nimmt den Wert Null an, wenn ein Objekt enthalten ist, ansonsten den Wert Eins. λ_{coord} und λ_{noobj} werden benötigt, um das Modell stabil halten zu können. Diese werden laut Autoren auf Fünf bzw. auf 0,5 gesetzt. [29]

Die erste Version von YOLO ist in einigen Hinsichten eingeschränkt. Diese Einschränkungen sind in Tabelle 2 aufgelistet.

Tabelle 2: Einschränkungen von YOLO (in Anlehnung an [29])

Einschränkung	Grund
Nur ein Objekt kann pro *Grid Cell* erkannt werden	Das Modell verwendet einen F*ully Connected Layer,* der das Vorhersagen stark räumlich einschränkt. Wie bereits erwähnt werden pro *Grid Cell* 2 BB vorhergesagt, aber eine *Grid Cell* kann nur 1 Objekt vorhersagen. D.h. wenn in einer *Grid Cell* z.B. ein Hund und eine Katze sichtbar sind, wird nur eines der beiden erkannt.
Nur grobe Merkmale werden erkannt	Da das Modell das Eingabebild in SxS *Grid Cells* unterteilt und d*owngesampelt* wird, können BB nur grobe Merkmale erkennen.

Schwierigkeiten beim Erkennen von kleinen Objekten	Zu wenig *Layer* um Details zu filtern

2.8.2.2 YOLOv2

Im Jahr 2017 präsentiert Redmon et al. [34] die zweite Version von YOLO. In der neuen Version von YOLO wurde daran gearbeitet die Einschränkungen der ersten Version zu eliminieren und gleichzeitig die Geschwindigkeit und Genauigkeit des Modelles zu erhöhen.

Ein neues Element, welches in YOLOv2 eingeführt wurde, nennt sich *Batch Normalization*. *Batch Normalization* wird nach jedem *Convolutional Layer* angewandt und ist für die Stabilisierung (*Re-Scaling, Re-Centering*) des Modelles zuständig. Ein weiterer Vorteil ist, dass das CNN dadurch genauer wird (mAP +2%, siehe Kapitel 4.5). [34]

In YOLOv1 werden die Koordinaten der BB mit Hilfe der *Fully Connected Layers* (siehe Abbildung 25) direkt nach den *Convolutional Layers* berechnet bzw. vorhergesagt. In YOLOv2 werden die *Fully Connected Layers* weggelassen und das Vorhersagen der BB wird mit Hilfe sogenannter *Anchor Boxes* (AB) vollzogen. Im Gegensatz zu YOLOv1 werden AB mit unterschiedlicher Größe vordefiniert. Diese müssen somit nicht mehr vom Algorithmus selbst bestimmt bzw. berechnet werden. Das neue YOLO-Modell berechnet *Offset*-Werte zu der GTB und nähert die AB auf diese an. Mit dieser Methode wird das Modell beim Training stabiler. Nachteil bleibt immer noch der, dass die AB zu Beginn selbst definiert werden müssen und somit eine große Abweichung zu der tatsächlichen Größe dieser ergeben kann. [34]

Um die Dimensionen der AB nicht selbst bestimmen zu müssen wird *K-Means Clustering* am Trainingsdatensatz angewendet. Mit dieser Vorgehensweise werden AB anhand der im Datensatz definierten GTB definiert. In Abbildung 26 (rechte Seite) ist ersichtlich, wie durch *K- Means Clustering* an den Datensätzen *COCO* (blau) und *VOC* (schwarz umrahmt) mit Hilfe der GTB verschiedener Objekte die Geometrie der AB aussieht. In der Abbildung 26 ist auf der linken Seite der Zusammenhang von IOU und Anzahl der AB (*#Clusters*) zu sehen. Weiters kann die Anzahl (#Clusters) der zu erstellenden AB beliebig gewählt werden, wobei die Autoren einen Wert von Fünf heranziehen. [34]

Bsp.: Angenommen auf einem Bild sind ein Mensch und ein Auto abgebildet, wobei bekannt ist, dass die Form des Menschen eines aufrechten Rechtecks und die Form des Autos eines liegenden Rechteckes entspricht. Anhand dieser Information wird dem Netzwerk das Lernen erleichtert, um die zwei Objekte unterscheiden zu können.

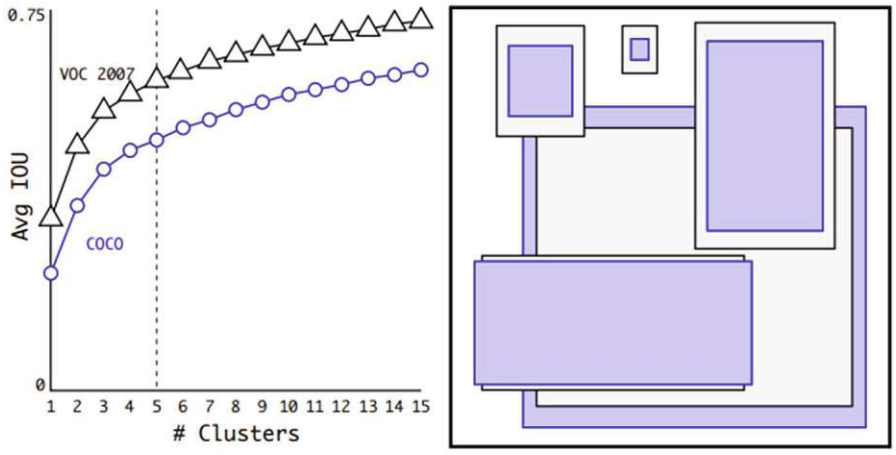


Abbildung 26: *K-Clustering* in YOLOv2 [34]

Eine weitere Neuheit in YOLOv2 ist, dass das Eingabebild auf 416x416 (siehe Abbildung 29) verkleinert wird. Dies hat den Vorteil, dass die Anzahl der *Grid Cells* immer ungerade ist und sich somit eine einzige zentrale *Grid Cell* bildet. Da vor allem große Objekte ihren Mittelpunkt in der Mitte eines Bildes haben, ist das ein Vorteil bei der Erkennung, da keine umliegenden *Grid Cells* betrachtet werden müssen. [34]

YOLOv2 macht Vorhersagungen über die Abweichungen (*Offsets*) zwischen AB (Abbildung 27, punktiertes Rechteck) und BB (Abbildung 27, blaues Rechteck). Als erstes wird die *Grid Cell* definiert, zu welcher die linke obere Ecke der AB gehört. Hierbei werden die Werte Cx und Cy (siehe Abbildung 27) sowie die Abmaße Pw und Ph der AB erzeugt. Im nächsten Schritt werden die sogenannten *Offsets* der BB (tx,ty,tw,th) vorhergesagt. Mit Hilfe der *Sigmoid*-Funktion werden diese Werte parametrisiert und fallen zwischen 0 und 1. Die endgültigen Koordinaten bzw. Abmaße der BB sind die Werte bx,, by, bw und bh. Wobei bx/ by die Ortskoordinaten und bw/ bh die Abmaße der BB darstellen (siehe Abbildung 27). [35]

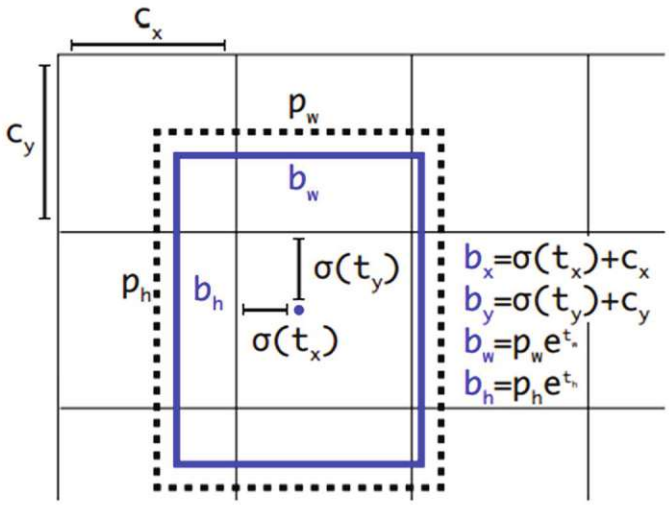


Abbildung 27: Direkte Voraussagung des Standortes einer BB [34]

Mit dem Verwenden von *K-Clustering* und der Berechnung von *Offsets* zwischen AB und BB, wird die Genauigkeit von YOLOv2 um 5% erhöht. [35]

Ein weiterer Unterschied gegenüber YOLOv1 ist die Architektur des Modelles (Abbildung 28).

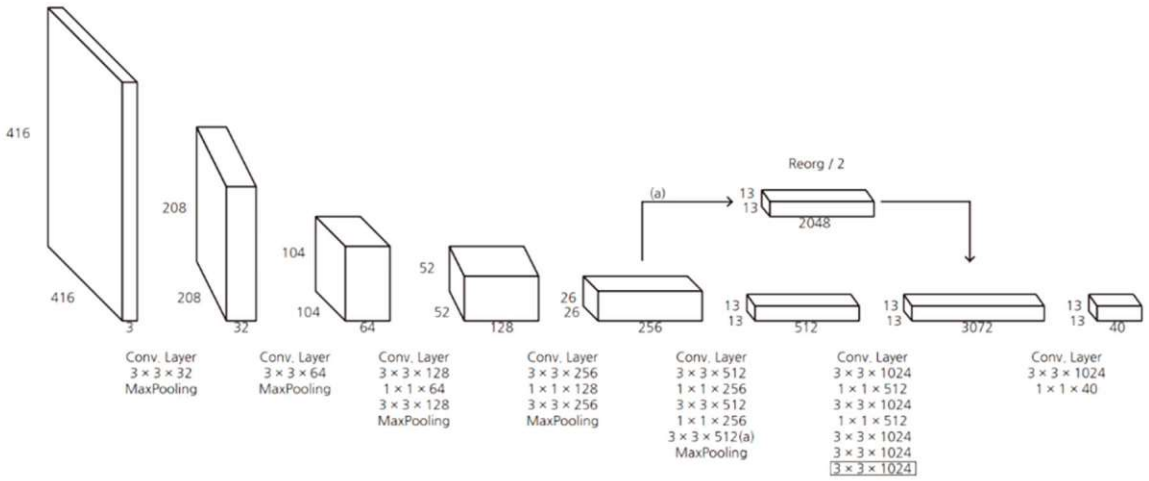


Abbildung 28: Architektur von YOLOv2 [36]

Das neu angewendete Netzwerk gilt als *Backbone* von YOLOv2 und wird *Darknet-19* genannt. Viele OD-Methoden verwenden VGG-16 (spezielles CNN) als *Feature Extractor*. Dieses benötigt aufgrund seiner Komplexität 30,69 Milliarden Operationen und ist infolgedessen langsamer als *Darknet-19* (siehe Abbildung 29) mit 8,52 Operationen. [35]

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 29: Darstellung vom *Backbone Darknet-19* welches in YOLOv2 verwendet wird [34]

Da sich der *Output* des Modelles in YOLOv2 vom *Output* in YOLOv1 unterscheidet, verändert sich auch die *Loss-Function*. Im Paper von Redmon et al. [34] wird mathematisch nicht auf die *Loss-Function* eingegangen. Diese kann aber an bereits implementierten Modellen extrahiert werden (*Darkflow*). [37]

2.8.2.3 YOLOv3

2018 stellten Redmon et al. [38] die nächste verbesserte Version von YOLO vor. Wie im Paper beschrieben, wurden kleine, aber effektive, Änderungen am Modell vorgenommen. YOLOv3 hat sich dabei vergrößert, ist aber dementsprechend genauer und schneller geworden. Im Vergleich zu den Vorgängerversionen wird die Erkennung an mehreren Stellen im Netzwerk vorgenommen. Das Erkennungsnetzwerk (*Feature Extractor*) ist stabiler und es wurden kleine Änderungen in der *Loss-Function* vorgenommen.

Die wohl größte Veränderung betrifft die Struktur des Netzwerkes. YOLOv2 verwendet *Darknet-19* als *Backbone,* welches als *Feature Extractor* dient. Dieses verwendet 19 *Layer*. Dieses Netzwerk hat jedoch Probleme damit, kleine Objekte zu erkennen. Das neue Netzwerk, welches in YOLOv3 als Backbone verwendet wird, heißt *Darknet-53*. Dieses besteht aus 53 Layer. Für die OD werden zusätzlich 53 *Layer* (*detection head*) darauf gestapelt, wodurch sich insgesamt 106 *Layer* für das gesamte Netzwerk ergeben [39]. In Abbildung 30 ist der Aufbau von *Darknet-53* zu sehen. Alle *Layer,* die sich in den Rechtecken befinden, bilden die sogenannten *Residual Blocks.* Das gesamte Netzwerk ist eine Kette aufeinanderfolgender *Residual Blocks,* die mit *Stride* 2 *Convolutional Layer* dazwischen verbunden sind, um die Größe des Inputs zu dimensionieren. In den Blöcken sind nur 1x1 und 3x3 C*onvolutional Layer* und eine *Skip Connection* vorhanden [40]. Durch das Verwenden von *Skip Connections* ist es möglich, einige *Layer* zu überspringen und den Output des ersten *Layers* als *Input* weitergeben zu können. Es ist empirisch bewiesen, dass mit dieser Methode die *Performance* des Netzwerkes gesteigert wird. [41]

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

Abbildung 30: Darstellung vom *Backbone Darknet-53* welches in YOLOv3 verwendet wird [38]

Eine weitere Änderung ist, dass keine *Maxpool Layer* verwendet werden. Stattdessen werden am Ende ein *Avgpool* Layer und eine *Softmax function* hinzugefügt.

Das Netzwerk (Abbildung 31) von YOLOv3 macht Voraussagungen bei drei verschiedenen Skalierungen (*Scales*). Die Stellen im Netzwerk sind an den Positionen 82, 94 und 106. Der Grund für die OD an drei verschiedenen Stellen ist die Größe des *Outputs*. Hier können bei der kleinsten Skalierung große Objekte und bei der größten Skalierung kleine Objekte sehr gut erkannt werden. Das Format 13x13 (Postion 82) ist für das Erkennen von großen, das Format 26x26 (Postion 94) für mittlere und das Format 52x52 (Postion 106) für kleine Objekte verantwortlich. [39]

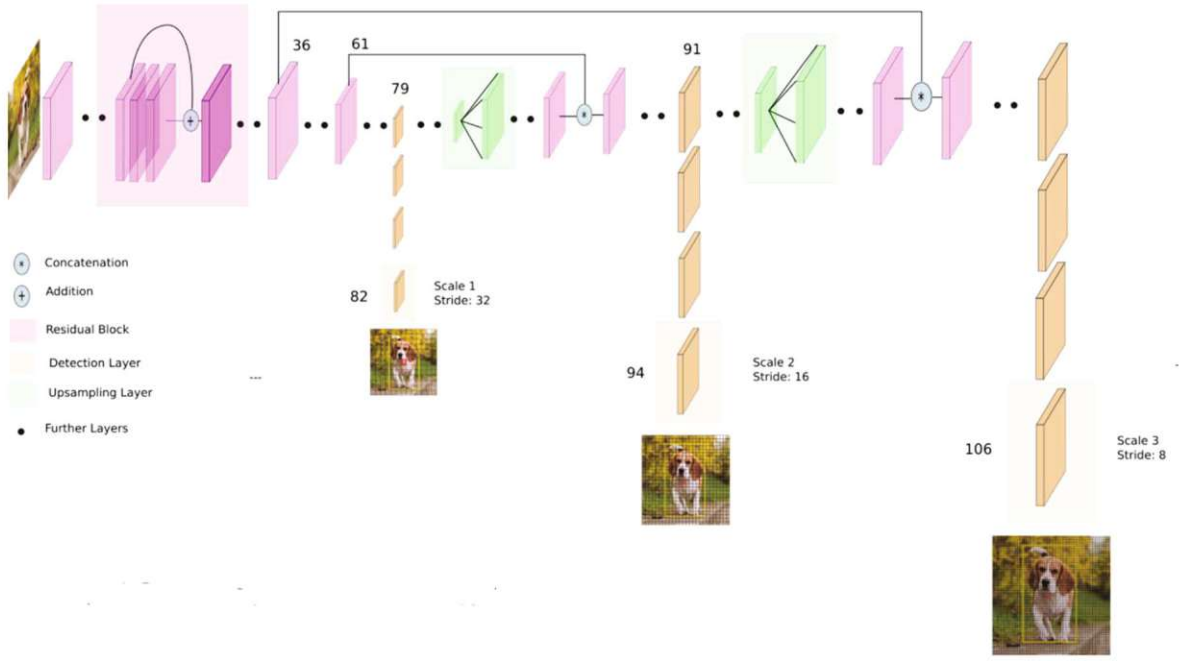


Abbildung 31: Netzwerkaufbau von YOLOv3 [39]

YOLOv3 verwendet für die Erstellung der BB drei AB pro *Grid Cell*. Die Formen der AB werden wie in YOLOv2 durch *K-Means Clustering* aus dem Trainingsdatensatz erstellt. Wie in Abbildung 32 zu sehen ist, wird das Bild in SxS *Grid Cells* aufgeteilt. Für eine Grid Cell (grün dargestellt) werden drei AB ausgewertet. Diese werden, wie in YOLOv2, an das Objekt angepasst. Der Output von YOLOv3 enthält die Koordinaten und Abmaße der BB, die Wahrscheinlichkeit, ob sich ein Objekt in der BB befindet und die Wahrscheinlichkeit um welches Objekt es sich dabei handelt. Diese Angaben werden für jede der drei BB angegeben (gelb, rot und blau in Abbildung 32 dargestellt). Um die BB mit der höchsten Genauigkeit zu erhalten, wird NMS angewendet. [42]

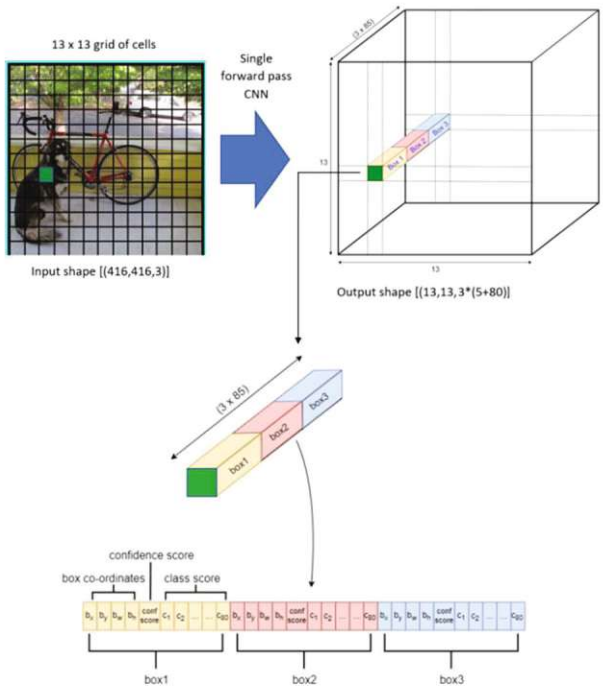


Abbildung 32: *Output* von YOLOv3 [42]

Die letzte große Veränderung in YOLOv3 betrifft die *Loss-Function*. Diese betrachtet weiterhin wie in YOLOv1 die Werte von x, y, h, w, C und P(obj). Der einzige Unterschied ist, dass die Wahrscheinlichkeit C, dass ein Objekt enthalten ist und die Objektklasse P(obj) nicht mehr durch die Summe der quadratischen Fehler berechnet wird, sondern mit logistischer Regression (*binary cross-entropy loss*). [39]

2.8.2.4 YOLOv4

YOLOv4 ist zum Zeitpunkt des Erstellens der vorliegenden Arbeit die aktuellste und beste Version von YOLO. Da sich Redmon aus persönlichen Gründen von der Weiterentwicklung von YOLO zurückzog, übernahmen Alexey Bochkovskiy et al. [30] diese Aufgabe [43]. Die letzte Version des OD-Algorithmus erzielt nochmals bedeutende Fortschritte in Bezug auf Genauigkeit (mAP, siehe Kapitel 4.5) und Geschwindigkeit (FPS, siehe Kapitel 5.5.2).

YOLO ist ein *One-Stage Detector*. Das bedeutet, dass anders als bei anderen OD-Netzwerken (z.B. R-CNN), das Erkennen von Objekten und das Erstellen der BB in einem Schritt passiert. In Abbildung 33 sind die Abläufe des *One-Stage* (grün strichliert) und des *Two-Stage Detectors* (violett strichliert) gekennzeichnet. Das OD-Netzwerk von YOLOv4 besteht nach dem *Input*, aus *Backbone*, *Neck* (nur bei

YOLOv4) und *Dense Prediction*. In Falle eines *Two-Stage Detectors* kommt noch der Schritt der *Sparse Prediction* (wie in R-CNN) hinzu. [44]

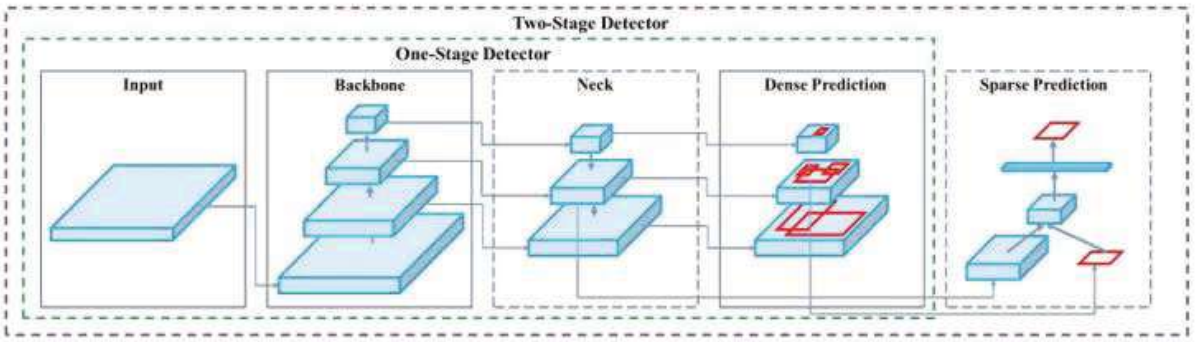


Abbildung 33: Netzwerkaufbau von YOLOv4 [30]

Laut Alexey Ab et al. werden *CSPResNext50*, *CSPDarknet53* und *EfficientNet-B3* als *Backbone* (*feature detector*) für YOLOv4 in Betracht gezogen. Dabei basieren *CSPResNext50* und *CSPDarknet53* auf *DenseNet* [45]. *DenseNet* wurde entwickelt um CNNs so zu verbinden, dass, während des Durchlaufens eines Netzwerkes, keine Verlustsignale auftreten und, durch die Wiederverwendung dieser Merkmale, die Anzahl der Netzparameter reduziert werden. Mit der Anwendung von *DenseNet* werden die extrahierten *Feature-Map's* von der Basisebene getrennt und deren Kopie durch den *Dense-Layer* direkt an die nächste Position weitergegeben (siehe Abbildung 34).

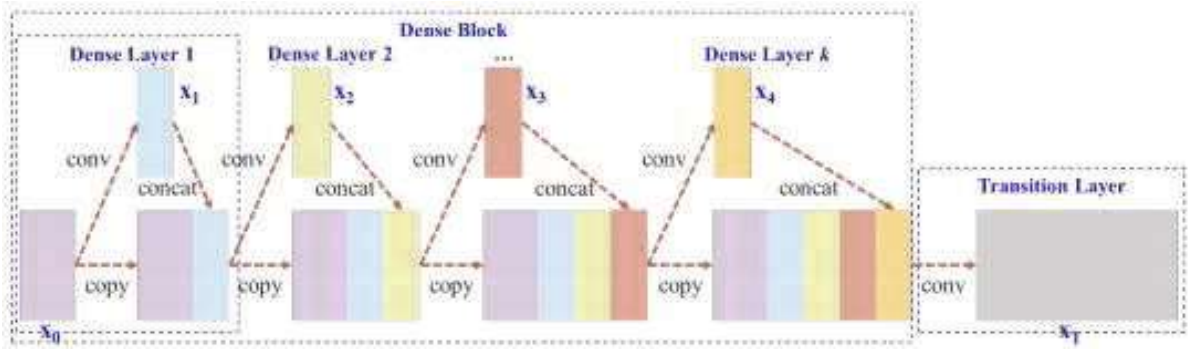


Abbildung 34: *DenseNet* Funktionsweise [22]

Durch die Anwendung von *DenseNet* in Yolov4 sollen durch das Weitergeben unbearbeiteter *Feature-Maps,* Rechenengpässe beseitigt und das Lernen verbessert werden [22]. *EfficientNet* wurde von *Goolge Brain* entwickelt und beschäftigt sich mit dem Skalierungsproblem von CNN. Es gibt mehrere Skalier-Möglichkeiten in einem CNN. Dazu gehört die Input Größe, Breite (*Width*), Tiefe (*Depth*), und Auflösung

(*Resolution*). In Abbildung 35 ist zusätzlich noch eine Kombination aus allen Varianten dargestellt. [46]

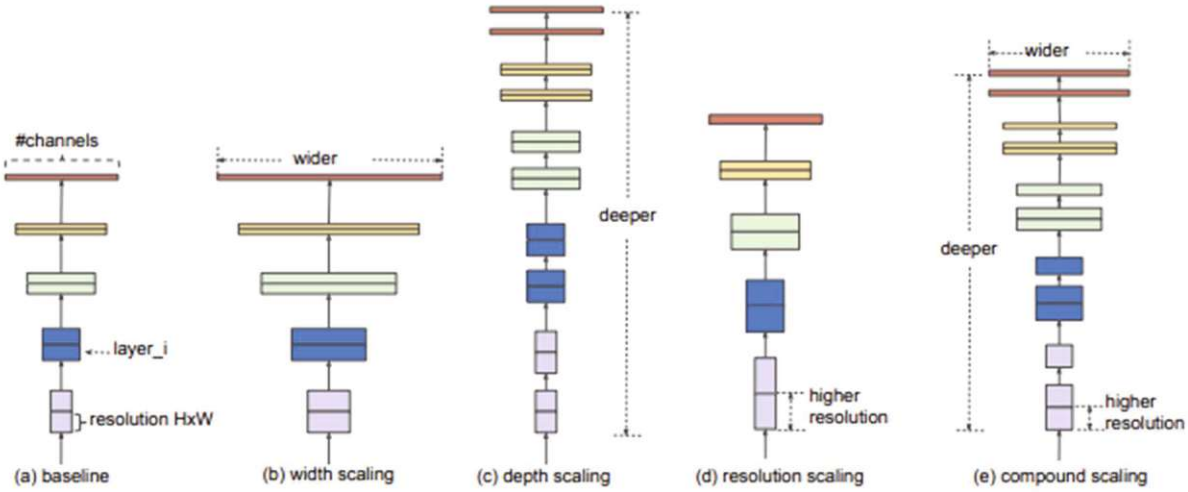


Abbildung 35: *EfficientNet* Skalierungen [46]

Durch viele Experimente mit den unterschiedlichen *Backbones*, sind Alexey AB et al. zu dem Schluss gekommen, dass das *CSPDarknet53* für YOLOv4 die besten Ergebnisse liefert. [22]

Der nächste Abschnitt im Netzwerk wird als *Neck* bezeichnet (siehe Abbildung 35) und wurde erst im YOLOv4 OD-Modell verwendet. Hierbei gilt es, die extrahierten *Features* (Merkmale), welche im *Backbone* gefiltert worden sind, aufzuarbeiten und für den Erkennungsschritt vorzubereiten. Auch hier werden laut Alexey AB et al. mehrere Möglichkeiten der Informationsverarbeitung herangezogen. Dabei handelt es sich unter anderem um FPN (*Feature Pyramid Network*), *PANet* (*Path Aggregation Network*) und *NAS-FPN*. In dieser Arbeit wird auf die verschiedenen *Necks* nicht weiter eingegangen.

Den letzten Abschnitt des OD-Netzwerkes bildet die *Dense Prediction* (auch *Head* genannt). Diese ist zuständig für das Voraussagen der BB bzw. deren Koordinaten und der Objektwahrscheinlichkeit. Für diesen Schritt wird derselbe AB-basierte Prozess wie in YOLOv3 angewendet (siehe YOLOv3).

Somit besteht das OD-Modell YOLOv4 aus drei Teilen. Es wird als:

- Backbone: CSPDarknet,
- Neck: SPP, PAN und für den
- Head: YOLOv3 verwendet.

Um die Leistung von YOLOv4 zu erhöhen, haben die Entwickler sogenannte *Bag of Freebies* angewendet. Dies ist anwendbar, ohne im Netzwerk selbst Änderungen vornehmen zu müssen und kann daher zeitsparend und unkompliziert umgesetzt werden. Bei der Anwendung von *Bag of Freebies* wird vor allem *Data Augmentation* betrieben. Durch diese Methode wird der ursprüngliche Trainingsdatensatz vergrößert, damit das Modell mehr Leistung erbringen kann. In CV ist *Data Augmentation* ein sehr bedeutendes Werkzeug. Die Anwendung in YOLOv4 soll, laut den Autoren, die Effektivität und Auswirkungen dieser Methoden aufzeigen. [22]

Eine weitere Strategie, welche zur Leistungssteigerung eingesetzt wird, bezeichnen die Autoren von YOLOv4 als *Bag of Specials*. Anders als bei den *Bag of Freebies* benötigt die Implementierung der verschiedenen *Bag of Specials* im Vergleich länger, führt aber auch zu einer weiteren Leistungssteigerung. Ein *Bag of Special* ist die Änderung der *Activation Function* (*Mish*). Eines der effektivsten Neuerungen, welche in YOLOv4 angewendet werden kann, ist die *DropBlock Regularization*. Hierbei wird ein Teil eines *Input*-Bildes ausgeblendet bzw. ist dieser für das Netzwerk nicht mehr ersichtlich. Mit dieser Vorgehensweise soll das Netzwerk Merkmale erlernen, die es bei der Betrachtung des gesamten *Inputs* nicht beachtet hat. Als Beispiel kann das Bild eines Hundes herangezogen werden. Es wird der Kopf mit einem schwarzen Block versehen, sodass das Netzwerk nur den Körper des Tieres sieht. Dies kann in weiterer Folge beim Erkennen hilfreich sein, da ein Hund auch erkannt werden soll, wenn ausschließlich der Rumpf zu sehen ist. [22]

YOLOv3 bzw. YOLOv4 entsprechen der State-of-the-Art im Bereich der OD-Modelle. Aufgrund der hohen Performance und der benutzerfreundlichen Implementierung werden diese zwei OD-Modelle für die Umsetzung im Praxisteil herangezogen.

3 State-of-the-Art-Analyse

Object Detection wird heutzutage in vielen Sparten und in den unterschiedlichsten Industrien angewendet. Anwendungsgebiete erstrecken sich unter anderem über die Logistik, verschiedene Anwendungen bei Arbeitsprozessen, Straßenbahnüberwachung, selbstfahrende Autos bis hin zur Gesichtserkennung. Im folgenden Kapitel werden durch eine State-of-the-Art Analyse die wichtigsten Anwendungsbereiche von OD aufgezeigt.

3.1 Anwendung im Straßenverkehr

Eine häufige Anwendung von DL bzw. OD ist die Überwachung des Straßenverkehrs. K. K. Santhosh et al. [47] zeigen in ihrer Arbeit „Anomaly Detection in Road Traffic Using Visual Surveillance: A Survey“ die unterschiedlichen Methoden zur Anomalie-Erkennung im Straßenverkehr auf. Als Anomalien werden Verhaltensweisen bezeichnet, die nicht der Norm entsprechen. So wird mit Hilfe von OD-Algorithmen das der Norm entsprechende Verhalten des Netzwerkes trainiert und erlernt. Alle Verhaltensweisen, die sich vom Trainierten stark unterscheiden, werden als Anomalien und somit als gefährlich eingestuft. Ungewöhnliche Verhaltensweisen wären z.B. das Überfahren einer roten Ampel, das Auffahren eines PKWs auf den Gehsteig oder das Wenden eines PKWs während einer roten Ampel.

Shengyu Lu et al. [2] beschreiben in ihrem Paper eine Methode zur Erkennung von Fahrzeugen und Fußgängern im Straßenverkehr. Dabei wird als OD-Algorithmus YOLO verwendet. Der Trainingsdatensatz besteht aus rund 6000 Bildern im Format 1280x720 wobei 12.000 Labels von fünf *Classes* (Objekte) vergeben wurden. Hierbei handelt es sich um PKWs, LKWs, Busse, Motorräder und Fußgänger. Zusätzlich wurde noch ein Testdatensatz von rund 2.000 Bildern verwendet, um den mAP-Wert zu ermitteln. Im Anwendungsfall wurde ein Video als *Input* verwendet, wobei jeder Bild-*Frame* durch den YOLO-Algorithmus verarbeitet wurde. Ein Beispiel eines *Outputs* ist in Abbildung 36 dargestellt. Als Mindesterkennungswert wurde dabei eine Wahrscheinlichkeit von 0,5 festgelegt.

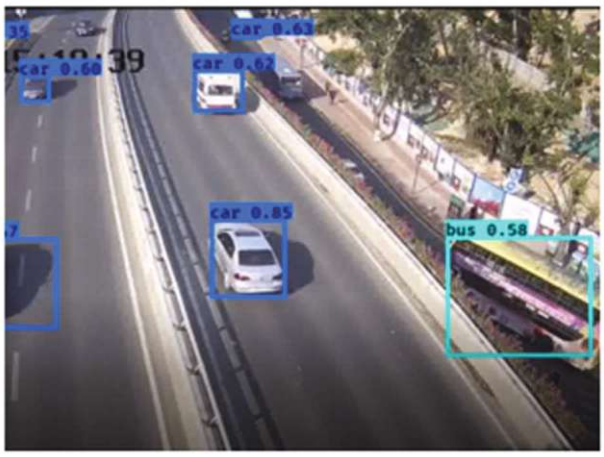


Abbildung 36: Fahrzeugerkennung [2]

Der verwendete YOLO-Algorithmus wurde anschließend mit anderen Algorithmen verglichen, um dessen Leistung festzustellen. Dabei stellte sich heraus, dass YOLO im Vergleich zu CNN, R-CNN und SSD die besten Ergebnisse erzielen konnte.

Ein weiteres Anwendungsfeld im Straßenverkehr ist die Nummernschilderkennung. Peker [48] beschreibt in seinem Paper eine Methode zur Erkennung von Autos und deren Nummernschildern. Es wurden die OD-Algorithmen *Faster* R-CNN, R-FCN (wird in dieser Arbeit nicht behandelt) und SSD miteinander verglichen. Der verwendete Trainingsdatensatz besteht aus 200 Bildern im Format 480x360, die mit verschiedenen Kameras aufgenommen wurden. Anschließend wurden diesen mit Hilfe des Programmes *LabelImg* die Klassen Auto und/oder Nummernschilder zugewiesen. Nach dem Trainieren wurden die Ergebnisse der erwähnten Algorithmen miteinander verglichen, wobei sich herausstellte, dass *Faster* R-CNN die besten Resultate lieferte.

Ein aktuelles Einsatzgebiete von OD im Zusammenhang mit dem Straßenverkehr ist der Einsatz bei selbstfahrenden Autos. Ramos et al. [49] stellen in deren Paper einen DL-Ansatz dar, mit dem kleine und unerwartete Hindernisse auf der Fahrbahn erkannt werden können. Mit Hilfe von CNN werden nicht nur verschiedene Klassen im Trainingsprozess erlernt, sondern das Modell ist auch in der Lage, aufgrund der gelernten Daten Verallgemeinerungen zu treffen. Dies bedeutet, dass auch sogenannte Ausreißer erkannt werden können. Als Ausreißer werden für das Modell nicht sichtbare Objekte bezeichnet. Beim Erkennungsvorgang wird am Eingabebild jeder einzelne Pixel betrachtet. Jedem dieser Pixel wird durch den DL-Ansatz eine bestimmte Klasse zugeordnet. Da bei der Hinderniserkennung der gesamte Fahrraum identifiziert bzw. kontrolliert werden muss, wurden insgesamt drei Klassen definiert: Freiraum, unerwartete Hindernisse am Fahrraum und Hintergrund. Der verwendete OD-Algorithmus wurde mit dem *Lost and found*-Datensatz ausgewertet und erreichte dabei im Vergleich zu anderen State-of-the-Art-Ansätzen eine um rund 27.4 Prozent bessere Leistung. So liegt die Erkennungsrate für Objekte, die bis zu 50 Meter entfernt sind, bei 90 Prozent.

3.2 Anwendungen in der Logistik

Im Bereich der Logistik sind vor allem zwei Aspekte der OD relevant. Dies ist zum einen die Objektlokalisierung und zum anderen die semantische Segmentierung (das Verständnis eines Bildes auf Pixelebene). Hierbei können alle Bereiche der Logistik, bei denen das Erkennen und die Positionierung von Objekten von Bedeutung ist, vom Einsatz dieser profitieren. Mit dem Einsatz von OD-Algorithmen kann ein hoher Grad an Standardisierung erreicht werden, was bedeutet, dass hohe Kosten vermieden werden können. Thiel et al. [50] beschreiben in ihrer Arbeit die verschiedenen Anwendungsgebiete in der Logistik, bei denen DL verwendet wird. In Tabelle 3 werden diese aufgezeigt.

Tabelle 3: Anwendungen von OD in der Logistik

Anwendung	Beschreibung
Fahrerlose Flurförderzeuge	• Erkennung von freien Fahrwegen • Hinderniserkennung und Lokalisierung; Unterschied zwischen statischen und dynamischen Objekten und ob das Ausweichen Sinn macht • Positionserkennung und Orientierung • Erkannte Objekte dienen zur Ortung des Fahrzeuges • Assistenzsystem; frühe Warnung an Fahrer
Kommissionierroboter	• Klassifizierung von Objekten • Lokalisierung von Objekten • Überprüfung manuell kommissionierter Artikel
Kollaborative Roboter	• Absicherung gegen Kollisionen • Gefahrenerkennung an Arbeitsplätzen • Automatische Positionierung bei Arbeitsplatzwechsel
Erstellen semantischer Karten	• 2-D Kartenerstellung eines Fabriklayouts
Augmented Reality (AR)	• Erkennung realer Objekte und deren virtuelle Beschriftung

	• Verwendung beim Einarbeiten neuer Mitarbeiter; Feldarbeiten für Service-Techniker

Ein System zur Kontrolle kommissionierter Artikel beschreiben Hochstein et al. [51] in ihrem Paper „Konsolidierassistent – Assistenzsystem für manuelle Konsolidier- und Sortierprozesse in Distributionszentren“. Das Ziel ist es, durch ein Assistenzsystem das Konsolidieren und Sortieren von Artikel zu erleichtern und somit auch die Zykluszeiten und möglichen Fehler zu minimieren. Hierbei wird der OD-Algorithmus YOLO verwendet. Zum Trainieren des Netzwerks für 20 Artikel (Objekte) werden rund 7.000 Bilder verwendet. Das Sortieren der Artikel erfolgt nach Kundenaufträgen, wobei diese der Reihe nach abgearbeitet werden. Jener Kundenauftrag, für den die meisten Artikel erkannt werden, wird als erster behandelt. Nach der Erkennung der Artikel werden diese dem Kundenauftrag zugeordnet und durch visuelle Erkennung auf den dazugehörigen Kleinladungsträger verwiesen. Durch eine Reihe von Versuchen wurde festgestellt, dass sich durch den Einsatz des Assistenzsystems die Bearbeitungszeit für das Sortieren verringert.

Um die Automatisierung logistischer Prozesse voranzutreiben, haben Schwäke et al. [52] ein System für einen vollautomatischen Kommissionierroboter entwickelt. Mit Hilfe eines Bildverarbeitungssystems wird ein Greifarm gesteuert, der Artikel erkennen und greifen kann.

Teil B

Praxisteil, Zusammenfassung und Ausblick

4 Konzept für die Umsetzung des Praxisteiles

Für die Umsetzung bzw. Realisierung des OD-Modelles wurde ein Konzept erarbeitet, in dem die Vorgänge im folgenden Kapitel Schritt für Schritt erklärt werden. Das Konzept wurde in Anlehnung an das CRISP-DM Prozessmodell (*Cross-Industry Standart Process for Data Mining*) erstellt. Das CRISP-DM Modell wurde bereits 1999 von Experten im Rahmen eines EU-Förderprojektes eingeführt und gilt bis heute als eines der am weitesten verbreiteten Standard-Prozess-Modelle für *Data Mining*. In Abbildung 37 ist der Ablauf des CRISP-DM Modelles zu sehen. [53]

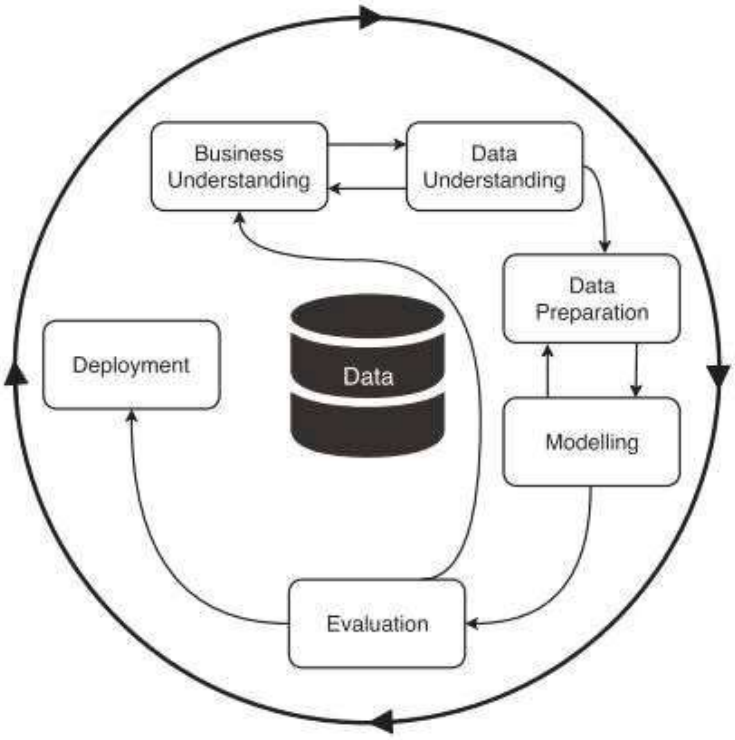


Abbildung 37: Darstellung des CRISP-DM Modelles [53]

Aus dem CRISP-DM Modell wurden folgende Punkte für das Umsetzen des Praxisteiles dieser Arbeit abgeleitet:

1. Definition der Aufgabenstellung (Business Understanding)
2. Sammeln der Daten (Data Understanding)
3. Erstellung eines Datensatzes (Data Preparation)
4. Training des Modelles (Modelling)
5. Messung der Performance des Modelles (Evaluation)
6. Einsatz im Anwendungsgebiet (Deployment)

In den folgenden Unterkapiteln werden die oben genannten Schritte beschrieben.

4.1 Definition der Aufgabenstellung

Um ein OD-Projekt erfolgreich umsetzen zu können ist es wichtig, die Rahmenbedingungen dafür festzulegen. Dies verringert die Fehleranfälligkeit und trägt dadurch zu einem effizienteren und leistungsfähigeren Modell bei. In diesem Schritt wird auch definiert, welches OD-Modell (siehe Kapitel 2.8) für die vorliegende Aufgabenstellung am besten geeignet ist.

4.2 Sammeln der Daten

Nach der Definition der Aufgabenstellung folgt das Sammeln der Daten. Hier gilt der Grundsatz, je besser die Daten sind, mit dem das OD-Modell trainiert wird, desto besser ist die Performance. Die Herkunft der gesammelten Daten spielt im Wesentlichen keine Rolle. Dies können z.B. Bilder aus dem Internet, selbst erstellte Fotos oder ein bereits vorhandener Datensatz einer dritten Person sein. Eine Webseite, die Bilder zur Verfügung stellt, ist z.B. ImageNet [54]. ImageNet wurde von der Universität Stanford und der Universität Princeton erstellt, um für wissenschaftliche Zwecke Datensätze bzw. Bilder für den Bereich DL zur Verfügung zu stellen. Eine weitere Möglichkeit bietet Google mit dessen *Open Image Dataset* [55]. Hier ist es möglich, bereits gelabelte Bilder bestimmter Objekte zu erhalten. Allerdings muss, je nach OD-Modell, das Format der .txt- Datei angepasst werden.

Um eine hohe Erkennungswahrscheinlichkeit vom OD-Modell zu erhalten, muss, je nach Anwendung und Anzahl der zu erkennenden Objekte, eine gewisse Anzahl an Daten verwendet werden. Bei einem Modell ohne Verwendung von *Transfer Learning* sollten ca. 2000 unterschiedliche Bilder pro Objekt verwendet werden. Ein weiterer ausschlaggebender Grund für eine gute Performance des Modelles ist es, eine hohe Anzahl unterschiedlicher Bilder des zu erkennenden Objektes zu verwenden. Konkret bedeutet dies, dass sich u.a. Hintergründe, Größen, Lichtverhältnisse und Perspektiven des Objektes auf den Bildern möglichst stark voneinander unterscheiden. [56]

Bei den gesammelten Daten sollte die gleiche Anzahl von Bildern pro Klasse verwendet werden, um ein gleichmäßiges Training ermöglichen zu können. [56]

4.3 Erstellung eines Datensatzes

Sobald genügend Daten gesammelt sind, wird ein Datensatz für das OD-Modell erstellt. Dabei werden zunächst alle Bilder in einem Ordner platziert. Damit die Daten für das OD-Modell verwendet werden können, müssen alle Bilder gelabelt werden (siehe Kapitel 2.3.1). Dieser Prozess kann mit Hilfe von Tools bewerkstelligt werden. Da die Programmiersprache Python verwendet wird, können die Tools LabelImg [57] und OpenLabeling [58] verwendet werden. In weiterer Folge wird der *Labeling*-Prozess mit dem Tool LabelImg beschrieben, da dieser für die Umsetzung in dieser Arbeit verwendet wurde. In Abbildung 38 ist das GUI (*Graphical User Interface*) von LabelImg zu sehen. Die Vorgehensreihenfolge ist mit den Nummern 1-6 in den rot markierten Feldern zu sehen.

1. Öffnen des Datensatzes
2. Auswählen des richtigen Formates für das OD-Modell
3. Rechteck über das Objekt ziehen
4. Benennung des Objektes
5. Speichern des gelabelten Bildes
6. Wechseln zum nächsten/vorherigen Bild

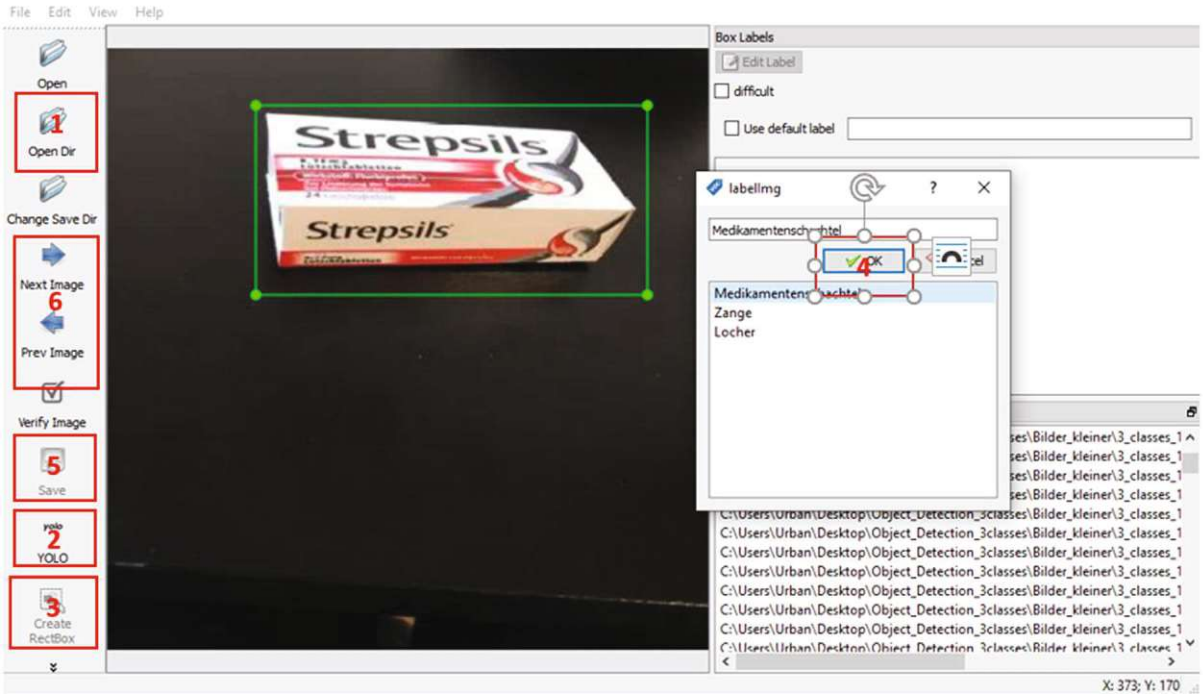


Abbildung 38: GUI von LabelImg

Beim Labeln wird für jedes Bild automatisch eine dazugehörige .txt-Datei erstellt, welche die für das OD-Modell relevanten Informationen enthält. Darin enthalten sind die Angabe der Klasse, die Koordinaten und Abmaße der erstellten GTB bezüglich des gesamten Bildes. In Abbildung 39 ist die .txt-Datei des Bildes, welches in Abbildung

41 verwendet wird, zu sehen. Die erste Zahl (1) gibt Auskunft darüber, um welches Objekt es sich handelt. Die restlichen Zahlen entsprechen den Angaben bzw. Koordinaten der BB.

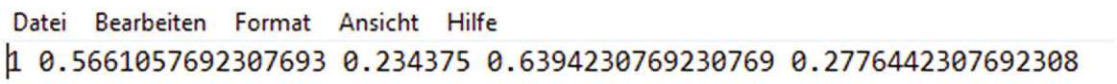


Abbildung 39: .txt- Datei die während des Labelns erzeugt wird

4.4 Training des Modelles

Nach dem Erstellen des Datensatzes wird das OD-Modell trainiert. Das Training muss nicht manuell programmiert werden, sondern kann von diversen *Github Repositories* heruntergeladen werden [59] [56]. Um ein OD-Modell umsetzten zu können, wird ein *Deep Learning Framework* benötigt. Bei *Deep Learning Frameworks* handelt es sich um Libraries oder Tools, die es Personen mit nur oberflächlichen Kenntnissen über KI bzw. ML ermöglichen, ein solches Modell nutzen zu können. Über diese Frameworks kann ein DL-Modell auf einen selbst erstellten Datensatz trainiert werden. Die wichtigsten Frameworks werden kurz genannt [60]:

- TensorFlow
- Keras
- PyTorch
- Theano
- DL4J
- Caffe
- Chainer
- Microsoft CNTK
- Darknet [61]

Die Frameworks können mit dem Einsatz von CPU (Central Processing Unit) oder GPU (Graphical Processing Unit) trainiert werden. Das Trainieren von NN ist ein rechenintensiver Vorgang und nimmt deshalb viele Ressourcen in Anspruch. Mittels GPU und dessen hoher Anzahl an *Processing Units* (Kerne) ist es möglich, DL-Probleme parallel zu lösen. Aus diesem Grund werden häufig GPU-Systeme für DL verwendet, da diese, im Vergleich zu CPU-Systemen, vier- bis fünfmal schneller sind. [62]

Da die Hardware für ein leistungsfähiges GPU-System sehr kostspielig ist, kann über Webseiten auf Server zugegriffen werden, auf welchen diese kostenlos genutzt werden können. Eine solche Website ist z.B. Google Colaboratory [63], kurz *Colab*

genannt. Hier ist es möglich eine *Virtual Machine* (VM) zu erstellen und mittels der Programmiersprache Python das DL-Problem bzw. die Aufgabenstellung zu beschreiben. *Colab* stellt mehrere GPU`s zur Verfügung. Diese werden jedoch per Zufall vergeben, sodass der Benutzer keine Auswahlmöglichkeit hat. Als GPU`s werden die NVIDIA K80s, T4s, P4s und P100s verwendet. Diese befinden sich in der Preisklasse zwischen 2.000€ bis 20.000€. [63]

Als Grundlage für die Implementierung in *Colab* wurden [64] und [65] herangezogen.

4.5 Messung der Performance des Modelles

Das trainierte OD-Modell muss nun auf dessen Performance geprüft werden, um gegebenenfalls die Effizienz steigern zu können. Die Performance wird anhand der *Loss-Function* (siehe Kapitel 2.3.2), dem *Recall*, der *Precision* und dem mAP-Wert gemessen.

Da es sich bei der OD um ein Entscheidungsproblem handelt, werden den erkannten Objekten durch Klassifizieren die Labels positiv oder negativ zugeordnet. Dieses Ergebnis kann in der *Confusion Matrix* (siehe Tabelle 4) dargestellt werden. Diese besteht aus vier Kategorien: den *True Positives* (TP), den *False Positives* (FP), den *True Negatives* (TN) und den *False Negatives* (FN). Als TP werden all jene Objekte bezeichnet, die korrekt erkannt wurden, als FP werden hingegen jene Objekte bezeichnet, die fälschlicherweise als richtig eingestuft bzw. erkannt wurden. Als TN werden jene Objekte bezeichnet, die korrekt als falsch bezeichnet wurden und die FN entsprechen dem richtigen Objekt, welches als falsch bewertet wurde. [66]

Tabelle 4: *Confusion Matrix* [66]

	Actual Positive	*Actual Negative*
Predicted Positive	TP	FP
Predicted Negative	FN	TN

Mittels der Ergebnisse der *Confusion Matrix*, können die Performanceindikatoren *Recall* (Formel 11) und *Precision* (Formel 12) errechnet werden. [66]

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

$$Precision = \frac{TP}{TP + FP} \tag{12}$$

Der errechnete *Precision*-Wert gibt dabei an, zu wie viel Prozent die erstellten Vorhersagen richtig sind. Der *Recall*-Wert hingegen gibt Auskunft darüber, ob das OD-Modell jedes Objekt erkannt hat, welches es erkennen soll. Um die Erkennung in den oben genannten Kategorien klassifizieren zu können, muss ein Schwellenwert (*Threshhold*) für die Erkennungswahrscheinlichkeit angegeben werden. Dieser Schwellenwert gibt an, ab welcher Wahrscheinlichkeit ein Objekt als TP oder FP bewertet wird. Dieser Wert kann grundsätzlich beliebig gewählt werden, wird aber meistens mit 0,5 bis 0,95 festgelegt. Eine kleinere Zahl wäre nicht sinnvoll, da diese die Aussagekraft des OD-Modelles negativ beeinflussen würde. Des Weiteren kann die *Average Precision* (AP) des OD-Modelles ermittelt werden. Sobald ein OD-Modell in der Lage ist, mehr als ein Objekt zu erkennen bzw. zu kategorisieren, spricht man von der *Mean Average Precision* (mAP). Hierbei wird das OD-Modell im Training durch einen Validierungsdatensatz getestet, indem die vom OD-Modell erstellten BB mit der GTB des Validierungs- bzw. Testdatensatzes abgeglichen werden [67]. Die genaue Berechnung des Wertes kann in [68] nachgelesen werden.

4.6 Einsatz im Anwendungsgebiet

Das trainierte OD-Modell kann nach Evaluierung der Performance im gewünschten Bereich eingesetzt werden. Für den Einsatz im Anwendungsgebiet dieser Arbeit siehe Kapitel 5.6.

5 Umsetzung / Implementierung

Die Umsetzung erfolgt nach dem Konzept, welches in Kapitel 4 beschrieben wurde. Als OD-Modell werden YOLOv3 und YOLOv4 verwendet, da diese den State-of-the-Art-OD-Modellen entsprechen. Die Ergebnisse und Performances dieser werden gegenübergestellt bzw. verglichen und in Kapitel 6 dargelegt.

Für die Umsetzung wurde folgende Hardware verwendet:

Tabelle 5: Verwendete Hardware für die Umsetzung der Aufgabenstellung

Hardware	Technische Daten
Rechner ASUS F550L	• Intel Core i5-4210U • NVIDIA GEFORCE 840M • 8GB RAM
Kamera iPhone 7	• 12 Megapixel Kamera
Webcam Logitech BRIO	• 4K Auflösung

5.1 Definition der Aufgabenstellung

Ziel dieser Arbeit ist es, ein OD-Modell so zu trainieren, dass die vordefinierten Objekte mit einer hohen Wahrscheinlichkeit erkannt werden können und es so in weiterer Folge beim Kommissionierprozess als zusätzliches Kontrollinstrument eingesetzt werden kann. Als OD-Modelle werden hierfür YOLOv3 und YOLOv4 verwendet, um etwaige Performanceunterschiede aufzeigen zu können.

Folgende Rahmenbedingungen wurden für die Umsetzung festgelegt:

- Drei zu erkennende Objekte wurden definiert: ein Locher, eine Medikamentenschachtel und eine Zange (siehe Abbildung 44). Bei der Auswahl wurde darauf geachtet, Objekte zu verwenden, die sich deutlich voneinander unterscheiden, um das OD-Modell mit einem kleinen Datensatz trainieren zu können.

- Die Objekte werden in einem typischen Lagerbehälter platziert, um eine möglichst realitätsnahe Umgebung darstellen zu können. Der verwendete Lagerbehälter ist in Abbildung 40 zu sehen und hat die Maße 46,5x31,5x20 cm.

Abbildung 40: Lagerbehälter mit den Abmaßen 46,5x31,5x20 cm

- Die Objekterkennung erfolgt mittels aufgenommener Bilder und Live-Video. Der Abstand zwischen Lagerbehälter und Aufnahmegerät beträgt ca. 50-80 cm.
- Die Ausgabe des OD-Modelles wird in einer .txt-Datei gespeichert und mit einer manuell erstellten Liste abgeglichen. Dies soll den Kommissionierprozess darstellen, wobei die manuell erstellte Liste der Bestellung entspricht. Diese wird anschließend mit der Liste aus dem OD-Modell verglichen und überprüft.

5.2 Sammeln der Daten

Wie bereits erwähnt, soll das OD-Modell die drei Artikel Locher, Medikamentenschachtel und Zange erkennen. In Abbildung 41 sind diese dargestellt.

a) b) c)

Abbildung 41: a) Locher, b) Medikamentenschachtel, c) Zange

Die Bilder wurden mit einem iPhone 7 aufgenommen. Dabei wurde darauf geachtet, die Objekte aus möglichst vielen unterschiedlichen Blickwinkeln und Entfernungen, sowie mit verschiedenen Hintergründen aufzunehmen. Dies ist von großer Bedeutung, da das OD-Modell dadurch sehr gute Lernergebnisse liefern kann. Wird der Hintergrund z.B. auf eine Farbe reduziert und das Objekt immer aus derselben Entfernung aufgenommen, ist das OD-Modell bei der Erkennung nicht in der Lage passende bzw. richtige Ergebnisse zu liefern. Diese Erkenntnisse wurden im Laufe der Erstellung des Modelles durch verschiedene Versuche erlangt.

Pro Objekt wurden rund 50 Bilder aufgenommen und in einem Ordner abgelegt.

5.3 Erstellung des Datensatzes

Im nächsten Schritt wurde der Datensatz für das OD-Modell erstellt. Dabei wurden alle Bilder mit dem Programm LabelImg gelabelt. Die detaillierte Vorgehensweise dazu ist in Kapitel 4.3 beschrieben. Der *Labeling*-Prozess nimmt sehr viel Zeit in Anspruch. Für das Labeln von 150 Bildern wurden ca. 30 Minuten benötigt. Das entspricht einer Dauer von zwölf Sekunden pro Bild.

Beim *Labeling*-Prozess wird zu jedem Bild eine .txt-Datei erstellt, in der alle relevanten Daten für das OD-Modell vorhanden sind. Abbildung 42 dokumentiert den Datensatz.

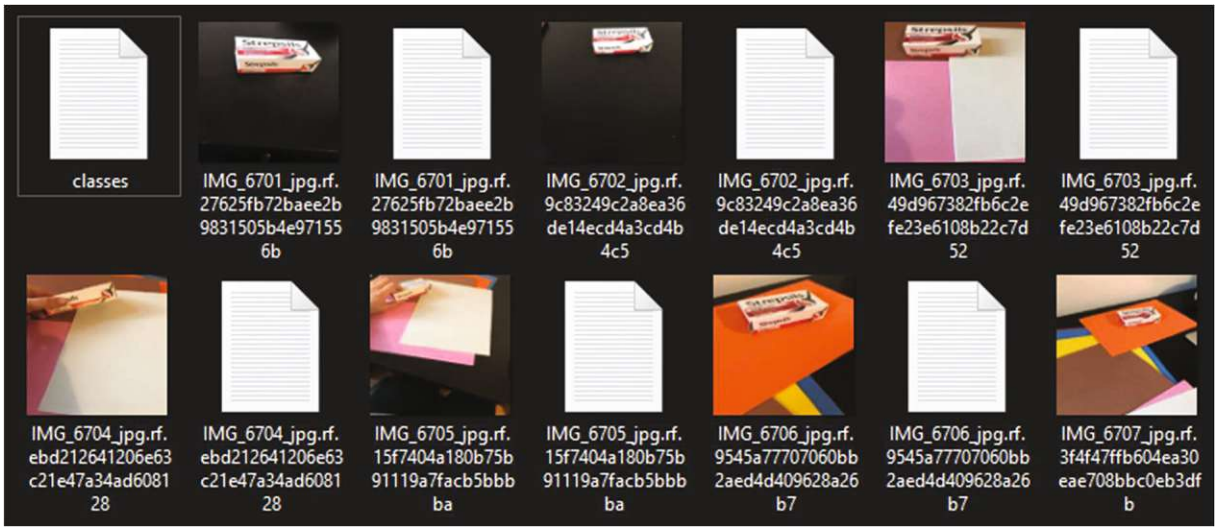


Abbildung 42: Darstellung des gelabelten Datensatzes

Die classes.txt-Datei benötigt das OD-Modell im Laufe des Trainings. Hier ist besonders darauf zu achten, dass die Reihenfolge der Objekte in der classes.txt-Datei mit der ersten Zahl in der .txt-Datei, des gelabelten Bildes, übereinstimmt, da diese

Zahl als Referenz für die unterschiedlichen Objekte dient. In der classes.txt-Datei werden die zu erkennenden Objekte untereinander aufgelistet. Jede Zeile entspricht einer Nummer, beginnend bei null. In Abbildung 43 a) ist die classes.txt-Datei ersichtlich. Hierbei wird dem Objekt Locher die Zahl Null, der Medikamentenschachtel die Eins und der Zange die Zwei zugwiesen.

a)

b)

c)

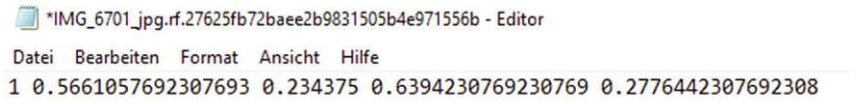


Abbildung 43: a) Darstellung der classes.txt-Datei, b) Bild aus dem Datensatz vom Objekt Medikamentenschachtel, c) .txt-Datei nach dem Label-Prozess

Abbildung 43 b) stellt ein Bild des Datensatzes vom Objekt Medikamentenschachtel dar und c) entspricht der dazugehörigen .txt-Datei nach dem *Labeling*-Prozess. Hier ist zu sehen, dass die erste Zahl in der .txt-Datei eins ist, welche laut classes.txt dem Objekt Medikamentenschachtel entspricht. Die Zahlen, die in der .txt-Datei enthalten sind, dürfen nach dem *Labeling*-Prozess nicht mehr verändert werden. Wird z.B. die erste Ziffer verändert ist das System nicht mehr im Stande, die Objekte richtig zu erkennen. Bei einer Veränderung der Angaben zur BB, lernt das OD-Modell im Trainingsprozess die falschen Merkmale und ist anschließend nicht in der Lage, die vordefinierten Objekte richtig zu erkennen.

Um den Trainingsprozess zu beschleunigen, werden alle Bilder und die dazugehörigen .txt-Dateien in das Format 416x416 umgewandelt. Dadurch wird dem OD-Modell dieser Schritt erspart. Da die Bilder auf diese Größe verändert werden, aktualisieren sich auch die jeweiligen .txt-Dateien. Dies wird mit Hilfe der Roboflow Webseite umgesetzt [69]. Der Datensatz besteht schlussendlich aus 301 Dateien. Von den 301 Dateien entsprechen 150 den Bildern der Objekte, 150 den dazugehörigen .txt-Dateien und eine der classes.txt-Datei.

Der Datensatz für das OD-Modell YOLOv4 wird nochmals in zwei Kategorien unterteilt. Dabei enthält der Trainingsdatensatz 241 Dateien. 120 dieser Dateien entsprechen den Bildern der Objekte, weitere 120 den dazugehörigen .txt-Dateien und eine der classes.txt- Datei. Die zweite Kategorie ist der Test- bzw. Validierungs-Datensatz. Dieser enthält 30 Bilder der Objekte und die dazugehörigen .txt-Dateien. Mit Hilfe dieser Aufteilung wird das OD-Modell während des Trainings getestet und kann somit Feedback über dessen Performance erhalten (siehe Abbildung 59).

Die Datensätze werden anschließend in eine zip-Datei komprimiert. Weiters ist noch die Benennung der Datensätze von großer Bedeutung, d.h. es sollen im Laufe der Implementierung immer dieselben Bezeichnungen verwendet werden, damit sich das OD-Modell beim Trainieren auf die richtigen Referenzen beziehen kann.

5.4 Training des Modelles

Das OD-Modell wird mittels einer VM auf *Colab* trainiert. [63] Hierbei wird die virtuelle Umgebung mittels der Programmiersprache Python beschrieben und implementiert. Auf den folgenden Abbildungen werden die Befehle für die Ausführungen des Python-Codes gezeigt. Diese werden durch das Klicken auf den Pfeil (immer links auf den Abbildungen zu sehen) ausgeführt. Im folgenden Kapitel wird das Erstellen der VM für YOLOv3 aufgezeigt. Für das Training mit YOLOv4 sind nur wenige Änderungen notwendig. Diese werden am Ende des Kapitels angeführt.

5.4.1 Training von YOLOv3

Zu Beginn werden alle Datensätze in einem Google *Drive Account* abgelegt. Über Google *Colab* wird auf Google *Drive* zugegriffen, um die benötigten Daten für die VM bzw. zum Trainieren des OD-Modelles zu erhalten. Diese Vorgehensweise hat den Vorteil, dass sie weniger zeitintensiv ist als das Hinzufügen aller Daten in Google *Colab* vom lokalen Rechner. Um dies durchführen zu können, werden die Befehle wie in Abbildung 44 verwendet.

```
# Zugriff auf Google Drive Ordner
from google.colab import drive
drive.mount('/content/gdrive')
!ln -s /content/gdrive/My\ Drive/ /mydrive
!ls /mydrive
```

Abbildung 44: Verbindung zwischen Google *Colab* und Google *Drive*

Im nächsten Schritt wird die Umgebung der VM erstellt. Dabei wird auf eine bereits erstellte Umgebung von AlexeyAB [56] zugegriffen und diese in die Google *Colab* Seite „geklont". In diesem Schritt werden alle benötigten Ordner und Dateien in die VM eingespielt. Das verwendete *Framework* ist *Darknet*. In Abbildung 45 ist der notwendige Befehl dargestellt.

```
# Klonen des Darknet Ordners in die VM
!git clone https://github.com/AlexeyAB/darknet
```

Abbildung 45: Klonen des Darknet Ordners in die VM

Um alle Abhängigkeiten, Umgebungsvariablen, Ordner oder Skripte, die ausgeführt werden sollen, richtig benützen zu können, wird ein *Make-File* benötigt. In dieser Datei werden alle Abhängigkeiten richtig verknüpft, um sicher zu gehen, dass die Kompilierwerkzeuge korrekt sind. Bevor das *Make-File* erstellt wird, werden die Konfigurationen festgelegt (Abbildung 46 a). Der Befehl für das Erstellen des *Make-File* ist in Abbildung 46 b) dargestellt.

a)

```
# Konfigurieren des Make-File
%cd darknet
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
```

b)

```
# Erstellen des Make-File
!make
```

Abbildung 46: a) Konfigurieren des *Make-File*; b) Erstellen des *Make-File*

Für das Training von YOLO benötigt man das YOLO.cfg-*File*. Diese ist bereits in der zuvor erstellten VM-Umgebung vorhanden und muss lediglich noch kopiert werden

(Abbildung 47 a) sowie einige Angaben darin geändert werden. Die Kopie des .cfg-*Files* wird als yolov3_training betitelt. In dem .cfg-*File* (Konfigurationsdatei) werden alle wichtigen Bedingungen bzw. Konfigurationen für das individuelle Training festgelegt. Wie in Abbildung 47 b) zu sehen ist, wird der Wert des *Batch* von 1 auf 64 verändert und die *Subdivisions* von 1 auf 16 verändert. Der *Batch* bzw. die *Batch Size* gibt an, wie viele Bilder pro Iteration verwendet werden. Die *Subdivisions* teilen die *Batch Size* nochmals weiter auf. In diesem Fall wird die *Batch Size* von 64 nochmals durch 16 geteilt. Dies bedeutet, dass die Bilder durch vier Vorgänge dem GPU zum Prozess weitergeleitet werden. Anschließend startet die zweite *Iteration*. Als nächstes wird die Anzahl der *max_batches* von 500200 auf 6000 geändert, da diese laut AlexeyAB [56] classes*2000, also 3*2000=6000, sein sollen. In der Zeile *Steps* werden diese auf 80% bzw. 90% der *max_batches* verändert (4800 und 5400). Da YOLOv3 Vorhersagen an drei verschiedenen Punkten im OD-Modell vornimmt, muss die Anzahl der Objekte (*classes*) und die der Filter (*filters*) an drei Stellen angepasst werden. Die Anzahl der *classes* wird auf drei gesetzt. Die der Filter wird mittels der Formel (classes+5)*3 berechnet. Daraus ergibt eine Anzahl von 24 Filtern.

a)

```
# Erstellen einer Kopie des yolov3.cfg-File
!cp cfg/yolov3.cfg cfg/yolov3_training.cfg
```

b)

```
#Verändern der Angaben in dem yolov3.cfg-File
!sed -i 's/batch=1/batch=64/' cfg/yolov3_training.cfg
!sed -i 's/subdivisions=1/subdivisions=16/' cfg/yolov3_training.cfg
!sed -i 's/max_batches = 500200/max_batches = 6000/' cfg/yolov3_training.cfg
!sed -i '610 s@classes=80@classes=3@' cfg/yolov3_training.cfg
!sed -i '696 s@classes=80@classes=3@' cfg/yolov3_training.cfg
!sed -i '783 s@classes=80@classes=3@' cfg/yolov3_training.cfg
!sed -i '603 s@filters=255@filters=24@' cfg/yolov3_training.cfg
!sed -i '689 s@filters=255@filters=24@' cfg/yolov3_training.cfg
!sed -i '776 s@filters=255@filters=24@' cfg/yolov3_training.cfg
```

Abbildung 47: a) Erstellen einer Kopie des yolov3.cfg-*File*; b) Verändern der Angaben in dem yolov3.cfg-*File*

Des Weiteren werden noch eine obj.names- und eine obj.data-Datei benötigt. Diese werden im Google *Drive* Ordner abgelegt, welcher bereits mit der VM verbunden ist. Die obj.names-Datei entspricht genau der classes.txt-Datei, welche beim *Labeling*-Prozess entstanden ist. Das bedeutet, es gibt genau drei Einträge, die den Objekten Locher, Medikamentenschachtel und Zange entsprechen. Die obj.data-Datei wird benötigt, um der VM die richtigen Pfade für das Training zuzuweisen. Diese enthält die

Anzahl der Objekte (*classes*), den Pfad der train.txt-, der test.txt-Datei und zusätzlich den Pfad des backup-Ordners. Die train.txt-Datei wird mit einem Python-Skript erstellt, welches ebenso in den Google Drive Ordner geladen wird. Diese enthält den vollständigen Pfad der benutzten Trainingsbilder. Der *Backup*-Ordner wird benutzt, um nach einer gewissen Anzahl von Iterationen eine Zwischenspeicherung des trainierten Netzwerkes erstellen zu können. Dadurch wird sichergestellt, dass bei einem möglichen Systemausfall ab diesem Zeitpunkt weitertrainiert werden kann. Weiters können die Resultate anhand verschiedener Versionen verglichen werden. In Abbildung 48 sind die Schritte für die Implementierung in Google *Colab* aufgezeigt. Mit den Befehlen in Abbildung 48 a) werden die obj.names- und obj.data-Dateien in die VM geladen. Um die train.txt-Datei erstellen zu können, muss der gezippte Trainingsdatensatz obj.zip in die VM geladen und dort entpackt werden (Abbildung 48 b). Anschließend wird das Python-Skript (generate_train.py) in die VM geladen und ausgeführt, um die train.txt-Datei erstellen zu können (Abbildung 51 c). Dies kann auch ohne Skript erstellt werden, würde dann aber mehr Zeit in Anspruch nehmen. Die Vorlage des Python-Skripts wurde aus einer *Github Repository* entnommen. [64]

a)

```
# upload der obj.names und obj.data Dateien
!cp /mydrive/Yolov3_3classes/obj.names ./data
!cp /mydrive/Yolov3_3classes/obj.data  ./data
```

b)

```
# upload des obj.zip Ordners und entpacken der Bilder in der VM
!cp /mydrive/Yolov3_3classes/obj.zip ../
```

```
!unzip ../obj.zip -d data/
```

c)

```
# upload der generate_train.py Datei in die VM und Ausführung dieser
!cp /mydrive/Yolov3_3classes/generate_train.py ./
```

```
!python generate_train.py
```

Abbildung 48: a) Upload der obj.names und obj.data- Dateien; b) Upload des obj.zip Ordners und entpacken der Bilder in der VM; c) upload der generate_train.py Datei in die VM und Ausführung dieser

Um beim Trainieren des Netzwerkes bessere Ergebnisse erzielen und Zeit einsparen zu können, wird die Methode *Transfer Learning* (siehe Kapitel 2.5) verwendet. Hierbei wird ein bereits vortrainiertes Netzwerk verwendet und nur spezifisch für die drei zu erkennenden Objekte trainiert. Das *Backbone*, welches hierfür verwendet wird, heißt darknet53.conv.74. Dieses wurde mittels des *ImageNet* Datensatzes vortrainiert [70].

Der Befehl, mit dem das vortrainierte Netzwerk in die VM geladen werden kann, wird in Abbildung 49 gezeigt.

```
# download bzw. upload des vortrainierten Netzwerkes
!wget https://pjreddie.com/media/files/darknet53.conv.74
```

Abbildung 49: Download bzw. Upload des vortrainierten Netzwerkes von YOLOv3

Nach all den vorgenommenen Implementierungen bzw. Verknüpfungen, kann das OD-Modell in der VM trainiert werden. Dabei wird der Befehl, wie in Abbildung 50 a) dargestellt, benutzt. Während des Trainings werden ab einer gewissen Anzahl an Iterationen automatisch Zwischenspeicherungen durchgeführt und im *Backup*-Ordner im Google *Drive Account* abgelegt. Sollte es zu etwaigen Unterbrechungen kommen, muss das Netzwerk nicht mehr von Beginn an trainiert werden, sondern es kann ab dem zuletzt gespeicherten Zeitpunkt weitertrainiert werden (Abbildung 50 b).

a)

```
# Start des Trainings in der VM
!./darknet detector train data/obj.data cfg/yolov3_training.cfg darknet53.conv.74 -dont_show
```

b)

```
# Start des Trainings ab dem letzten Stand
!./darknet detector train data/obj.data cfg/yolov3_training.cfg /mydrive/Yolov3_3classes/backup/yolov3_training_last.weights -dont_show
```

Abbildung 50: a) YOLOv3 Start des Trainings in der VM; b) YOLOv3 Start des Trainings ab den letzten Stand

Während des Trainings werden die Werte der *Loss-Function* aufgezeichnet. Aufgrund dieser Werte kann das Training beliebig abgebrochen werden, sobald es den Erwartungen entspricht. Wenn das Training nicht manuell abgebrochen wird, läuft es so lange weiter, bis die Anzahl der *max_batches* erreicht ist. Es ist nicht zwingend notwendig das Netzwerk bis zu diesem Zeitpunkt trainieren zu lassen, da sich nach einem gewissen Zeitpunkt der Wert der *Loss-Function* einpendelt bzw. sogar wieder anfängt zu steigen. Der Grund dafür ist, dass die verwendete *Loss-Function* keine oder nur mehr wenige lokale Minima erkennen kann und zwangsläufig Werte verwendet werden, die sich rund um ein Minimum befinden.

In Abbildung 51 ist die aufgezeichnete *Loss-Function* zu sehen. Dabei ist gut zu erkennen, wie sich die *Learning Rate* bzw. der Wert der *Loss-Function* mit der Anzahl der Iterationen minimiert. Der erreichte *Loss*-Wert beträgt nach 2300 Iterationen

0.0599. Nachdem dieser Wert erreicht ist, wird das Training abgebrochen, da sich dieser in den letzten paar hundert Iterationen nicht mehr wesentlich verändert hätte.

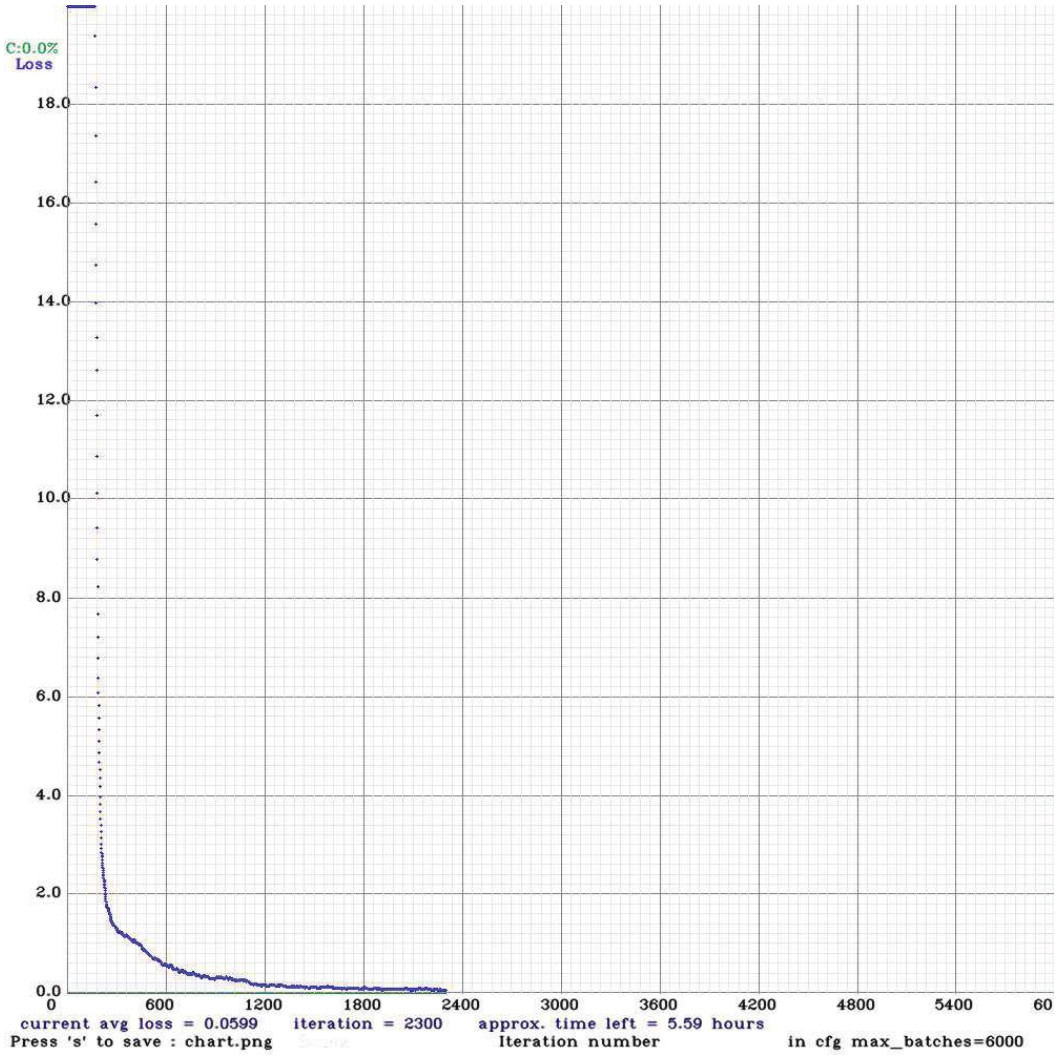


Abbildung 51: Ergebnisse der *Loss-Function* im Laufe des Trainings

5.4.2 Training von YOLOv4

Beim Training für das OD-Modell mit YOLOv4 wird die VM auf Google *Colab* bis auf wenige Unterschiede gleich erstellt wie mit YOLOv3. Ein Unterschied betrifft den Datensatz. Dieser wird, wie in Kapitel 5.3 erwähnt, in Trainings- und Testdatensatz aufgeteilt. Beide müssen auf Google *Drive* abgelegt werden, damit die VM auf diese zugreifen kann. In Abbildung 52 werden die benötigten Befehle dargestellt, um die Bilder in die VM zu laden und anschließend entpacken zu können. Der Trainingsdatensatz wird obj.zip und der Testdatensatz test.zip genannt.

```
# upload des obj.zip und test.zip Ordners und entpacken der Bilder in der VM
!cp /mydrive/Yolov4_3classes/obj.zip ../
!cp /mydrive/Yolov4_3classes/test.zip ../
```

```
!unzip ../obj.zip -d data/
!unzip ../test.zip -d data/
```

Abbildung 52: Upload des obj.zip und test.zip Ordners und entpacken der Bilder in der VM

Wie in YOLOv3 wird die Anzahl der *Classes* und *Filter* angepasst. Diese müssen in der .cfg-Datei entsprechend adaptiert werden. Die Zahl der Filter wird nur direkt vor dem Abschnitt YOLO (in der .cfg-Datei) dreimal verändert. Alle anderen Filter bleiben unverändert.

Da zwei Datensätze verwendet werden, müssen jeweils eine train.txt- und eine test.txt-Datei erstellt werden, in denen der Pfad der verwendeten Bilder aufgelistet wird. Diese werden, wie für YOLOv3, mit einem Python-Skript [65] erstellt (Abbildung 53). Sie können aber auch manuell erstellt werden, was aber zu einem beträchtlich höheren Zeitaufwand führen würde.

```
# upload der generate_train.py und generate_test.py Datei in die VM und Ausführung dieser
!cp /mydrive/Yolov4_3classes/generate_train.py ./
!cp /mydrive/Yolov4_3classes/generate_test.py ./
```

```
!python generate_train.py
!python generate_test.py
```

Abbildung 53: Upload der generate_train.py und generate_test.py-Datei in die VM und Ausführung dieser

Wie in Kapitel 2.8.5 beschrieben, verändert sich bei YOLOv4 das verwendete Netzwerk bzw. das *Backbone*. Als *Backbone* wird yolov4.conv.137 verwendet, welches mit dem MS COCO Datensatz vortrainiert wurde [56]. Dieses Netzwerk wird mit dem in Abbildung 54 ersichtlichen Befehl in die VM geladen.

```
# download bzw. upload des vortrainierten Netzwerkes
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.conv.137
```

Abbildung 54: Download bzw. Upload des vortrainierten Netzwerkes von YOLOv4

Wenn all diese Schritte erfolgreich durchgeführt wurden, kann mit dem Befehl in Abbildung 55 a) das Training gestartet werden. Dieses kann wie bei YOLOv3 nach einer unerwarteten Unterbrechung ab dem letzten Trainingsstand wiederaufgenommen werden. (Abbildung 55 b).

a)

```
# Start des Tranings in der VM
!./darknet detector train data/obj.data cfg/yolov4-obj.cfg yolov4.conv.137 -dont_show -map
```

b)

```
# Start des Trainings ab dem letzten Stand
!./darknet detector train data/obj.data cfg/yolov4-obj.cfg /mydrive/Yolov4_3classes/backup/yolov4-obj_last.weights -dont_show
```

Abbildung 55: a) YOLOv4 Start des Trainings in der VM; b) YOLOv4 Start des Trainings ab den letzten Stand

In Abbildung 56 werden die Werte der *Loss-Function* und die ermittelten mAP-Werte, die während des Trainings berechnet wurden, dargestellt. Das Training wurde nach 2400 Iterationen und einem Wert der *Loss-Function* von 0.29 abgebrochen, da sich dieser ab diesem Zeitpunkt nicht mehr relevant verändern würde. Da ein Testdatensatz verwendet wird, kann während des Trainings auch der mAP-Wert aufgezeigt werden. Dieser beträgt nach der genannten Anzahl an Iteration rund 100% und bedeutet, dass die erstellten BB des OD-Modelles mit der GTB im Testdatensatz bis zu 100% übereinstimmen.

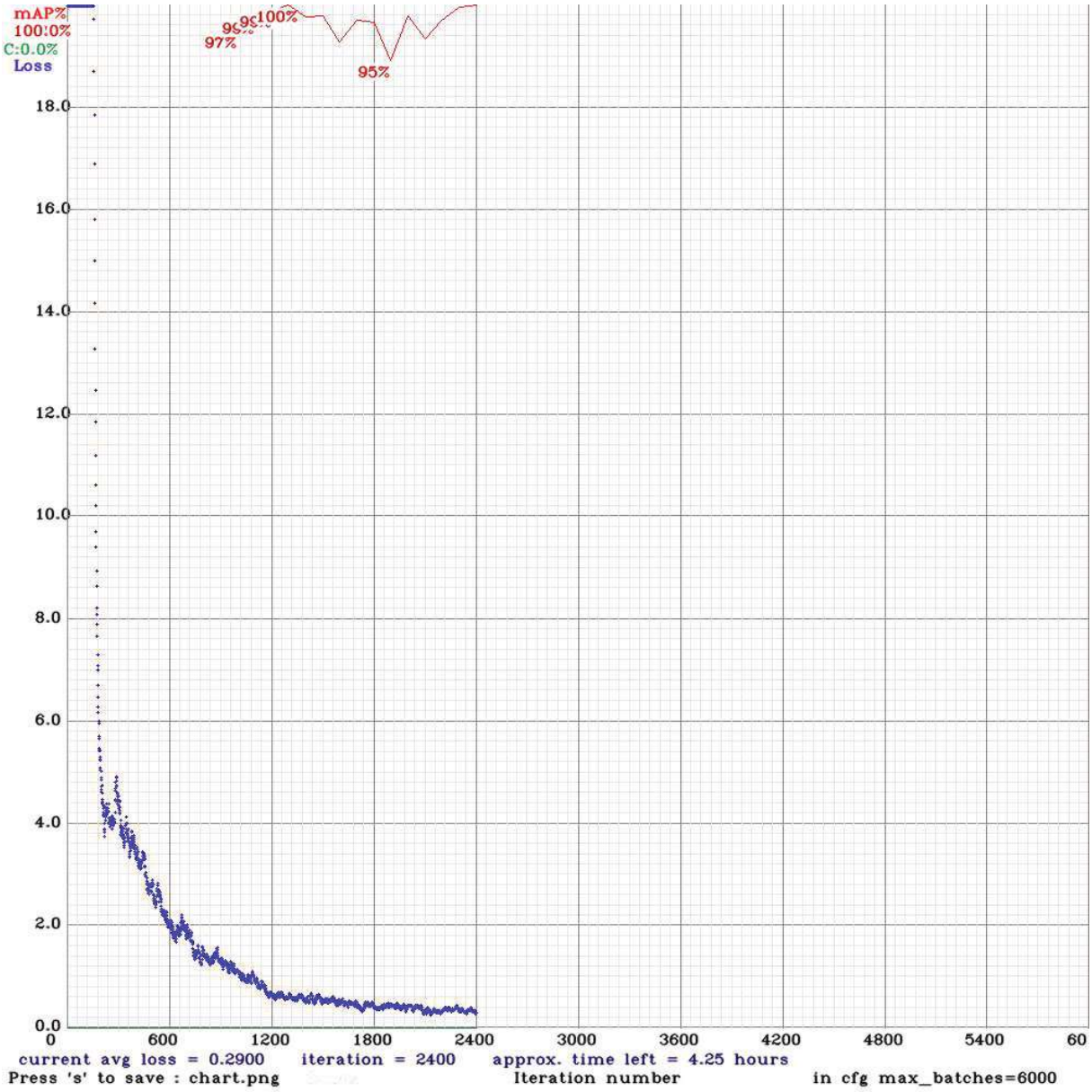


Abbildung 56: Ergebnisse der *Loss-Function* und mAP-Werte im Laufe des Trainings

5.5 Messung der Performance der Modelle

5.5.1 Messung der Performance auf Bildern

Zur Messung der Performance wurden sowohl in der VM *Colab* als auch lokal am Rechner Tests durchgeführt. Um die Performance testen zu können, werden das .cfg-File und das trainierte .weights-File benötigt. Dabei müssen in dem .cfg-File die Parameter des *Batches* und der *Subdivisions* auf eins gesetzt werden, da beim Testen jeweils nur ein Bild untersucht wird.

Für das Testen in der VM werden fünf Bilder verwendet, die mit einem iPhone aufgenommen wurden. Um die Performance lokal testen zu können, wurden 25 Bilder mit einer Webcam (Logitech BRIO) aufgenommen. Für die Aufnahme der Bilder wurde ein Python-Skript erstellt (siehe Anhang A), welches durch das Betätigen der Leertaste Bilder schnell aufnehmen kann. Weiters ist es möglich die Größe der aufgenommenen Bilder zu verändern. Der Grund für diese Vorgehensweise ist, dass bei einer möglichen Anwendung in der Industrie schnell und unkompliziert Bilder aufgenommen werden können. Die aufgenommenen Bilder für das Testen der Performance enthalten unterschiedliche Kombinationen der Objekte, um möglichst viele Szenarien testen zu können. Bei beiden Varianten (YOLOv3, YOLOv4) kann durch das Verändern des Schwellenwertes (*Threshhold*, siehe Kapitel 4.5) angegeben werden, ab welcher Wahrscheinlichkeit das Objekt auf dem Bild erkannt werden soll. Dieser wurde in der VM auf 0.5 und im lokalen Versuch auf 0.7 gesetzt. In Abbildung 57 wird das Szenario für das Aufnehmen der Bilder dargestellt. Dabei wird darauf geachtet, dass im Bild der Kamera nur der Lagerbehälter und dessen Inhalt aufgenommen wird.

Abbildung 57: Aufnahme der Bilder

Um die Bilder in der VM auswerten zu können, werden diese in den Google Drive Ordner geladen. Anhand der Befehle in Abbildung 58 a)YOLOv3, b)YOLOv4 wird das OD-Modell benutzt, um feststellen zu können, ob die eintrainierten Objekte auf dem Bild erkannt werden.

a)

```
# Start der Objekterkennung im Bild
!./darknet detector test data/obj.data cfg/yolov3_training.cfg /mydrive/Yolov3_3classes/backup/yolov3_training_last.weights /mydrive/Images/IMG_7382.JPG -thresh 0.5
```

b)

```
# Start der Objekterkennung im Bild
!./darknet detector test data/obj.data cfg/yolov4-obj.cfg /mydrive/Yolov4_3classes/backup/yolov4-obj_last.weights /mydrive/Images/IMG_2.JPG -thresh 0.5
```

Abbildung 58: Start der Objekterkennung in der VM

Für das Testen am lokalen Rechner wurden zwei weitere Python-Skripte (YOLOv3, YOLOv4) in Anlehnung an [71] erstellt (siehe Anhang B1 bzw. B2). In der folgenden Tabelle 6 sind die Ergebnisse aufgelistet. Die ersten fünf Bilder wurden in der VM ausgewertet und alle weiteren mittels des Python-Skriptes am lokalen Rechner. In der letzten Zeile der Tabelle ist der arithmetische Mittelwert (Durchschnitt) aller Werte angegeben. Bei einigen Bilder kam es vor, dass bei Verwendung von YOLOv3 oder YOLOv4 manche Objekte nicht erkannt wurden. Ein Grund dafür ist, dass die Objekte zu nahe aneinander oder teilweise überlappend auf den Bildern abgebildet waren. Diese Felder wurden in der Tabelle mit „nicht erkannt“ gekennzeichnet und mit einer Wahrscheinlichkeit von 0% für den Durchschnittswert berücksichtigt. Die leerstehenden Felder (mit / gekennzeichnet) in der Tabelle bedeuten, dass das jeweilige Objekt nicht am Foto abgebildet war.

Tabelle 6: Ergebnisse der Erkennungswahrscheinlichkeit

	Locher		Medikamentenschachtel		Zange	
Bildnummer	YOLOv3	YOLOv4	YOLOv3	YOLOv4	YOLOv3	YOLOv4
1	100%	100%	100%	100%	100%	100%
2	98%	97%	92%	100%	100%	100%
3	99%	100%	97%	100%	100%	100%
4	100%	100%	100%	100%	93%	100%
5	100%	100%	100%	100%	99%	98%
6	/	/	94,70%	99,80%	/	/
7	/	/	87,10%	99,70%	/	/
8	/	/	/	/	100%	99,90%
9	91%	99,60%	/	/	/	/
10	99,90%	99,80%	/	/	/	/
11	99,90%	99,80%	97,40%	99,80%	/	/
12	99%	99,30%	/	/	99,80%	99,60%
13	/	/	95,40%	99,50%	94%	99,10%
14	/	/	90,50%	99,10%	nicht erkannt	nicht erkannt
15	76,50%	nicht erkannt	nicht erkannt	99,10%	/	/
16	98,90%	98,20%	/	/	nicht erkannt	nicht erkannt
17	98,80%	99,60%	93,90%	99,60%	94,80%	97,80%
18	99,80%	99,60%	83,90%	99,80%	86,30%	96,90%
19	/	/	94,10%	99,80%	99,90%	98,60%
20	/	/	95%	99,80%	99,80%	97,50%
21	99,80%	98,80%	98,50%	99,90%	100%	99,90%
22	98,10%	99,80%	78,30%	98,20%	88,90%	99,10%
23	93,90%	99,50%	77,30%	99%	83,50%	99,10%
24	93,90%	99,80%	88,20%	99,60%	93,70%	99,10%
25	74,40%	99,60%	95,40%	99,70%	nicht erkannt	95,70%
26	99,80%	99,60%	86,30%	99,80%	83,80%	96,40%
27	81,30%	99,70%	98,80%	99,40%	91,10%	99,80%
28	81,90%	99,50%	98,20%	99,60%	94,80%	99,80%
29	99,10%	99,90%	80%	99,50%	95,30%	99,60%
30	98,50%	99,40%	91%	98,60%	88,10%	99,70%
Durchschnitt:	94,85%	95,15%	88,52%	99,57%	82,74%	90,65%

In Tabelle 6 sind die Durchschnittswerte in Tabellenform dargestellt. Es ist deutlich zu erkennen, dass das OD-Modell mit YOLOv4 (orange) bei der Erkennung besser abschneidet als YOLOv3 (blau). Das Objekt mit der höchsten Erkennungswahrscheinlichkeit mit YOLOv3 ist der Locher mit 94,85%. Hingegen das mit der niedrigsten Wahrscheinlichkeit ist die Zange mit 82,74%. Mit YOLOv4 liegen alle Erkennungswahrscheinlichkeiten bei über 90%, wobei die Medikamentenschachtel mit 99,57% die höchste Erkennungswahrscheinlichkeit aufweist.

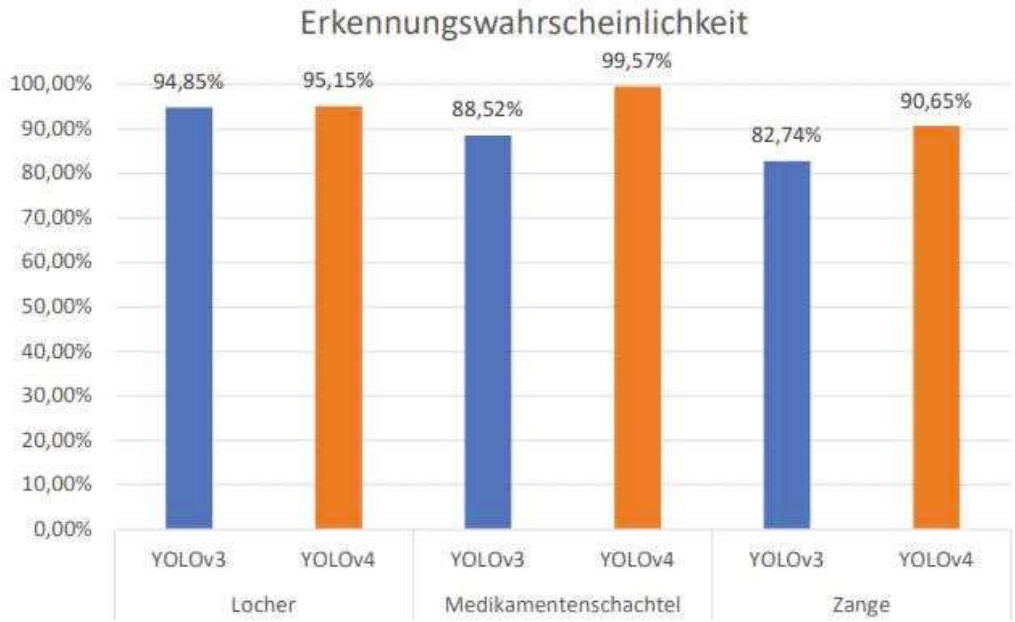


Abbildung 59: Darstellung der Durchschnittswerte der Erkennungswahrscheinlichkeit

In Abbildung 59 ist die Erkennungswahrscheinlichkeit nochmals anhand eines Balkendiagrammes aufgezeigt. Auf der y-Achse werden die Prozente der Wahrscheinlichkeiten dargestellt. Die x-Achse gibt die Objekte und das für die Erkennung verwendete OD-Modell an.

Mit der Formel in Kapitel 4.5 werden der *Recall* (Formel 13/15) und die *Precision* (Formel 14/16) für beide OD-Modelle berechnet.

- OD-Modell mit YOLOv3:

$$Recall_{YOLOv3} = \frac{TP_{YOLOv3}}{TP_{YOLOv3} + FN_{YOLOv3}} = 0{,}944 = 94{,}4\% \quad (13)$$

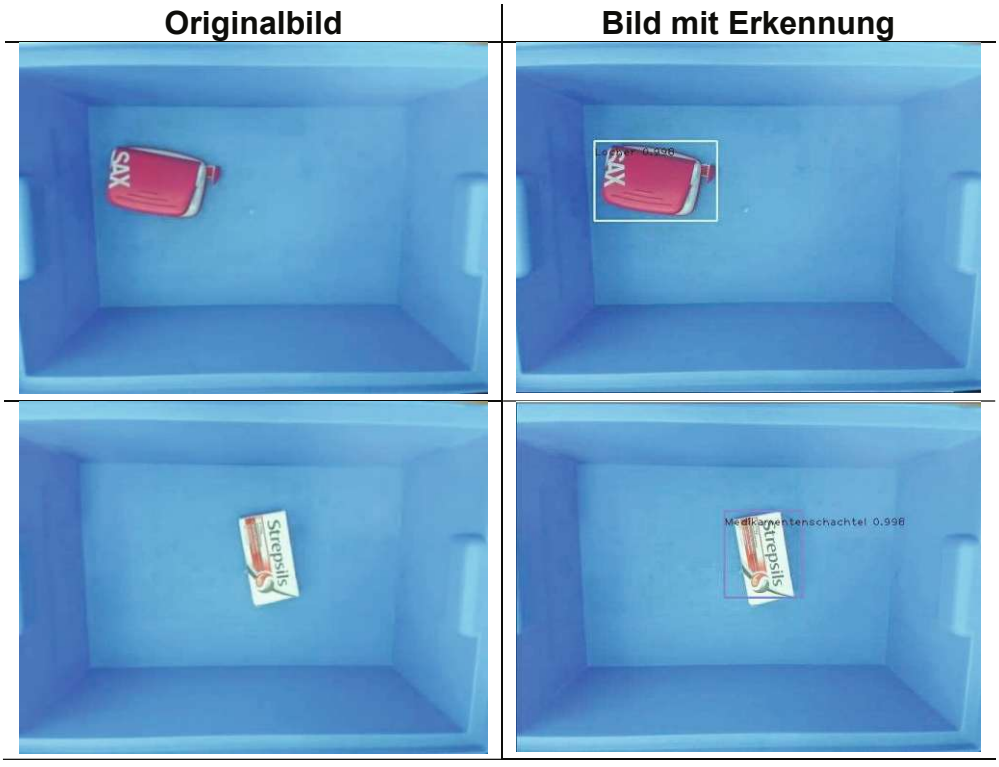
$$Precision_{YOLOv3} = \frac{TP_{YOLOv3}}{TP_{YOLOv3} + FP_{YOLOv3}} = 1 = 100\% \quad (14)$$

- OD-Modell mit YOLOv4:

$$Recall_{YOLOv4} = \frac{TP_{YOLOv4}}{TP_{YOLOv4} + FN_{YOLOv4}} = 0{,}958 = 95{,}8\% \quad (15)$$

$$Precision_{YOLOv4} = \frac{TP_{YOLOv4}}{TP_{YOLOv4} + FP_{YOLOv4}} = 1 = 100\% \quad (16)$$

Insgesamt schneidet das OD-Modell mit YOLOv4, mit einem *Recall* von 95,8% und einer *Precision* von 100%, besser ab als das OD-Modell mit YOLOv3, welches einen *Recall* von 94,4% und einer *Precision* von ebenfalls 100% aufweist. Der Wert der *Precision* ergibt bei dieser Versuchsreihe 100%, da bei der Auswertung keine FN aufgetreten sind. In Abbildung 60 werden einige Beispiele dafür angeführt, wie die Erkennung mittels der OD-Modelle auf dem Bild gekennzeichnet wird. Dabei wird dem Originalbild das Bild mit der Erkennung gegenübergestellt.



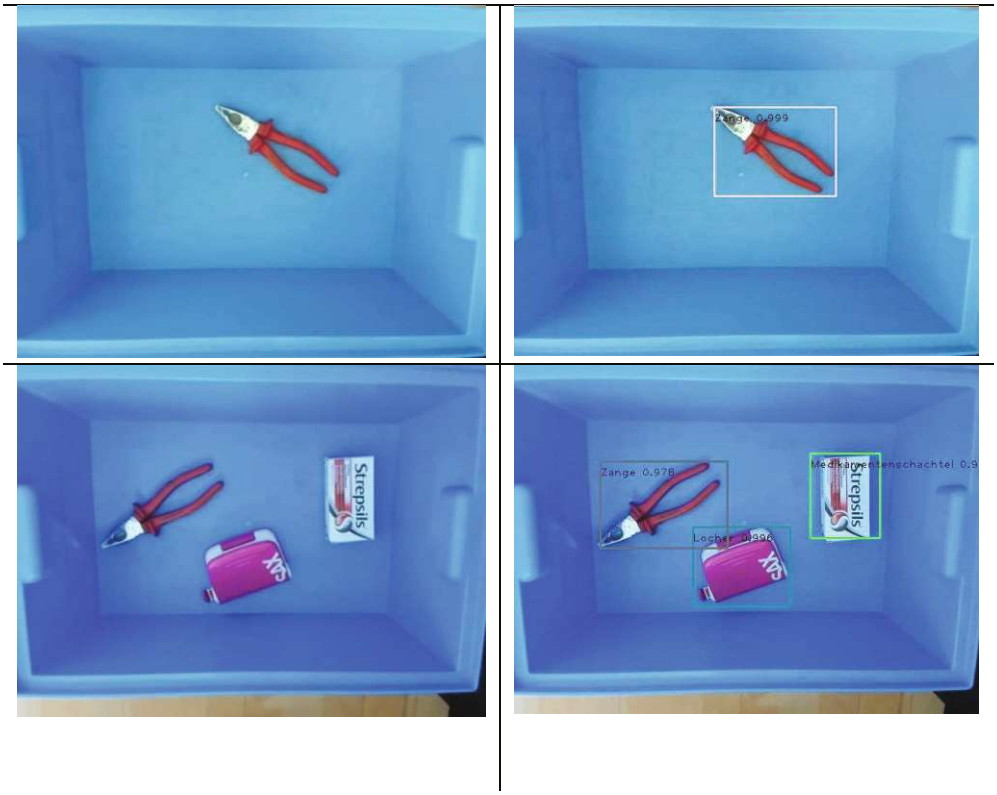


Abbildung 60: Gegenüberstellung der Originalbilder und der Bilder mit der Erkennung des OD-Modelles

Da für diesen speziellen Anwendungsfall genau die drei eintrainierten Objekte erkannt werden sollen und nicht eine grundsätzliche Erkennung eines beliebigen Objekts angestrebt wird, tendieren die OD-Modelle zu *Overfitting* (siehe Kapitel 2.3.5). Das bedeutet, dass das OD-Modell nicht in der Lage ist, die Objektarten Locher, Medikamentenschachtel und Zange im Allgemeinen zu erkennen, sondern nur diese drei speziellen Objekte erkennen kann. Da man in der Aufgabenstellung den Anwendungsfall so definiert hat, dass die Objekte, die sich im Kommissionierprozess bzw. im Lager befinden immer einem speziellen Typ von Objekt entsprechen sollen, stellt dies kein Problem dar, sondern ist sogar von Vorteil. Der wesentliche Vorteil ist, dass mit wenigen Trainingsdaten und einer damit verbundenen kurzen Trainingsphase sehr gute Ergebnisse erzielt werden können.

5.5.2 Messung der Performance im Live-Video

Um die OD-Modelle auch im Live-Video einsetzen zu können wurden für YOLOv3 und YOLOv4 jeweils ein Python-Skript erstellt (siehe Anhang C1 bzw. C2). Es wird dieselbe .cfg- und .weights-Datei, wie bei der Messung der Performance auf Bildern, verwendet, wobei in der .cfg-Datei die Parameter *Batch* und *Subdivisions* auf eins gestellt werden. Zur Testung des Systems wurde das Python-Skript gestartet und der Lagerbehälter,

in dem sich die Objekte befinden, bereitgestellt. Anschließend wurde der Inhalt des Lagerbehälters mit der darüber positionierten Kamera geprüft. Die Entfernung zwischen Box und Kamera betrug ca. 50 cm, sodass sich der gesamte Behälter immer im Blickfeld befand (siehe Abbildung 61). Während des Testens wurden die Objekte in unterschiedlichen Kombinationen in den Behälter gelegt.

Abbildung 61: Testen der OD-Modelle im Live-Video

Beim Testen mit YOLOv4 wurde festgestellt, dass die Erkennungswahrscheinlichkeit, wie beim Messen der Performance auf Bildern, bessere Werte erzielt als jene mit YOLOv3. Ein ausschlaggebender Punkt für das Testen per Live-Video sind die *frames per seconds* (fps). Die fps geben an, wie viele Bilder pro Sekunde verarbeitet werden. Dieser Wert ist ein Indikator dafür, wie schnell das OD-Modell die Bilder verarbeiten kann bzw. verarbeitet. Der fps-Wert hängt auch mit der CPU-Leistung des Rechners zusammen. Da für die Umsetzung ein Rechner mit einer niedrigen CPU-Leistung verwendet wurde, sind die fps-Werte gering ausgefallen. Dabei ergab sich für das OD-Modell mit YOLOv3 ein fps-Wert von maximal 0,72 fps und für das OD-Modell mit YOLOv4 ein Maximalwert von 0,6 fps.

Aufgrund der mangelnden Leistung des Rechners wurde eine Überprüfung mit einem Live-Video nicht weiterverfolgt.

5.6 Einsatz im Anwendungsgebiet

Um die kommissionierte Ware überprüfen zu können, reicht die Erkennung der Objekte nicht aus, da das Ergebnis jedes Mal manuell überprüft werden muss. Um dies zu vermeiden, wurde im verwendeten Python-Skript (siehe Anhang B1 bzw. B2) eine Funktion implementiert, die eine Liste der erkannten Objekte und deren Erkennungswahrscheinlichkeit erstellt. Da es üblich ist, dass ein Kommissionierprozess ausgelöst wird, sobald eine Bestellung aufgenommen wird, wird die Bestellung als Liste (Auftragsliste) an das Lager weitergegeben [72]. Diese Auftragsliste wird mit der durch den Erkennungsprozess erstellten Liste verglichen. Mögliche Differenzen werden identifiziert bzw. angezeigt. So kann überprüft werden, ob die kommissionierte Ware dem tatsächlichen Soll entspricht.

Um ein solches Szenario darstellen zu können, wird wie bereits erwähnt im Erkennungsprozess eine .txt-Datei erstellt. Die Ware, die kommissioniert werden soll, scheint in einer zweiten .txt-Liste auf (Auftragsliste). Mittels eines Python-Skripts (siehe Anhang D) werden diese zwei Listen miteinander verglichen und mögliche Differenzen festgestellt. Als erste Rückmeldung wird bei dem Vergleich der Listen ein Feedback erstellt, in welchem eine mögliche Differenz oder Übereinstimmung der beiden Listen angegeben wird. Weiters werden im Fall von Differenzen die Objekte, die in Liste 1 (erkannte Objekte des OD-Modelles) und in Liste 2 (Auftragsliste) nicht übereinstimmen, aufgezeigt. In folgender Tabelle 7 werden die möglichen Ausgaben der Überprüfung, die im Terminal ausgegeben werden, aufgezeigt.

Tabelle 7: Ausgaben des Python-Skripts bei der Überprüfung der Listen

Nr.	Ausgaben im Python-Skript
1. Ausgabe	• „Alle Objekte richtig kommissioniert!“ (grün hinterlegt) • „Achtung! Es liegt ein Fehler vor!“ (rot hinterlegt)
2. Ausgabe	• „Dieses Objekt fehlt in Liste 2:“
3. Ausgabe	• „Dieses Objekt ist in beiden Listen vorhanden:“
4. Ausgabe	• „Dieses Objekt fehlt in Liste 1:“

Die erste Ausgabe gibt an, ob alles richtig kommissioniert wurde oder ob ein Fehler vorliegt. Sollte alles richtig kommissioniert worden sein, bleiben die zweite und vierte Ausgabe leer. In der dritten Ausgabe werden jene Objekte aufgelistet, die sowohl in der ersten als auch in der zweiten Liste vorkommen. Wird in der ersten Ausgabe ein

Fehler bemerkt, so werden in der zweiten Ausgabe jene/s Objekt/e aufgelistet, welche/s erkannt worden sind, aber laut Liste2 nicht kommissioniert werden soll/en. In der vierten Ausgabe werden jene Objekte aufgelistet, welche/s nicht erkannt worden ist/sind, aber sehr wohl kommissioniert werden soll/en. Mit diesen Ausgaben kann überprüft werden, ob, wie viele und welche/r Fehler beim Kommissionieren aufgetreten sind. Im Falle einer fehlerhaften Kommissionierung, müssen die falsch kommissionierten Artikel bzw. Objekte korrigiert werden und können durch das OD-Modell nochmals überprüft werden. Der Überprüfungsprozesses ist in Abbildung 62 schematisch dargestellt.

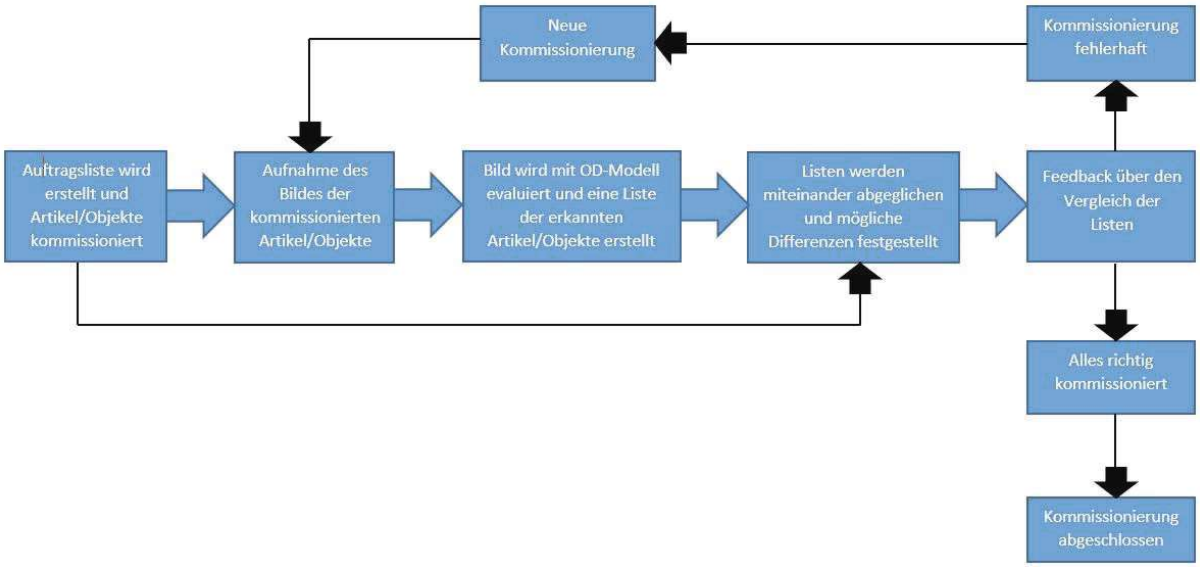


Abbildung 62: Darstellung des Überprüfungsprozesses der kommissionierten Artikel bzw. Objekte

6 Erkenntnisse der Implementierung

Im folgenden Kapitel wird auf die gewonnenen Erkenntnisse bzw. Herausforderungen während der Umsetzung des Praxisteiles eingegangen. Weiters werden die im Kapitel 1.2 definierten Forschungsfragen beantwortet.

6.1 Herausforderungen in der Umsetzung bzw. Implementierung der OD-Modelle

Bei der Umsetzung und Implementierung der OD-Modelle sind Herausforderungen und Probleme aufgetreten, die in der Umsetzung zu einem hohen Zeitaufwand geführt haben. Auf diese Herausforderungen und die vorgenommenen Maßnahmen, zur Beseitigung dieser, wird anschließend eingegangen.

Wie in Kapitel 5.3 beschrieben, ist ein Datensatz mit optimalen Bildern eine Grundvoraussetzung dafür, dass damit ein OD-Modell zufriedenstellende Ergebnisse liefern kann. Durch mehrere Versuche und Recherchen wurden folgende Eigenschaften, die ein Datensatz und dessen Daten aufweisen müssen, festgestellt:

- Beim Erstellen der Bilder müssen die zu erkennenden Objekte vor vielen verschiedenen Hintergründen abgebildet sein, damit das OD-Modell auf unterschiedliche Trainingsdaten zurückgreifen kann.
- Um ein ausgewogenes Training gewährleisten zu können, muss die Anzahl der Bilder für jedes Objekt gleich sein. Dies ist die Grundlage dafür, dass das OD-Modell im Idealfall bei der Erkennung annähernd gleiche Erkennungswahrscheinlichkeiten erzielt.
- Beim *Labeling*-Prozess mittels LabelImg ist darauf zu achten, dass die verwendeten Bilder ein gewisses Größenformat (3024x4032) nicht überschreiten. Während des Markierens der Objekte auf dem Bild werden die manuell erstellten BB zwar richtig gesetzt, aber die Koordinaten, die in der .txt- Datei eigetragen werden verschieben sich. Dies hat zur Folge, dass sich die Markierungen an falschen Stellen auf dem Bild befinden und somit das OD-Modell nicht richtig trainieren kann bzw. die Objekte beim Testen nur mit einer geringen Erkennungswahrscheinlicht oder gar nicht erkannt werden.
- Die classes.txt-Datei enthält die Namen der Klassen, die zeilenweise aufgelistet werden. Jede Zeile entspricht einer Nummer, wobei die erste Zeile der Nummer Null entspricht. Die Reihenfolge, in welcher die Objekte eingetragen sind, müssen mit der Reihenfolge in der obj.names-Datei übereinstimmen. Stimmen diese nicht überein, gibt das OD-Modell während des Trainings Fehlermeldungen aus.

Weitere Herausforderungen haben sich bei beim Ändern der Daten in der .cfg- und der obj.data-Datei ergeben:

- In der .cfg-Datei müssen alle notwendigen Parameter richtig gesetzt werden (siehe Kapitel 5.4), da es ansonsten zu Fehlermeldungen oder zu schlechten Ergebnissen beim Testen kommt.
- Die Pfade, die in der obj.data-Datei angegeben werden, müssen korrekt gesetzt werden. Hierbei ist eine sinnvolle Ordnerstruktur empfehlenswert.

Auch in der VM, die in Google *Colab* erstellt wurde, sind ein einige Dinge zu beachten:

- Die kostenfreie Nutzungsdauer von *Colab* beträgt zwölf Stunden, danach wird der Zugang gesperrt und der Benutzer muss einige Zeit warten, bis der Zugang erneut möglich ist. Dies stellt ein Problem dar, da das Training auch einige Tage lang dauern kann. Das Training kann aufgrund des Zwischenspeicherns der Gewichtungen vom letzten Trainingsstand aus fortgeführt werden. Jedoch geht bei einer Unterbrechung des Trainings die Aufzeichnung des Verlaufes der *Loss-Function* verloren.
- Befindet sich der Rechner zu lange im Ruhemodus, wird die Verbindung zu *Colab* ebenfalls getrennt. Für dieses Problem wurde in der Google Chrome *Console* ein Code geschrieben, der alle Zehn Minuten einen Mausklick simuliert.
- Die maximale Zugangsdauer zur VM variiert und ist von der Häufigkeit bzw. Dauer der Nutzung abhängig. D.h. es kann vorkommen, dass nach einer dreitägigen Nutzung der VM der Zugang für einige Zeit nicht mehr möglich ist.

Eine Einschränkung bezüglich der Geschwindigkeit der Erkennung sowohl auf den Bildern als auch im Live-Video kann die Leistung des Rechners darstellen.

6.2 Ergebnisse in Bezug auf die Forschungsfragen

Welche Möglichkeiten zum Erkennen von Objekten mit KI im Kommissionierprozess gibt es und welche signifikanten Unterschiede gibt es zwischen diesen?

Mittels KI können durch verschiedene Methoden Artikel und Objekte erkannt werden. Diese Methoden werden im Kapitel 2.8 aufgezeigt. Bei allen erwähnten OD-Modellen, die für die OD eingesetzt werden können, ist die Vorgehensweise bzw. die Funktionsweise sehr ähnlich. Der größte Unterschied ergibt sich aus dem Aufbau der verwendeten Netzwerke, wobei diese sich eher in der Größe oder Tiefe unterscheiden, die verwendeten Bausteine aber dieselben sind. Aufgrund dieses Unterschiedes können einige OD-Modelle bessere Ergebnisse liefern als andere.

Wie können die kommissionierten Objekte im Kommissionierprozess durch Bild- oder Videoaufnahmen mittels KI mit einer hohen Wahrscheinlichkeit erkannt werden?

Der Begriff der Künstlichen Intelligenz KI stellt ein großes Gebiet dar, welches die Themen Neuronale Netze, Maschinelles Lernen und *Deep Learning* umfasst. Für die Umsetzung und Realisierung dieser Arbeit wurden Neuronale Netze durch maschinelles Lernen so weit gebracht, dass diese mit einer hohen Wahrscheinlichkeit (über 91%) die zu kommissionierenden Objekte bzw. Artikel erkennen können. Die verwendeten Neuronalen Netze entsprechen den *Deep Neural Networks*, da diese viele Schichten (*Layer*) aufweisen. Diese Netzwerke, im Speziellen *Convolutional Neural Networks* (CNN), werden wie bereits erwähnt mittels eines Datensatzes für die zu erkennenden Objekte trainiert und können anschließend für die Erkennung auf Bildern oder Videoaufnahmen eingesetzt werden.

Wie könnte ein Prozess für die Umsetzung bzw. die Anwendung eines solchen Assistenzsystems im Unternehmen aussehen?

Eine mögliche Umsetzung, um eine solche Aufgabenstellung lösen zu können, wird im Kapitel 5 beschrieben. Dabei wird zu Beginn das NN für das Erkennen von Artikeln (Locher, Medikamentenschachtel und Zange) trainiert und ist anschließend in der Lage, diese auf Bildern und im Live-Video zu erkennen. Um die Richtigkeit der kommissionierten Artikel zu überprüfen, werden zwei Listen miteinander verglichen. Eine Liste entspricht der Auftragsliste, welche die zu kommissionierenden Artikel enthält und eine weitere Liste entspricht der Ergebnisliste, die das OD-Modell während der Erkennung erstellt. Anschließend werden beide Listen miteinander abgeglichen und mögliche Differenzen aufgezeigt. Sobald die Listen nicht übereinstimmen, muss der Kommissionierprozess erneut durchlaufen werden (siehe Abbildung 65).

7 Zusammenfassung und Ausblick

Die Zusammenfassung geht auf die Kernaussagen dieser Arbeit ein und fasst die Ergebnisse des Praxisteiles nochmals zusammen. Des Weiteren wird am Ende dieser Arbeit ein Ausblick auf mögliche Weiterentwicklungen bzw. Verbesserungen der Umsetzung gegeben.

7.1 Zusammenfassung

In der vorliegenden Arbeit wird ein Lösungskonzept präsentiert, welches es durch den Einsatz von KI ermöglicht, Objekte während des Kommissionierprozesses zu überprüfen. Zu Beginn der Arbeit wurden die theoretischen Hintergründe bzw. Funktionsweisen der OD aufgezeigt. In der State-of-the-Art-Analyse wurden verschiedene Einsatzgebiete vor allem in der Logistik vorgestellt. Auf Basis der OD-Modelle YOLOv3 und YOLOv4 wurde für diese Arbeit ein Modell für die Erkennung von drei spezifischen Objekten (Locher, Medikamentenschachtel und Zange) entwickelt. Das Konzept des Modells wurde in Kapitel 4 Schritt-für-Schritt vorgestellt und in Kapitel 5 in die Praxis umgesetzt und implementiert. Das Training wurde in einer VM auf *Colab* durchgeführt. Anschließend wurden die OD-Modelle auf ihre Performance bei der Erkennung auf Bildern und im Live-Video getestet, indem ein Python-Skript in der IDE PyCharm erstellt wurde. Dabei lieferte das OD-Modell mit YOLOv4 deutlich bessere Ergebnisse als jenes mit YOLOv3. Um die Objekte im Kommissionierprozess überprüfen zu können, erstellt das Python-Skript bei jeder Erkennung eine .txt-Liste, in der die erkannten Objekte aufgelistet werden. Diese wird mit einer zweiten Liste, die die Auftragsliste darstellen soll mittels eines Python-Skripts verglichen, um mögliche Fehler, die beim Kommissionieren auftreten können, feststellen zu können.

Um die Erkennungswahrscheinlichkeit zu steigern, wurden bewusst drei möglichst unterschiedliche Objekte verwendet. Aufgrund der erheblichen Unterschiede der Objekte, war es möglich, mit einer geringen Anzahl an Daten aussagekräftige Ergebnisse zu erzielen. Da das Training der OD-Modelle mit einer geringen Anzahl an Daten durchgeführt wurde, haben diese Schwierigkeiten, ähnliche Objekte zu unterscheiden bzw. erkennen Objekte falsch, sobald es sich nicht um die drei spezifischen Objekte handelt. Um diesen Schwierigkeiten entgegenzuwirken, wurde versucht *Data Augmentation* anzuwenden. Dies lieferte aber keine zufriedenstellenden Ergebnisse. Dennoch weisen in der vorgelegten Arbeit beide OD-Modelle eine hohe Erkennungswahrscheinlichkeit auf. Bei YOLOv3 liegt diese bei über 82% und bei YOLOv4 sogar bei über 90%. Dies kann als sehr zufriedenstellendes Ergebnis betrachtet werden.

7.2 Ausblick

Da die OD-Modelle mit YOLOv3 und YOLOv4 Schwierigkeiten haben, zwischen ähnlichen Objekten differenzieren zu können, könnte im nächsten Schritt der Datensatz erweitert werden, um die Erkennungswahrscheinlichkeit zu erhöhen.

Des Weiteren kann die Erkennung mittels Live-Videos weiterverfolgt werden. Ein dafür mögliches Szenario für den Einsatz dieser Technologie, wäre die Erkennung von Objekten am Fließband bzw. Gurtförderband. Dazu würde ein Rechner mit hoher Rechenleistung benötigt werden.

Da viele Unternehmen für die Produktverwaltung PLM-Systeme (*Product Lifecycle Management*-Systeme) benutzen, muss eine Schnittstelle zwischen diesen und der Erkennungssoftware (OD-Modell) geschaffen werden. Dies kann in ähnlicher Art und Weise realisiert werden, wie es in der vorliegenden Arbeit aufgezeigt wurde. So könnte das PLM-System eine .txt-Datei mit den geforderten Objekten erstellen und diese dem OD-Modell übermitteln. Diese Liste, in dieser Arbeit als Auftragsliste betitelt, wird mit der erstellten Liste des OD-Modelles abgeglichen, um mögliche Differenzen festzustellen.

Aufgrund des hohen Zeitaufwandes des *Labeling*-Prozesses, sollte dessen Automatisierung angestrebt werden. Dafür gibt es bereits einige Ansätze wie z.B. [51], die diesen Prozess zum größten Teil automatisieren. Einige Schritte müssen aber immer noch manuell vollzogen werden. Ein vollkommen automatisierter *Labeling*-Prozess würde zur Folge haben, dass OD-Modelle für verschiedene Sparten zugänglicher werden.

Eine zusätzliche Kontrolle im Zusammenspiel mit der Objekterkennung könnte die Erfassung des Gewichtes der kommissionierten Artikel sein. Dieses wird auf die Plausibilität des Gesamtgewichtes geprüft, indem ein vorgegebenes Soll-Gewicht mit dem Ist-Gewicht verglichen wird.

Abschließend lässt sich anhand der vorliegenden Arbeit sagen, dass mit dieser Umsetzung und den daraus resultierenden Ergebnissen der Einsatz von KI für die Überprüfung von Artikeln im Kommissionierprozess ein hohes Potenzial besitzt.

Anhang

A. Aufnahme der Bilder

```
1  import cv2
2
3  cam = cv2.VideoCapture(1, cv2.CAP_DSHOW)
4
5  cv2.namedWindow("test")
6
7  img_counter = 0
8
9  while True:
10     ret, frame = cam.read()
11     if not ret:
12         print("failed to grab frame")
13         break
14     cv2.imshow("test", frame)
15
16     k = cv2.waitKey(1)
17     if k%256 == 27:
18         # ESC drücken
19         print("Escape hit, closing...")
20         break
21     elif k%256 == 32:
22         # SPACE drücken
23         img_name = "Name Bild .jpeg".format(img_counter)
24         cv2.imwrite(img_name, frame)
25         print("{} written!".format(img_name))
26         img_counter += 1
27
28 cam.release()
```

B1. Bilderkennung mit YOLOv3

```
1  import cv2
2  import numpy as np
3
4
5  net = cv2.dnn.readNetFromDarknet('yolov3_testing.cfg', r'Pfad .weights-
6  Datei')
7
8  classes = []
9  with open('classes_3.txt', 'r') as f:
10     classes = f.read().splitlines()
11
12 img = cv2.imread('Pfad Bilddatei')
13 hight, width, _ = img.shape
14
15 blob = cv2.dnn.blobFromImage(img, 1/255,(416,416),(0,0,0),swapRB =
16 True,crop= False)
```

```
17
18 net.setInput(blob)
19
20 output_layers_name = net.getUnconnectedOutLayersNames()
21 layerOutputs = net.forward(output_layers_name)
22
23 boxes = []
24 confidences = []
25 class_ids = []
26
27 for output in layerOutputs:
28     for detection in output:
29         scores = detection[5:]
30         class_id = np.argmax(scores)
31         confidence = scores[class_id]
32         if confidence > 0.7:
33             center_x = int(detection[0] * width)
34             center_y = int(detection[1] * hight)
35             w = int(detection[2] * width)
36             h = int(detection[3] * hight)
37
38             x = int(center_x - w / 2)
39             y = int(center_y - h / 2)
40
41             boxes.append([x, y, w, h])
42             confidences.append((float(confidence)))
43             class_ids.append(class_id)
44
45 print(len(boxes))
46 indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.7, 0.1)
47 print(indexes.flatten())
48
49 font = cv2.FONT_HERSHEY_PLAIN
50 colors = np.random.uniform(0, 255, size=(len(boxes), 3))
51 if len(indexes) > 0:
52     for i in indexes.flatten():
53         x, y, w, h = boxes[i]
54         label = str(classes[class_ids[i]])
55         confidence = str(round(confidences[i], 3))
56         color = colors[i]
57         cv2.rectangle(img, (x,y), (x+w, y+h), color, 2)
58         cv2.putText(img, label + " " + confidence, (x, y+20), font, 1,
59 (0,0,0), 1)
60         print(label, confidence)
61         file = open ("Liste.txt", "a")
62         file.write(label + '\n')
63         #file.write("  " + confidence + '\n')
64         file.close()
65
66 cv2.imshow('Image', img)
67 cv2.waitKey(0)
68 cv2.destroyAllWindows()
```

B2. Bilderkennung mit YOLOv4

```
import cv2
import numpy as np


net = cv2.dnn.readNetFromDarknet('yolov4-obj.cfg', r'Pfad .weights-Datei')

classes = []
with open('classes_3.txt', 'r') as f:
    classes = f.read().splitlines()

img = cv2.imread('Pfad Bilddatei')
hight, width, _ = img.shape

blob = cv2.dnn.blobFromImage(img, 1/255,(416,416),(0,0,0),swapRB =
True,crop= False)

net.setInput(blob)

output_layers_name = net.getUnconnectedOutLayersNames()
layerOutputs = net.forward(output_layers_name)

boxes = []
confidences = []
class_ids = []

for output in layerOutputs:
    for detection in output:
        scores = detection[5:]
        class_id = np.argmax(scores)
        confidence = scores[class_id]
        if confidence > 0.7:
            center_x = int(detection[0] * width)
            center_y = int(detection[1] * hight)
            w = int(detection[2] * width)
            h = int(detection[3] * hight)

            x = int(center_x - w / 2)
            y = int(center_y - h / 2)

            boxes.append([x, y, w, h])
            confidences.append((float(confidence)))
            class_ids.append(class_id)

print(len(boxes))
indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.7, 0.1)
print(indexes.flatten())

font = cv2.FONT_HERSHEY_PLAIN
colors = np.random.uniform(0, 255, size=(len(boxes), 3))
if len(indexes) > 0:
    for i in indexes.flatten():
        x, y, w, h = boxes[i]
        label = str(classes[class_ids[i]])
        confidence = str(round(confidences[i], 3))
        color = colors[i]
```

```
56          cv2.rectangle(img, (x,y), (x+w, y+h), color, 2)
57          cv2.putText(img, label + " " + confidence, (x, y+20), font, 1,
58 (0,0,0), 1)
59          print(label, confidence)
60          file = open ("Liste.txt", "a")
61          file.write(label + '\n')
62          #file.write("  " + confidence + '\n')
63          file.close()
64
65 cv2.imshow('Image', img)
66 cv2.waitKey(0)
67 cv2.destroyAllWindows()
```

C1. Live-Video Erkennung YOLOv3

```
 1  import cv2
 2  import numpy as np
 3  import time
 4
 5  net = cv2.dnn.readNetFromDarknet('yolov3_testing.cfg', r'Pfad
 6  .weights-Datei'
 7
 8  classes = []
 9  with open('classes_3.txt', 'r') as f:
10      classes = f.read().splitlines()
11
12
13  cap = cv2.VideoCapture(1, cv2.CAP_DSHOW)
14  starting_time = time.time()
15  frame_id = 0
16
17  while True:
18      _, img = cap.read()
19      hight, width, _ = img.shape
20      frame_id += 1
21
22
23      blob = cv2.dnn.blobFromImage(img, 1/255,(416,416),(0,0,0),swapRB = True,crop=
24      net.setInput(blob)
25      output_layers_name = net.getUnconnectedOutLayersNames()
26      layerOutputs = net.forward(output_layers_name)
27
28      boxes = []
29      confidences = []
30      class_ids = []
31
32      for output in layerOutputs:
33          for detection in output:
34              scores = detection [5:]
35              class_id = np.argmax(scores)
36              confidence = scores[class_id]
37              if confidence > 0.2:
38                  center_x = int(detection[0] * width)
39                  center_y = int(detection[1] * hight)
40                  w = int(detection[2] * width)
```

```
                h = int(detection[3] * hight)

                x = int(center_x - w / 2)
                y = int(center_y - h / 2)

                boxes.append([x, y, w, h])
                confidences.append((float(confidence)))
                class_ids.append(class_id)

    indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.2, 0.1)

    font = cv2.FONT_HERSHEY_PLAIN
    colors = np.random.uniform(0, 255, size=(len(boxes), 3))
    if len(indexes) > 0:
        for i in indexes.flatten():
            x, y, w, h = boxes[i]
            label = str(classes[class_ids[i]])
            confidence = str(round(confidences[i], 2))
            color = colors[i]
            cv2.rectangle(img, (x, y), (x+w, y+h), color, 2)
            cv2.putText(img, label + " " + confidence, (x, y+20),
font, 1, (255,255,255), 1)

    elapsed_time = time.time() - starting_time
    fps = frame_id / elapsed_time
    cv2.putText(img, "FPS: " + str(fps), (10, 39), font, 3,
 (0,0,0), 1)

    cv2.imshow('frame', img)
    key = cv2.waitKey(1)
    if key==27:
        break
cap.release()
cv2.destroyAllWindows()
```

C2. Live-Video Erkennung YOLOv4

```
import cv2
import numpy as np
import time

net = cv2.dnn.readNetFromDarknet('yolov4-obj.cfg', r'Pfad
.weights-Datei'

classes = []
with open('classes_3.txt', 'r') as f:
    classes = f.read().splitlines()


cap = cv2.VideoCapture(1, cv2.CAP_DSHOW)
starting_time = time.time()
frame_id = 0

while True:
```

```
18      _, img = cap.read()
19      hight, width, _ = img.shape
20      frame_id += 1
21
22
23      blob = cv2.dnn.blobFromImage(img, 1/255,(416,416),(0,0,0),swapRB = True,crop=
24      net.setInput(blob)
25      output_layers_name = net.getUnconnectedOutLayersNames()
26      layerOutputs = net.forward(output_layers_name)
27
28      boxes = []
29      confidences = []
30      class_ids = []
31
32      for output in layerOutputs:
33          for detection in output:
34              scores = detection [5:]
35              class_id = np.argmax(scores)
36              confidence = scores[class_id]
37              if confidence > 0.2:
38                  center_x = int(detection[0] * width)
39                  center_y = int(detection[1] * hight)
40                  w = int(detection[2] * width)
41                  h = int(detection[3] * hight)
42
43                  x = int(center_x - w / 2)
44                  y = int(center_y - h / 2)
45
46                  boxes.append([x, y, w, h])
47                  confidences.append((float(confidence)))
48                  class_ids.append(class_id)
49
50      indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.2, 0.1)
51
52      font = cv2.FONT_HERSHEY_PLAIN
53      colors = np.random.uniform(0, 255, size=(len(boxes), 3))
54      if len(indexes) > 0:
55          for i in indexes.flatten():
56              x, y, w, h = boxes[i]
57              label = str(classes[class_ids[i]])
58              confidence = str(round(confidences[i], 2))
59              color = colors[i]
60              cv2.rectangle(img, (x, y), (x+w, y+h), color, 2)
61              cv2.putText(img, label + " " + confidence, (x, y+20),
62  font, 1, (255,255,255), 1)
63
64      elapsed_time = time.time() - starting_time
65      fps = frame_id / elapsed_time
66      cv2.putText(img, "FPS: " + str(fps), (10, 39), font, 3,
67   (0,0,0), 1)
68
69      cv2.imshow('frame', img)
70      key = cv2.waitKey(1)
71      if key==27:
72          break
73  cap.release()
74  cv2.destroyAllWindows()
```

D. Listenvergleich

```
1  from termcolor import colored
2
3
4 with open('Liste.txt','r') as f1:
5     L1=f1.read().splitlines()
6 L1.sort()
7
8 with open('Liste2.txt','r') as f2:
9     L2=f2.read().splitlines()
10 L2.sort()
11
12
13 nicht_gemeinsam = []
14 gemeinsam = []
15 nur_in_L2 = []
16
17 for i in L1:
18     if i not in L2:
19         nicht_gemeinsam.append(i)
20     elif i in L1:
21         gemeinsam.append(i)
22
23 for i in L2:
24     if i not in L1:
25         nur_in_L2.append(i)
26
27 if L1 == L2:
28     print (colored('Alle Objekte richtig kommssioniert!','green',))
29 else:
30     print (colored('Achtung! Es liegt ein Fehler vor!','red',))
31 print('\n')
32
33 print ('Dieses Objekt fehlt in Liste 2: ')
34 print(nicht_gemeinsam,'\n')
35
36 print('Dieses Objekt ist in beiden Listen vorhanden: ')
37 print(gemeinsam,'\n')
38
39 print('Dieses Objekt fehlt in Liste 1: ')
40 print(nur_in_L2,'\n')
```

Literaturverzeichnis

[1] A. Younis, L. Shixin, S. Jn, und Z. Hai, „Real-Time Object Detection Using Pre-Trained Deep Learning Models MobileNet-SSD“, in *Proceedings of 2020 the 6th International Conference on Computing and Data Engineering*, New York, NY, USA, Jän. 2020, S. 44–48. doi: 10.1145/3379247.3379264.

[2] S. Lu, B. Wang, H. Wang, L. Chen, M. Linjian, und X. Zhang, „A real-time object detection algorithm for video“, *Comput. Electr. Eng.*, Bd. 77, S. 398–408, Juli 2019, doi: 10.1016/j.compeleceng.2019.05.009.

[3] J. Dümmel, M. Hochstein, J. Glöckle, und K. Furmans, „Effizientes Labeln von Artikeln für das Einlernen Künstlicher Neuronaler Netze“, *Logist. J. Proc.*, Bd. 2019, Nr. 12, Dez. 2019, doi: 10.2195/lj_Proc_duemmel_de_201912_01.

[4] R. T. Kreutzer und M. Sirrenberg, „Was versteht man unter Künstlicher Intelligenz und wie kann man sie nutzen?“, in *Künstliche Intelligenz verstehen: Grundlagen – Use-Cases – unternehmenseigene KI-Journey*, R. T. Kreutzer und M. Sirrenberg, Hrsg. Wiesbaden: Springer Fachmedien, 2019, S. 1–71. doi: 10.1007/978-3-658-25561-9_1.

[5] A. Rosebrock, *Deep Learning for Computer Vision with Python: Starter Bundle*. PyImageSearch, 2017.

[6] M. DelSole, „What is One Hot Encoding and How to Do It“, *Medium*, Apr. 24, 2018. https://medium.com/@michaeldelsole/what-is-one-hot-encoding-and-how-to-do-it-f0ae272f1179 (zugegriffen Apr. 29, 2021).

[7] D. Stodolkiewicz, „One Hot Encoding and Label Encoding, Data Science & Machine Learning“, *Data Science & Machine Learning*. http://www.stodolkiewicz.com/2019/12/16/one-hot-encoding/ (zugegriffen Apr. 29, 2021).

[8] V. Bushaev, „How do we ‘train’ neural networks ?“, *Medium*, Okt. 22, 2018. https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73 (zugegriffen Apr. 29, 2021).

[9] „Softmax Function“, *DeepAI*, Mai 17, 2019. https://deepai.org/machine-learning-glossary-and-terms/softmax-layer (zugegriffen Apr. 30, 2021).

[10] S. SHARMA, „Epoch vs Batch Size vs Iterations“, *Medium*, März 05, 2019. https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9 (zugegriffen Mai 02, 2021).

[11] I. Goodfellow, Y. Bengio, und A. Courville, *Deep Learning*. MIT Press, 2016.

[12] „CS231n Convolutional Neural Networks for Visual Recognition“. https://cs231n.github.io/convolutional-networks/ (zugegriffen März 12, 2021).

[13] „A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way | by Sumit Saha | Towards Data Science“. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (zugegriffen Apr. 30, 2021).

[14] Jefkine, „Backpropagation In Convolutional Neural Networks“, *DeepGrid*, Sep. 05, 2016. https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/ (zugegriffen Okt. 10, 2021).

[15] T. Balodi, „Convolutional Neural Network with Python Code Explanation | Convolutional Layer | Max Pooling in CNN“. https://www.analyticssteps.com/blogs/convolutional-neural-network-cnn-graphical-visualization-code-explanation (zugegriffen Mai 01, 2021).

[16] B. Curry, „An Introduction to Transfer Learning and Meta-Learning in Machine Learning and Artificial…“, *Medium*, Juni 14, 2020. https://medium.com/kansas-city-

machine-learning-artificial-intelligen/an-introduction-to-transfer-learning-in-machine-learning-7efd104b6026 (zugegriffen Mai 02, 2021).
[17] „Object Detection Guide | Fritz AI“. https://www.fritz.ai/object-detection/ (zugegriffen Apr. 24, 2021).
[18] Z. Zhao, P. Zheng, S. Xu, und X. Wu, „Object Detection With Deep Learning: A Review“, *IEEE Trans. Neural Netw. Learn. Syst.*, Bd. 30, Nr. 11, S. 3212–3232, Nov. 2019, doi: 10.1109/TNNLS.2018.2876865.
[19] W. Liu *u. a.*, „SSD: Single Shot MultiBox Detector“, *ArXiv151202325 Cs*, Dez. 2016, doi: 10.1007/978-3-319-46448-0_2.
[20] T.-Y. Lin, P. Goyal, R. Girshick, K. He, und P. Dollár, „Focal Loss for Dense Object Detection“, *ArXiv170802002 Cs*, Feb. 2018, Zugegriffen: Okt. 10, 2021. [Online]. Verfügbar unter: http://arxiv.org/abs/1708.02002
[21] J. Dai, Y. Li, K. He, und J. Sun, „R-FCN: Object Detection via Region-based Fully Convolutional Networks“, *ArXiv160506409 Cs*, Juni 2016, Zugegriffen: Okt. 10, 2021. [Online]. Verfügbar unter: http://arxiv.org/abs/1605.06409
[22] J. S. Jun 04 und 2020 10 Min Read □ □ □ □, „Breaking Down YOLOv4“, *Roboflow Blog*, Juni 04, 2020. https://blog.roboflow.com/a-thorough-breakdown-of-yolov4/ (zugegriffen März 02, 2021).
[23] R. Girshick, J. Donahue, T. Darrell, und J. Malik, „Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation“, 2014, S. 580–587. Zugegriffen: Apr. 09, 2021. [Online]. Verfügbar unter: https://openaccess.thecvf.com/content_cvpr_2014/html/Girshick_Rich_Feature_Hierarchies_2014_CVPR_paper.html
[24] R. Gandhi, „R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms“, *Medium*, Juli 09, 2018. https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e (zugegriffen Apr. 21, 2021).
[25] „Faster R-CNN Explained for Object Detection Tasks“, *Paperspace Blog*, Nov. 13, 2020. https://blog.paperspace.com/faster-r-cnn-explained-object-detection/ (zugegriffen Apr. 22, 2021).
[26] R. Girshick, „Fast R-CNN“, 2015, S. 1440–1448. Zugegriffen: Apr. 09, 2021. [Online]. Verfügbar unter: https://openaccess.thecvf.com/content_iccv_2015/html/Girshick_Fast_R-CNN_ICCV_2015_paper.html
[27] S. Ren, K. He, R. Girshick, und J. Sun, „Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks“, *ArXiv150601497 Cs*, Jän. 2016, Zugegriffen: Apr. 09, 2021. [Online]. Verfügbar unter: http://arxiv.org/abs/1506.01497
[28] A. Choudhury, „Top 8 Algorithms For Object Detection One Must Know“, *Analytics India Magazine*, Juni 16, 2020. https://analyticsindiamag.com/top-8-algorithms-for-object-detection/ (zugegriffen Feb. 27, 2021).
[29] J. Redmon, S. Divvala, R. Girshick, und A. Farhadi, „You Only Look Once: Unified, Real-Time Object Detection“, 2016, S. 779–788. Zugegriffen: Feb. 17, 2021. [Online]. Verfügbar unter: https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Redmon_You_Only_Look_CVPR_2016_paper.html
[30] A. Bochkovskiy, C.-Y. Wang, und H.-Y. M. Liao, „YOLOv4: Optimal Speed and Accuracy of Object Detection“, *ArXiv200410934 Cs Eess*, Apr. 2020, Zugegriffen: Feb. 14, 2021. [Online]. Verfügbar unter: http://arxiv.org/abs/2004.10934

[31] „Intersection over Union (IoU) for object detection“, *PyImageSearch*, Nov. 07, 2016. https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/ (zugegriffen Juli 23, 2021).
[32] „Python Lessons“. https://pylessons.com/YOLOv3-introduction/ (zugegriffen März 12, 2021).
[33] D. Kapil, „YOLO v1: Part 2“, *Medium*, Okt. 06, 2018. https://medium.com/@divakar_239/yolo-v1-part-2-bfc686ae5560 (zugegriffen März 06, 2021).
[34] J. Redmon und A. Farhadi, „YOLO9000: Better, Faster, Stronger“, 2017, S. 7263–7271. Zugegriffen: Feb. 17, 2021. [Online]. Verfügbar unter: https://openaccess.thecvf.com/content_cvpr_2017/html/Redmon_YOLO9000_Better_Faster_CVPR_2017_paper.html
[35] „Understanding YOLO and YOLOv2“, *Manal El Aidouni*, Juni 25, 2019. manalelaidouni.github.io/Understanding%20YOLO%20and%20YOLOv2.html (zugegriffen März 30, 2021).
[36] Seong, S. Song, J.-H. Yoon, J. Kim, und C.-S. Choi, „Determination of Vehicle Trajectory through Optimization of Vehicle Bounding Boxes Using a Convolutional Neural Network“, *Sensors*, Bd. 19, S. 4263, Sep. 2019, doi: 10.3390/s19194263.
[37] Y. Y. / SPARKLE / @sparkle_twtt, „How does YOLOv2 work“, *Medium*, Dez. 28, 2018. https://sparkle-mdm.medium.com/how-does-yolov2-work-daaaa967c5f7 (zugegriffen März 30, 2021).
[38] J. Redmon und A. Farhadi, „YOLOv3: An Incremental Improvement“, *ArXiv180402767 Cs*, Apr. 2018, Zugegriffen: Feb. 14, 2021. [Online]. Verfügbar unter: http://arxiv.org/abs/1804.02767
[39] A. Kathuria, „What’s new in YOLO v3?“, *Medium*, Apr. 29, 2018. https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b (zugegriffen Feb. 25, 2021).
[40] E. Y. Li, „Dive Really Deep into YOLO v3: A Beginner’s Guide“, *Medium*, März 17, 2020. https://towardsdatascience.com/dive-really-deep-into-yolo-v3-a-beginners-guide-9e3d2666280e (zugegriffen Feb. 25, 2021).
[41] N. Adaloglou, „Intuitive Explanation of Skip Connections in Deep Learning“, *AI Summer*, März 23, 2020. https://theaisummer.com/skip-connections/ (zugegriffen Aug. 16, 2021).
[42] M. Mantripragada, „Digging deep into YOLO V3 — A hands-on guide Part 1“, *Medium*, Aug. 18, 2020. https://towardsdatascience.com/digging-deep-into-yolo-v3-a-hands-on-guide-part-1-78681f2c7e29 (zugegriffen Apr. 09, 2021).
[43] J. Cohen, „Introduction to YOLOv4: Research review“, *Medium*, Okt. 15, 2020. https://heartbeat.fritz.ai/introduction-to-yolov4-research-review-5b6b4bd5f255 (zugegriffen März 02, 2021).
[44] P. RUGERY, „Explaining YoloV4 a one stage detector“, *Medium*, Sep. 15, 2020. https://becominghuman.ai/explaining-yolov4-a-one-stage-detector-cdac0826cbd7 (zugegriffen März 02, 2021).
[45] G. Huang, Z. Liu, L. van der Maaten, und K. Q. Weinberger, „Densely Connected Convolutional Networks“, *ArXiv160806993 Cs*, Jän. 2018, Zugegriffen: Apr. 14, 2021. [Online]. Verfügbar unter: http://arxiv.org/abs/1608.06993
[46] M. Tan und Q. V. Le, „EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks“, *ArXiv190511946 Cs Stat*, Sep. 2020, Zugegriffen: Apr. 14, 2021. [Online]. Verfügbar unter: http://arxiv.org/abs/1905.11946
[47] K. K. Santhosh, D. P. Dogra, und P. P. Roy, „Anomaly Detection in Road Traffic Using Visual Surveillance: A Survey“, *ACM Comput. Surv.*, Bd. 53, Nr. 6, S. 119:1-119:26, Dez. 2020, doi: 10.1145/3417989.

[48] M. Peker, „Comparison of Tensorflow Object Detection Networks for Licence Plate Localization“, in *2019 1st Global Power, Energy and Communication Conference (GPECOM)*, Juni 2019, S. 101–105. doi: 10.1109/GPECOM.2019.8778602.
[49] S. Ramos, S. Gehrig, P. Pinggera, U. Franke, und C. Rother, „Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling“, in *2017 IEEE Intelligent Vehicles Symposium (IV)*, Juni 2017, S. 1025–1032. doi: 10.1109/IVS.2017.7995849.
[50] M. Thiel, J. Hinckeldeyn, und J. Kreutzfeldt, „Deep-Learning-Verfahren zur 3D-Objekterkennung in der Logistik“, *Logist. J. Proc.*, Bd. 2018, Nr. 01, Nov. 2018, doi: 10.2195/lj_Proc_thiel_de_201811_01.
[51] M. Hochstein, C. Kunert, J. Glöckle, M. Averweg, H. Weil, und K. Furmans, „Konsolidierassistent – Assistenzsystem für manuelle Konsolidier- und Sortierprozesse in Distributionszentren“, *Logist. J. Proc.*, Bd. 2017, Nr. 10, Okt. 2017, doi: 10.2195/lj_Proc_hochstein_de_201710_01.
[52] K. Schwäke, I. Dick, R. Bruns, und S. Ulrich, „Entwicklung eines flexiblen, vollautomatischen Kommissionierroboters“, *Logist. J. Proc.*, Bd. 2017, Nr. 10, Okt. 2017, doi: 10.2195/lj_Proc_schwaeke_de_201710_01.
[53] F. Martínez-Plumed *u. a.*, „CRISP-DM Twenty Years Later: From Data Mining Processes to Data Science Trajectories“, *IEEE Trans. Knowl. Data Eng.*, Bd. 33, Nr. 8, S. 3048–3061, Aug. 2021, doi: 10.1109/TKDE.2019.2962680.
[54] „ImageNet“. https://www.image-net.org/ (zugegriffen Juni 17, 2021).
[55] „Open Images V6“. https://storage.googleapis.com/openimages/web/index.html (zugegriffen Juni 17, 2021).
[56] Alexey, *AlexeyAB/darknet*. 2021. Zugegriffen: Apr. 10, 2021. [Online]. Verfügbar unter: https://github.com/AlexeyAB/darknet
[57] darrenl, *tzutalin/labelImg*. 2021. Zugegriffen: Mai 28, 2021. [Online]. Verfügbar unter: https://github.com/tzutalin/labelImg
[58] Cartucho, *Cartucho/OpenLabeling*. 2021. Zugegriffen: Mai 28, 2021. [Online]. Verfügbar unter: https://github.com/Cartucho/OpenLabeling
[59] *tensorflow/models*. tensorflow, 2021. Zugegriffen: Mai 29, 2021. [Online]. Verfügbar unter: https://github.com/tensorflow/models/blob/0c9253b4a0b34935cf78bd13e6520bbeee2f5f92/research/object_detection/g3doc/tf1_detection_zoo.md
[60] „Top 8 Deep Learning Frameworks You Should Know in 2021“, *Simplilearn.com*. https://www.simplilearn.com/tutorials/deep-learning-tutorial/deep-learning-frameworks (zugegriffen Mai 29, 2021).
[61] „Darknet: Open Source Neural Networks in C“. https://pjreddie.com/darknet/ (zugegriffen Mai 29, 2021).
[62] E. BUBER und B. DIRI, „Performance Analysis and CPU vs GPU Comparison for Deep Learning“, in *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, Okt. 2018, S. 1–6. doi: 10.1109/CEIT.2018.8751930.
[63] „Google Colaboratory“. https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index (zugegriffen Juni 04, 2021).
[64] T. A. Guy, *YOLOv3-Cloud-Tutorial*. 2021. Zugegriffen: Aug. 17, 2021. [Online]. Verfügbar unter: https://github.com/theAIGuysCode/YOLOv3-Cloud-Tutorial
[65] T. A. Guy, *YOLOv4-Cloud-Tutorial*. 2021. Zugegriffen: Aug. 17, 2021. [Online]. Verfügbar unter: https://github.com/theAIGuysCode/YOLOv4-Cloud-Tutorial

[66] J. Davis und M. Goadrich, „The relationship between Precision-Recall and ROC curves“, in *Proceedings of the 23rd international conference on Machine learning*, New York, NY, USA, Juni 2006, S. 233–240. doi: 10.1145/1143844.1143874.

[67] „GitHub - rafaelpadilla/Object-Detection-Metrics: Most popular metrics used to evaluate object detection algorithms.“, *GitHub*. https://github.com/rafaelpadilla/Object-Detection-Metrics (zugegriffen Juli 23, 2021).

[68] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, und A. Zisserman, „The Pascal Visual Object Classes (VOC) Challenge“, *Int. J. Comput. Vis.*, Bd. 88, Nr. 2, S. 303–338, Juni 2010, doi: 10.1007/s11263-009-0275-4.

[69] „Roboflow: Give your software the power to see objects in images and video“. https://roboflow.ai (zugegriffen Juni 23, 2021).

[70] „ImageNet Classification“. https://pjreddie.com/darknet/imagenet/ (zugegriffen Juli 03, 2021).

[71] S. Canu, „YOLO object detection using Opencv with Python“, *Pysource*, Juni 27, 2019. https://pysource.com/2019/06/27/yolo-object-detection-using-opencv-with-python/ (zugegriffen Okt. 25, 2021).

[72] „Kommissionierung - Prozessschritte“, *Logistik KNOWHOW*, März 01, 2017. https://logistikknowhow.com/lagerungstechniken/kommissionierung-prozessschritte/ (zugegriffen Juli 31, 2021).

Abbildungsverzeichnis

Formelverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AB	Anchor Boxes
AR	Augmented Reality
BB	Bounding Boxes
bzw.	beziehungsweise
CNN	Convolutional Neural Networks
Colab	Google Colaboratory
ConvLayer	Convolutional Layer
CPU	Central Processing Unit
CV	Computer Vision
DL	Deep Learning
etc.	et cetera
FN	False Negatives
FP	False Positives
FPN	Feature Pyramid Network
fps	Frames per Seconds
GPU	Graphical Processing Unit
GTB	Ground Truth Box
GUI	Graphical User Interface
IDE	Integrated Development Environment
IOU	Intersection Over Union
IR	Image Recognition
KI	Künstliche Intelligenz
KNN	Künstliche neuronale Netze
mAP	Mean Average Precision
ML	Machine Learning, Maschinelles Lernen
MSE	Mean Square Error
NMS	Non Maximum Suppression
NN	Neural Network, Neuronale Netze
OD	Object detection
PANet	Path Aggregation Network
PLM	Product Lifecycle Management
R-CNN	Region CNN
ROI	Region of Interest
SGD	Stochastic Gradient Descent
SSD	Single Shot Detector
SSE	Sum Squared Error
SVM	Suport Vector Machine
TN	True Negatives
TP	True Positives
VM	Virtual Machine
YOLO	You Only Look Once
z.B.	Zum Beispiel