

Machbarkeitsstudie - Smart Warehouse -

Echtzeit-Objektdetektoren im Vergleich

Studienarbeit

im Rahmen der Prüfung zum
Bachelor of Science (B.Sc.)

des Studienganges Angewandte Informatik
an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Felix Hausberger und Robin Kuck

Oktober 2019 - Juni 2020

-Sperrvermerk-

Abgabedatum: 01. Juni 2020
Bearbeitungszeitraum: 30.09.2019 - 01.06.2020
Matrikelnummer, Kurs: 2773463, 4409176, TINF17B2
Ausbildungsfirma: SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf, Deutschland
Gutachter an der DHBW: PD Dr.-Ing. Markus Reischl

Eidesstattliche Erklärung

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema:

Machbarkeitsstudie: Smart Warehouse

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 24. Mai 2020

Gez. Felix Hausberger und Robin Kuck

Hausberger, Felix und Kuck, Robin

Abstract

- *English* -

In this thesis the object detectors *You Only Look Once* and *Single Shot MultiBox Detector* are compared for precision, reactivity, training and inference behaviour and examined for their potential for industrial use. The background scenario of the *Smart Warehouse* offers live video data of a drone with goods in a warehouse, which are to be classified and localized in real time. In the future, this should make it possible to carry out inventories and inventory analyses of a warehouse in a time- and cost-efficient manner conserving resources.

The goal of this feasibility study is to find out whether the *Smart Warehouse* scenario is technically feasible. In addition, the focus is also on the object detectors themselves, their differences in architecture, behavior and how well they are generally suitable for industrial application scenarios.

Abstract

- *Deutsch* -

In dieser Arbeit werden die Objektdetektoren *You Only Look Once* und *Single Shot MultiBox Detector* nach Präzision, Reaktionsvermögen, Trainings- und Inferenzverhalten miteinander verglichen und auf deren Potential zum industriellen Einsatz untersucht. Das Hintergrundszenario des *Smart Warehouses* bietet dabei Live-Video Daten einer Drohne mit Warengegenständen in einem Warenhaus, die in Echtzeit klassifiziert und lokalisiert werden sollen. Dadurch sollen in Zukunft in der Industrie Inventuren und Bestandsanalysen eines Warenhauses zeit- und kostengünstig sowie ressourcenschonend ermöglicht werden können.

Diese Machbarkeitsstudie hat zum Ziel herauszufinden, ob das Szenario des *Smart Warehouse* technisch umsetzbar ist. Zusätzlich liegt der Fokus ebenso auf den Objektdetektoren selbst, deren Unterschiede hinsichtlich Architektur, Verhalten und wie gut sie allgemein für industrielle Anwendungsszenarien grundsätzlich geeignet sind.

Inhaltsverzeichnis

Abkürzungsverzeichnis	6
Abbildungsverzeichnis	8
Formelverzeichnis	10
Listenverzeichnis	11
0 Vorwort	12
1 Einführung	13
1.1 Forschungsumfeld	13
1.2 Problemstellung und Motivation	14
1.3 Vorgehensweise und Zielsetzung	14
2 Grundlagen und Forschungsstand	16
2.1 Deep Learning zur Bildverarbeitung	16
2.2 Neuronale Netze	18
2.3 Hyperparameter	19
2.4 Anforderungen an einen Datensatzes zur Erstellung eines Deep Learning Modells	25
2.5 Grundlagen zu Objektdetektoren	27
2.6 Objektdetektoren	30
2.7 Datensatzformate	38
2.8 Cloud Infrastruktur	39
2.9 Drohnen	44
3 Konzeption	45
3.1 Übersicht	45
3.2 Erstellen eines Trainingsdatensatzes	46
3.3 Einführen von Bewertungskriterien	49
3.4 Auswahl der Objektdetektoren	51
3.5 Auswahl der Trainingsinfrastruktur	53
3.6 Auswahl einer Drohne	56

3.7 Spezifikation der Inventursoftware	57
4 Realisierung	59
4.1 Umsetzung der Objektdetektoren	59
4.2 Dashboard Entwicklung	61
4.3 Zählalgorithmus	63
4.4 Drohnen Anbindung	63
5 Ergebnisse	69
5.1 Präzision und Inferenzverhalten	69
5.2 Reaktionsvermögen	77
5.3 Trainingsverhalten	77
6 Diskussion	79
6.1 Einsatz der Objektdetektoren in der Industrie	79
6.2 Machbarkeit des Smart Warehouse Szenarios	83
7 Zusammenfassung und Ausblick	86
Literaturverzeichnis	89
A Anhang	96

Abkürzungsverzeichnis

AI	Artificial Intelligence
ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
AWS	Amazon Web Services
CNN	Convolutional Neural Network
COCO	Common Objects in Context
CLI	Command Line Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
cuDNN	CUDA Deep Neural Network Library
DBN	Deep Belief Network
DevOps	Development Operations
DLL	Dynamic Link Library
EASA	European Aviation Safety Agency
ELU	Exponential Linear Unit
FCN	Fully Convolutional Network
FLOPS	Floating Point Operations Per Second
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GCP	Google Cloud Platform
GPU	Graphics Processing Unit
IoU	Intersection over Union
LReLU	Leaky Rectified Linear Unit

MLP	Multi-Layer Perceptron
mAP	mean Average Precision
PReLU	Parametric Rectified Linear Unit
PascalVOC	Pascal Visual Object Classes
PaaS	Platform-as-a-Service
R-CNN	Regional Convolutional Neural Network
ReLU	Rectified Linear Unit
REST	Representational State Transfer
RoI	Region of Interest
RPN	Region Proposal Network
SaaS	Software-as-a-Service
SDK	Software Development Kit
SSD	Single Shot MultiBox Detector
SVM	Support Vector Machine
TOPS	Tera Operations Per Second
TPU	Tensor Processing Unit
UDP	User Datagram Protocol
XML	Extensible Markup Language
YOLO	You Only Look Once

Abbildungsverzeichnis

2.1	Anwendungsgebiete von Deep Learning zur Bildverarbeitung im Überblick	17
2.2	Gradientenverfahren	22
2.3	ReLU-Aktivierungsfunktionen	23
2.4	ELU-Aktivierungsfunktion	24
2.5	Intersection over Union	28
2.6	Berechnung mAP	29
2.7	R-CNN Architektur	30
2.8	Faster R-CNN Architektur	31
2.9	SSD Bounding Box Vorschläge	33
2.10	Bounding Boxes	34
2.11	SSD Architektur	34
2.12	Vereinfachte Darstellung des YOLO Algorithmus	36
2.13	YOLO Architektur	37
3.1	Konzeptionelle Schritte	45
3.2	Die neun Datensatz-Kategorien	46
3.3	Annotieren der Bilddaten mit labelImg	47
3.4	Ausschnitt I aus dem Smart Warehouse Datensatz	48
3.5	Ausschnitt II aus dem Smart Warehouse Datensatz	49
3.6	Vergleich SSD auf PascalVOC 2007	51
3.7	Architektur des Smart Warehouse Szenarios	57
3.8	Smart Warehouse User Interface Prototyp	58
4.1	Webapplikation Smart Warehouse	62
4.2	Drohnenflugsequenz	65
4.3	Ablaufsequenz für die Verarbeitung von Drohnenbefehlen und Inferenz von Bildern	66
4.4	Bild der Video-Kamera der Ryze Tello EDU Drohne	67
5.1	Entwicklung der SSD Trainings- und Testverlustkurve im Training	69
5.2	Entwicklung der YOLO Testverlustkurve und der <i>mAP</i> im Training	70
5.3	Detektionsverhalten von SSD bei extremen Blicklagen	72
5.4	Detektionsverhalten von YOLO bei extremen Blicklagen	73
5.5	Detektionsverhalten von SSD bei unterschiedlichen Entfernungen	74

5.6	Detektionsverhalten von YOLO bei unterschiedlichen Entfernungen	75
5.7	Detektionsverhalten von SSD bei unterschiedlichen Beleuchtungsverhältnissen	75
5.8	Detektionsverhalten von YOLO bei unterschiedlichen Beleuchtungsverhältnissen	76
6.1	Bildauszug aus Video-Stream nach der Inferenz mit YOLO	85
A.1	Linear Threshold Unit	96
A.2	Das einschichtige Perzeptron	98
A.3	Convolutional Layer	102
A.4	Zero-Padding	102
A.5	Veranschaulichung von Feature Maps	103
A.6	Pooling Layer	104

Formelverzeichnis

2.1	Die Softmax-Funktion	18
2.2	Die Smooth L1 Funktion	19
2.3	Superpositionsprinzip anhand der Varianz	20
2.4	Standardverteilung nach Xavier Initialisierung	20
2.5	Momentum Optimierung	21
2.6	Precision und Recall	28
A.1	Die Heaviside-Funktion	97
A.2	Die RMSE-Funktion	99
A.3	Neuberechnung der Gewichtungsmatrix durch partielle Differentiation . .	100
A.4	Die Sigmoid Funktion	100
A.5	Mathematische Faltung	102

Listenverzeichnis

2.1	PascalVOC Bildannotation	38
2.2	Konfigurationsdatei zum Trainingsjob auf FloydHub	42
4.1	Streaming der inferierten Bilddaten	61
4.2	Zählalgorithmus zum Zählen der detektierten Objekte	63
4.3	Initialisierung der Socket Klasse	64

0. Vorwort

Besonderen Dank ist an unseren Betreuer PD Dr. -Ing. Markus Reischl auszusprechen, ohne den die folgenden Forschungsergebnisse nicht zustande gekommen wären. Auch dem Informatik Labor unter Enrico Hühneborg der DHBW ist für die nötige finanzielle Unterstützung zum Erwerb der Drohne zu danken.

1. Einführung

1.1. Forschungsumfeld

Einen Teilbereich des maschinellen Lernens (engl.: machine learning) stellt das *Deep Learning* dar, welches auf künstlichen neuronalen Netzen (engl.: artificial neural networks) (ANNs) basiert. Unter einer Vielzahl von Typen von ANNs wie Autoencodern, Deep Boltzmann Machines oder rekurrenten neuronale Netzen befindet sich ebenso die Klasse der *Convolutional Neural Networks* (CNNs), welche hauptsächlich zur Lösung von Klassifikationsproblemen in der Audio-, Text- und Bildverarbeitung genutzt werden [1, 2].

Ein Forschungsfeld im *Deep Learning* stellen Objektdetektoren dar, welche basierend auf CNNs neben Bildklassifikationsproblemen ebenso in der Lage sind, Lokalisationsprobleme zu lösen. Solchen Objektdetektoren werden in der heutigen Zeit immer mehr Bedeutung zugesprochen angesichts neuer Herausforderungen wie autonomen Fahren, automatisierter industrieller Verarbeitung oder aber auch staatlicher Überwachung. Verschiedene Ansätze werden zur Realisierung von Objektdetektoren verwendet, unter anderem Netzarchitekturen wie *Regional Convolutional Neural Networks* (R-CNNs), *You Only Look Once* (YOLO) oder der *Single Shot MultiBox Detector* (SSD).

Gerade in Zeiten des industriellen Wandels in Richtung Industrie 4.0 können solche Objektdetektoren ein großes Optimierungspotential für bestehende Industrieszenarien bieten, beispielsweise in der Lagerhaltung und Logistik. Kombiniert mit einer autonomen Drohne können Objektdetektoren es ermöglichen, ohne menschliche Hilfe Inventuren und Bestandsprüfungen in einem Lager- oder Warenhaus durchzuführen. Start-up Unternehmen wie *doks. innovation* werben bereits mit ähnlichen Lösungen, die 80% Zeiteinsparung und 90% Kostensenkung versprechen [3]. Lösungen wie *inventAIRyX* beschränken sich allerdings speziell auf Lagerhäuser, in denen die verpackten Waren mittels Sensoren identifiziert werden, was Großhändler mit Warenhäusern wie *Baumarkt* oder *Selgros* ausschließt. Statt Waren mittels RFID Chips oder Barcodes zu identifizieren, soll in dieser Arbeit der Einsatz von Objektdetektoren für dieses Szenario evaluiert werden.

Wie sich die unterschiedlichen Objektdetektoren unter Echtzeitvoraussetzungen im Betrieb verhalten, soll anhand des Industriebeispiels *Smart Warehouse* innerhalb dieser Arbeit untersucht werden.

1.2. Problemstellung und Motivation

Das *Smart Warehouse* beschreibt ein Warenhaus, welches unter Einsatz einer Drohne in der Lage sei soll, Inventuren und Bestandsprüfungen weitgehend ohne menschliche Hilfe durchzuführen. Das Live-Bild der Drohne soll von den Objektdetektoren dazu genutzt werden, Warengegenstände zu lokalisieren und zu klassifizieren.

Neben der Frage, ob ein solches Industrieszenario überhaupt umsetzbar ist, sollen die Objektdetektoren in diesem Anwendungsszenario nach verschiedenen Kriterien miteinander verglichen und beurteilt werden. Diese Kriterien lassen sich hauptsächlich in die Kategorien Präzision, Reaktionsvermögen, Trainings- und Inferenzverhalten untergliedern und werden genauer eingeführt. Dadurch lassen sich Aussagen darüber treffen, wie gut nach dem momentanen Forschungsstand um Objektdetektoren solche das Potential bieten, industriell eingesetzt zu werden.

Falls die Machbarkeitsstudie des *Smart Warehouse* glückt, so kann der Industrie ein kostengünstiges, zeitsparendes und ressourcenschonendes Modell zur Inventurverwaltung eines Warenhauses angeboten werden.

1.3. Vorgehensweise und Zielsetzung

Im Grundlagenkapitel 2 muss sich mit den theoretischen Grundlagen von CNNs und Objektdetektoren auseinander gesetzt werden. Hierzu ist zunächst eine Einführung in *Deep Learning* zur Bildverarbeitung und neuronale Netz erforderlich als auch zu Hyperparametern zum Trainieren eines neuronalen Netzes (siehe Kapitel 2.1, 2.2 und 2.3).

Um weitere Grundlagen zum Training von neuronalen Netzen einzuführen, wird anschließend über die Anforderungen eines Datensatzes gesprochen (siehe Kapitel 2.4), bevor weitere Grundlagen zu Objektdetektoren eingeführt werden (siehe Kapitel 2.5).

Nachdem zu Beginn des Kapitels 2.5 kurz auf den Grundbaustein moderner Objektdetektoren eingegangen wird, den CNNs, können anschließend die Funktionsweisen und Architekturen der drei miteinander verglichenen Objektdetektoren der *R-CNN* Familie, *YOLO* und des *SSDs* erläutert werden (siehe Kapitel 2.6). Bei *R-CNN* und *YOLO* ist zu bemerken, dass unterschiedliche Evolutionsstufen der Detektoren zu betrachten sind.

Um weitere Grundlagen zum Training von neuronalen Netzen einzuführen, wird anschließend über zwei wesentlichen Speicherformate eines Datensatzes gesprochen (siehe Kapitel 2.7), bevor verschiedene Cloud Anbieter für das Trainieren von *Deep Learning* Modellen aufgezeigt werden (siehe Kapitel 2.8).

Zuletzt wird ein kurzer Überblick über ausgewählte Drohnenangebote gegeben und dabei auch ein Einblick in die gesetzlichen Rahmenbedingungen zu Drohnen gegeben (siehe Kapitel 2.9)

In Kapitel 3 werden chronologisch Teilziele der Konzeption beschrieben, darunter das Erstellen eines Trainingsdatensatzes (siehe Kapitel 3.2), dem Einführen von Bewertungskriterien (siehe Kapitel 3.3), der Auswahl von Objektdetektoren, der Trainingsinfrastruktur und der Drohne (siehe Kapitel 3.4, 3.5, 3.6) und die Spezifikation der Inventursoftware (siehe Kapitel 3.7).

In der Realisierung werden die Herausforderungen zur Steuerung und Anbindung der Drohne betrachtet und zudem die Objektdetektoren auf die realen Datensätze trainiert. Auch die Entwicklung der Webapplikation zur Visualisierung des Live-Bildes und der erkannten Objekte sowie der darin realisierte Zählalgorithmus zur Durchführung der Inventur wird Bestandteil dieses Kapitels sein. Die Ergebnisse der Realisierungsphase werden nach den in Unterkapitel 3.3 definierten Bewertungskriterien im folgenden Kapitel dargestellt.

Ziel der Arbeit ist es Aussagen über die Fähigkeit von Objektdetektoren zum Einsatz in der Industrie zu treffen, indem eine Bewertung der Verhaltensweisen der Objektdetektoren nach den eingeführten Bewertungskriterien durchgeführt wird. Als Umgebung und Rahmenszenario für die Evaluation dient das *Smart Warehouse* Szenario, dessen Machbarkeit ebenfalls herausgestellt werden soll (siehe Kapitel 5).

Zuletzt wird das Wesen der Arbeit nochmals kurz zusammengefasst und anschließend auf mögliche Verbesserungen und Ausblicke in die Zukunft aufmerksam gemacht.

2. Grundlagen und Forschungsstand

Neben einer Einführung in den Anwendungsbereich von *Deep Learning* zur Bildverarbeitung soll sich das folgende Kapitel speziell mit Architekturen unterschiedlicher Objektdetektoren auseinander setzen und herausstellen, wie sich diese voneinander abgrenzen. Davor wird allerdings zunächst grundlegendes Wissen über neuronale Netze und wie diese „lernen“ vermittelt sowie über Hyperparameter und wie ein eigener Datensatz zu gestalten ist.

2.1. Deep Learning zur Bildverarbeitung

Ein klassisches Anwendungsgebiet von *Deep Learning* zu Bildverarbeitung oder auch allgemein von maschinellem Lernen beschreibt die *Klassifikation*. Hierbei werden bestimmte Kategorien, auch *Klassen* genannt, definiert, in die ein Bild eingeordnet werden soll. Die *Klassifikation* wird anhand von aus dem Bild extrahierten Merkmalen, auch *Features* genannt, getroffen. Die Merkmale werden zu einem *Merkmalsvektor* oder auch *Feature Map* zusammengefasst und von einem *neuronalen Netz* verarbeitet. Das Ergebnis der Verarbeitung durch das neuronale Netz ist die Einordnung in eine bestimme Klasse.

Zusätzlich zur *Klassifikation* eines Bildes kann das auf dem Bild abgebildete Objekt ebenfalls lokalisiert werden. Es wird dann von sogenannter *Objektdetektion* gesprochen. Es können auch mehrere Objekte auf einem Bild detektiert werden. Ergebnis der Objektdetektion ist somit nicht nur eine Klasseneinordnung sondern ebenso eine klare Positionsangabe des Objektes auf dem Bild. Die Positionsangabe erfolgt durch Angabe einer sogenannten *Bounding Box*. Diese umrahmt das jeweils detektierte Objekt und wird durch ihren linken oberen Eckpunkt sowie ihre Höhe und Breite beschrieben.

Neben der klassischen *Klassifikation* und der *Objektdetektion* existiert ebenso ein drittes Anwendungsgebiet von *Deep Learning* zu Bildverarbeitung, die *Segmentierung*. Bei der *semantischen Segmentierung* wird versucht, jede einzelne Pixel eines Bildes einer Klasse zuzuordnen und dementsprechend farblich im Bild zu hinterlegen. *Instanzbasierte Segmentierung* hingegen zielt darauf ab, nicht nur jeden Pixel zu einer Klasse zuzuordnen,

sondern ebenso eine Identität zu einem Objekt zuzuweisen. Es setzt sich zusammen aus *semantischer Segmentierung* und paralleler *Objektdetektion* [4, 5, 6].

Einen Überblick über die vorgestellten Anwendungsgebiete ist in Abbildung 2.1 zu sehen.

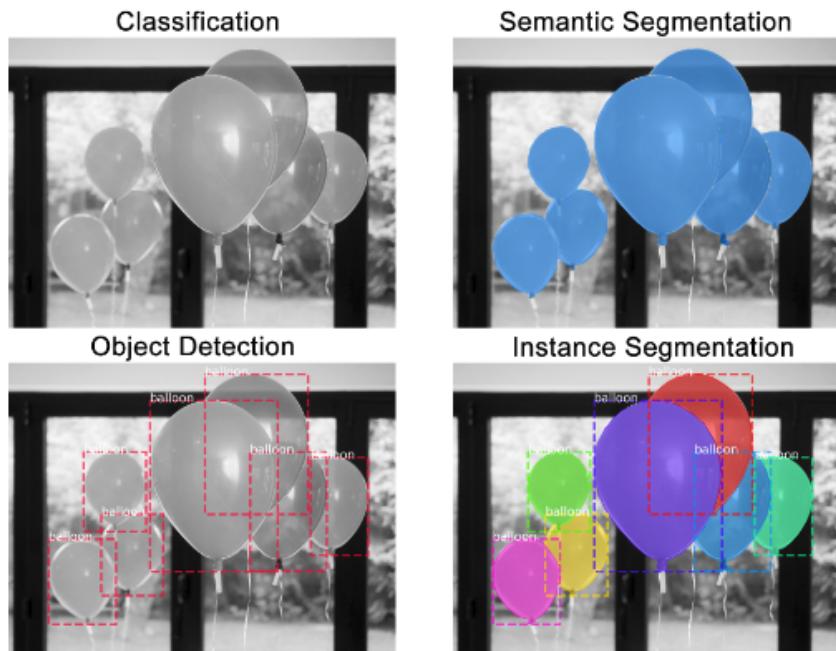


Abbildung 2.1.: Anwendungsgebiete von Deep Learning zur Bildverarbeitung im Überblick [7]

Für eine einfache *Klassifikation* eines Bildes können einfache sogenannte *Feed-Forward* Netze verwendet werden. Es kann hierbei aber auch auf konventionelle Methoden der Bildverarbeitung zurück gegriffen werden. Für die *Objektdetektion* stehen Architekturen wie *You Only Look Once* (YOLO), der *Single Shot MultiBox Detector* (SSD) oder neuronale Netze der *Regional Convolutional Neural Networks* (R-CNN) bereit. Aus dieser Familie entstammt ebenso das *Mask R-CNN* Netz, das zur *instanzbasierten Segmentierung* von Objekten verwendet wird.

2.2. Neuronale Netze

Ein neuronales Netz bildet die Grundlage des *Deep Learnings*. Neuronale Netze sind komplexe Datenstrukturen, deren kleinste Einheiten aus sogenannten *Linear Threshold Units* (LTUs) bestehen. Diese geben auf Basis von mehreren Eingangswerten einen durch eine Aktivierungsfunktion beschriebenen Ausgangswert aus. Mehrere LTUs sind zusammen in einer eindimensionalen Schicht angeordnet. Ein oder mehrere solcher Schichten bilden ein Perzeptron, den Grundbaustein eines ANNs. Dabei ist jede LTU einer Schicht mit allen LTUs einer folgenden Schicht verbunden. Hier wird auch von sogenannten vollständig verbundenen Schichten (engl.: *Fully-Connected Layer*) gesprochen. Die Verbindungen sind mit einer Gewichtung versehen, die im Lernprozess angepasst werden und somit für bestimmte Eingangsdaten nur bestimmte LTUs aktivieren. Diejenige LTU die am Ende des Netzes alleinig aktiviert ist gibt die Klassifizierung der Eingangsdaten an. Da allerdings meist nicht nur eine LTU alleinig aktiviert ist, findet die Klassifikation auf Basis von Wahrscheinlichkeiten statt. Hierzu wird meist die *Softmax-Funktion*

$$h_w(x) = \sigma(z)_j = \frac{e^{z_j}}{\sum_{i=0}^n e^{z_i}} \quad (2.1)$$

verwendet, die den Wert des j -ten LTUs einer Schicht mit allen anderen n Werten der LTUs derselben Schicht ins Verhältnis setzt. Falls das neuronale Netz von der Eingangsschicht zur Ausgangsschicht unidirektional von den Eingangsdaten durchlaufen wird, ohne zu bereits besuchten Schichten zurückzukehren, nennt man das neuronale Netz auch *Feed-Forward Network*.

Im Wesentlichen existieren vier Methoden, mit deren Hilfe neuronale Netze trainiert werden können: *Überwachten Lernverfahren*, *Halbüberwachte Lernverfahren*, *unüberwachte Lernverfahren* und das *Reinforcement Learning*. Für die Aufgabe der Objektdetektion sind allerdings wesentlich *überwachten Lernverfahren* von Relevanz, die den Trainingsdaten Lösungen, sogenannte *Labels* oder *Annotations*, hinzufügen. Der Lernprozess wird hierbei über das Gradientenverfahren, basierend auf einer Kostenfunktion, und dem Backpropagation Algorithmus ermöglicht. Die Kostenfunktion ist ein Qualitätsmaß dafür, wie weit die Ausgabe einer LTU vom erwarteten Wert abweicht [1]. Eine bekannte

Kostenfunktion für das *überwachte Lernen* ist beispielsweise die *Smooth L1* Funktion

$$SM_{L1}(\mathbf{z}, \mathbf{o}) = \begin{cases} \sum_{k=0}^n 0.5(z_k - o_k)^2 & \text{wenn } |x| < 1 \\ \sum_{k=0}^n |z_k - o_k| - 0.5 & \text{sonst} \end{cases} \quad (2.2)$$

, bei der \mathbf{z} der erwartete Ausgabevektor des Perzeptrons ist, während \mathbf{o} die momentane Ausgabe darstellt. Die Kostenfunktion ist nur einer der vielen Parameter, die für das Trainingsverfahren eines neuronalen Netzes festgelegt werden kann. Weitere sogenannte *Hyperparameter* folgen in nachfolgendem Kapitel. Außerdem sind weitere Informationen über das Perzepron, über Lernmethoden, das Gradientenverfahren und den Backpropagation Algorithmus im Anhang hinterlegt.

2.3. Hyperparameter

Hyperparameter sind die Parameter, die zur anfänglichen Konfiguration des neuronalen Netzes als auch zur Konfiguration des Lernprozesses herangezogen werden. Um im Laufe der Arbeit verstehen zu können, wie die Objektdetektoren auf Seiten der Netzarchitektur und des Lernverhaltens optimiert wurden, ist demnach ein kurzer Einblick in den Themenbereich der Hyperparameter von Nöten.

Anzahl der LTUs

Die Anzahl der LTUs im ANN ist dafür ausschlaggebend, wie hoch der Komplexitätsanspruch eines Klassifizierungsproblems sein darf, um noch vom ANN gelöst werden zu können. Die Anzahl der LTUs hängt hauptsächlich von den Eingangsdaten ab. Über die optimalste Anzahl an LTUs pro Schicht lässt sich allerdings nur schwer etwas vorhersagen. Generell gilt, dass bei gleicher Anzahl an LTUs tiefere Netze eine weitaus höheren Parametereffizienz aufweisen als breitere Netze, da diese schneller gegen den gewünschten Zustand konvergieren. Zudem lassen sie sich somit schneller und kostengünstiger trainieren. So müssten bei einem 2x32 Netz 1024 Gewichtungen angepasst werden, während es bei einem 32x2 Netz dies nur 128 sind [1].

Initialisierung der Gewichtungen

Auch stellt die Initialisierung der Gewichte eines ANNs zu Beginn des Trainingsprozesses eine berechtigte Frage dar. Falls keine bereits trainierten ANNs für ein Klassifikationsproblem vorliegen, so werden die Gewichtungen meist zufällig nach einer Normalverteilung gewählt [1].

Dies hat allerdings zur Folge, dass nach der Berechnung der gewichteten Summen aller LTUs die Gewichtungswerte der folgenden Schicht nicht mehr normalverteilt sind, da für die Varianz zweier unkorrelierter Zufallsvariablen das Superpositionsprinzip

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) \quad (2.3)$$

gilt.

Durch die größer werdende Standardabweichung können demnach Gewichtungswerte entstehen, die weit vom Mittelwert Null abweichen. Dies kann wiederum dazu führen, dass der Gradientenabstieg während des Backpropagation-Verfahrens nur langsam vollzogen werden kann, da der Gradient bei bestimmten Aktivierungsfunktionen wie der *Sigmoid-Funktion* an solchen Stellen gegen Null konvergiert [1].

Eine *Xavier Initialisierung* umgeht das Problem der sogenannten *schwindenden Gradi- enten*, indem die Gewichte nach

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \quad (2.4)$$

gleichverteilt werden, wobei n_j die Anzahl an LTUs der j -ten Schicht sind [8].

Anzahl an Epochen

Die Anzahl der Epochen beschreibt die Durchläufe durch einen bestimmten Trainingsdatensatz während der Trainingsphase. Ist die Anzahl zu hoch gewählt, wird Gefahr gelaufen, sogenanntes *Overfitting* des ANNs zu erreichen. Dies bedeutet ein fehlendes

Abstraktionsvermögen des ANNs durch „Auswendiglernen“ der Trainingsdaten und damit eine alleinige richtige Erkennung der Trainingsdatensätze.

Lernrate

Die Lernrate η (A.3) gibt an, wie groß die Sprünge im Gradientenverfahren sein sollen und damit indirekt wie viele Iterationen benötigt werden, um das Minimum der Kostenfunktion zu erreichen. Ziel der Anpassung einer Lernrate ist es, mit möglichst wenig Iterationen und Testdaten die optimale Konstellation des neuronalen Netzes zu berechnen. Deshalb wird sie standardmäßig zu Beginn der Iterationen groß gewählt, um sich dem Minimum schnell zu nähern, während sie am Ende immer kleiner gewählt wird, um nicht über das globale Minimum hinaus zu gehen. Dieses Vorgehen wird als *Simulated Annealing* bezeichnet, während das Funktion zum Festlegen der Lernrate als *Learning Schedule* betitelt wird [1].

Die Anzahl der Durchläufe wird zu Beginn des Verfahrens zunächst hoch angesetzt, das Verfahren wird aber genau dann gestoppt, sobald der Gradientenvektor unter eine gewisse Abbruchgrenze fällt. Zwar ist das globale Minimum zu diesem Zeitpunkt noch nicht erreicht, allerdings kann es auch nie vollkommen erreicht werden, da die für das Gradientenverfahren genutzten Aktivierungsfunktionen nie einen partiellen Ableitungswert gleich Null zulassen [1]. In diesem Sinne wird auch von *Toleranz* gesprochen.

Moment

Das Gradientenverfahren kann beschleunigt werden, indem während des Gradientenabstiegs frühere Gradienten Einfluss auf den nächsten Gradientenschritt nehmen. Es wird ein „Momentum“ aufgebaut. Damit das Momentum

$$m_x = \beta \cdot m_{x-1} + \eta \frac{\partial E}{\partial w_{ij}} \quad (2.5)$$

$$w_{ijt} = w_{ijt-1} - m$$

allerdings nicht zu groß wird, beschränkt der Hyperparameter $\beta \in [0, 1]$ die Größe des Momentum [1].

Die Momentum Optimierung kann dazu benutzt werden, das *stochastische Gradientenverfahren* bzw. *Mini-Batch* Verfahren zu beschleunigen und lokale Minima besser zu überwinden.

Auswahl des Gradientenverfahrens

Generell wird zwischen drei verschiedenen Arten unterschieden, das Gradientenverfahren durchzuführen (siehe Abbildung 2.2):

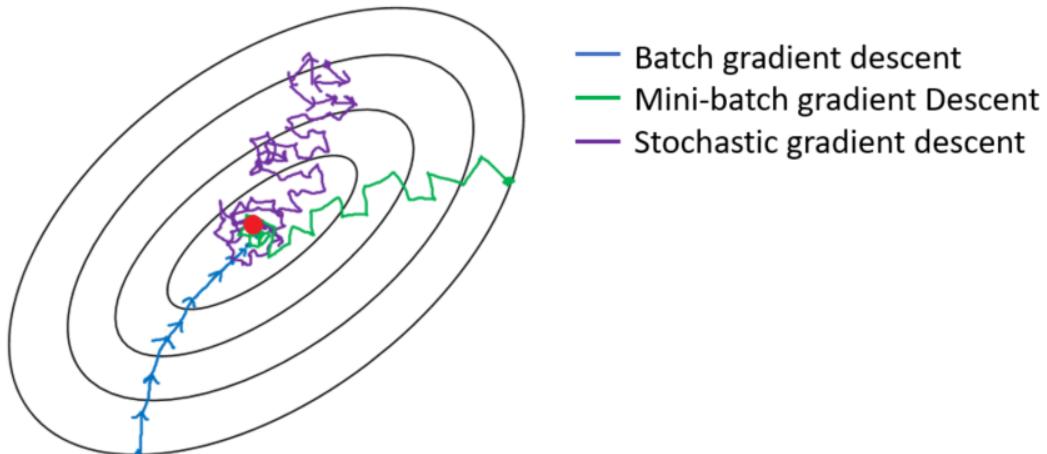


Abbildung 2.2.: Gradientenverfahren [9]

Beim *Batch* Verfahren werden in einem Trainingsdurchlauf, der *Epoche*, alle vorhandenen Daten des Trainingsdatensatzes herangezogen, um einen Gradientenabstieg zu vollziehen. Dies ist bei großen Trainingsdatensätzen auffällig langsam, dafür aber hinsichtlich der Erreichung des lokalen Minimums sehr zielstrebig [1].

Das *stochastische Gradientenverfahren* führt nach jedem einzelnen Dateneintrag im Trainingsdatensatz einen Gradientenabstieg durch. Da nur wenige Daten des ANNs verändert werden müssen, ist dieses Verfahren deutlich schneller, dafür aber unregelmäßiger

hinsichtlich der Erreichung des Minimums. Oft wird das stochastische Gradientenverfahren verwendet, wenn nicht der komplette Trainingsdatensatz in den Hauptspeicher oder Grafikspeicher geladen werden kann. Diese Fähigkeit wird oft als *Out-of-Core* Fähigkeit bezeichnet. Es hat auch den Vorteil, besser das globale Minimum der Kostenfunktion aufzufinden, da bei lokalen Minima die Chance besteht, durch den unregelmäßigen Gradientenabstieg das lokale Minimum wieder zu überwinden [1].

Ein Kompromiss der beiden Verfahren bietet das *Mini-Batch* Verfahren, bei dem wiederholt Teilmengen des gesamten Datensatzes für einen Gradientenabstieg verwendet werden. Die Anzahl an Elementen einer solchen Teilmenge wird als *Batchgröße* bezeichnet. Genauso wie das *Batch* Verfahren bietet das *Mini-Batch* Verfahren den Vorteil, die partiellen Ableitungen als Matrizenoperationen auf die Grafikkarten auszulagern, um die Performanz durch Parallelisierung zu steigern. Zusätzlich muss weniger Speicher für einen Gradientenabstieg reserviert werden [1].

Aktivierungsfunktionen

Zwei bekannte und ähnliche Aktivierungsfunktionen sind die *Sigmoid-Funktion* und die *Tangens Hyperbolicus* Funktion. Da diese allerdings durch ihr schnelles Konvergieren gegen den Grenzwert anfällig für das Problem *schwindender Gradienten* sind [1], wird die *Rectified Linear Unit* (ReLU) bzw. *Parametric/Leaky Rectified Linear Unit* (PReLU/LReLU) Aktivierungsfunktion bevorzugt (siehe Abbildung 2.3).

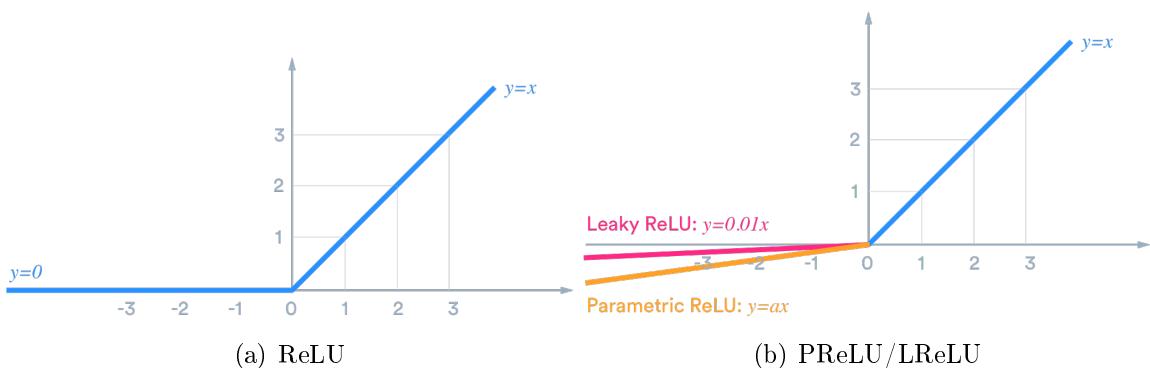


Abbildung 2.3.: ReLU-Aktivierungsfunktionen [10]

Bei *ReLU* kann es während des Trainingsprozesses dazu kommen, dass LTUs nach dem Gradientenabstieg einen negativen Wert aufweisen, weshalb sie nicht weiter aktiviert werden und für den Rest der Trainingsdauer „tot“ sind. Um dies zu verhindern, wurde *LReLU* dazu genutzt, um eine Reaktivierung zu ermöglichen, da auch für negative LTU Werte ein Gradient der Aktivierungsfunktion bestimmt werden kann. Bei *LReLU* ist die Steigung der Funktion im zweiten Quadranten statisch gewählt, während sie bei *PReLU* dynamisch von neuronalen Netz während des Trainingsprozesses selbst gelernt werden kann [1].

Eine letzte Variante der Aktivierungsfunktionen beschreibt die *ELU* Funktion (siehe Abbildung 2.4.).

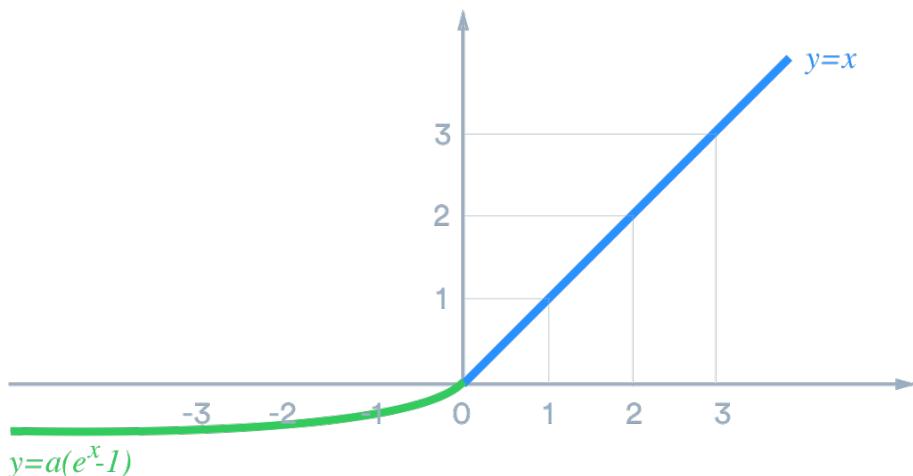


Abbildung 2.4.: ELU-Aktivierungsfunktion [10]

Sie besitzt nicht nur die Eigenschaft schwindende Gradienten und damit nicht anpassbare, sogenannte „tote“ LTUs zu verhindern, sondern ist im gesamten Definitionsbereich ebenso eine stetig differenzierbare Funktion, was das Gradientenverfahren beschleunigt. Als Standardwert für den Streckungsfaktor α der niederen Funktion wird oft Eins verwendet. Nachteil der *ELU* Funktion ist der erhöhte Rechenaufwand, was aber durch die schnellere Konvergenz kompensiert wird [1].

2.4. Anforderungen an einen Datensatzes zur Erstellung eines Deep Learning Modells

Datensatzzusammensetzung

Zum Erstellen und Auswählen eines *Deep Learning* Modells wird der Datenbestand in der Regel in drei Kategorien unterteilt. Ein Datensatz wird für das Training des Modells verwendet. Durch das anschließende Anwenden des Modells auf zuvor ungesehene Daten, den Testdaten, wird der *Verallgemeinerungsfehler* gemessen, der möglichst niedrig ausfallen sollte. Fällt der allgemeine Fehler während des Trainings niedrig aus, der *Verallgemeinerungsfehler* während des Testdurchlaufs allerdings hoch, so liegt klassisches *Overfitting* vor, die Trainingsdaten wurden auswendig gelernt [1].

Anschließend werden in mehreren Durchläufen die Hyperparameter des Trainingsprozesses angepasst, sodass letztendlich der *Verallgemeinerungsfehler* für die Testdaten niedrig ausfällt. Kommt es anschließend zum Einsatz des Modells in der Produktivumgebung, so können trotz allem unerwartete Ergebnisse bezüglich des Abstraktionsvermögens des Modells auftreten, was daran liegt, dass das Modell allein auf die Testdaten hin optimiert wurde. Um dies zu vermeiden, wird ein dritter Datensatz, der Validierungsdatensatz, eingeführt. Mehrere Modelle werden dabei durch den Validierungsdatensatz getestet und das am besten abschneidende Modell mit dessen Hyperparametern ausgewählt. Der eigentliche Testdatendatatz wird anschließend nur noch zur Abschätzung des *Verallgemeinerungsfehlers* verwendet [1].

Oft wird der Trainingsdatensatz mit dem Validierungsdatensatz zum sogenannten *Trainval* Datensatz zusammengeführt. Dies steht im Kontext des sogenannten *K-Kreuzvalidierungsverfahrens*. Dabei wird der *Trainval* Datensatz in K gleich große, komplementäre Untermengen unterteilt. Eine dieser Untermengen dient anschließend als Validierungsdatensatz. Für jedes zu trainierende Modell mit unterschiedlichen Hyperparametern wird eine andere Untermenge als Validierungsdatensatz ausgewählt. Hierdurch steigt die Aussagekraft des Abstraktionsvermögens nach der Validierung und zudem müssen keine Trainingsdaten dauerhaft für die Validierung zurück gelegt werden. In der Regel werden 80% der Gesamtdaten als *Trainval* Datensatz verwendet [1].

Qualität und Quantität der Daten

Um ein funktionsfähiges Modell zu trainieren, muss der Datensatz einem gewissen Standard nachkommen. Demnach müssen die zu klassifizierenden Objekte vollständig im Bild enthalten und gut erkennbar sein. Zwar gibt es gerade im *PascalVOC* Datensatzformat¹ ebenso die Möglichkeit, Objekte als „schwierig erkennbar“ zu markieren, dennoch sollen solche Objekte nicht die Mehrheit im gesamten Datensatz ausmachen. Auch die Aufnahme von Objekten in unterschiedlichen Umgebungen, Verdeckungsgraden, Entfernung und Blicklagen fördert langfristig das Abstraktionsvermögen des Modells.

Ebenso muss ein ausreichend großer Datensatz vorliegen, um das gewünschte Abstraktionsvermögen des Modells zu erreichen. Die Ergebnisse aus Abbildung 3.6 wurden beispielsweise durch Kombination der *Trainval* Datensätze von *PascalVOC* 2007 und 2012 erzielt und umfasst 16.551 Bilder im Trainingsverfahren [11, 12, 13].

Techniken zum Trainieren bei geringen Datenmengen

Bei Betrachtung der obigen Ergebnisse wird schnell deutlich, dass für ein komplexeres *Deep Learning* Modell ein umfangreicher Datensatz von Nöten ist. Allerdings gibt es drei bekannte Techniken, wie auch mit kleineren Datenbeständen ein sehenswertes Ergebnis erzielt werden kann. Eines davon ist das *K-Kreuzvalidierungsverfahrens*, das bereits betrachtet wurde.

Daneben können beim sogenannten *Transfer Learning* von einem bereits für ein ähnliches Problem trainiertes Modell die ersten Schichten des neuronalen Netzes für das neue Modell wiederverwendet werden. Die übernommenen Gewichtungen werden nicht mit trainiert. Neben einer kleineren Datenmenge zum Trainieren hat das *Transfer Learning* ebenso den Vorteil das Training selbst zu beschleunigen [1].

Eine weitere Technik beschreibt das künstliche Vergrößern des Datensatzes durch affine Transformationen wie Translation, Rotation oder Skalierung und wird *Data Augmentation* genannt [1].

¹PASCAL Visual Object Classes - Gängiges, standardisiertes Datensatzformat für Objektdetektion

2.5. Grundlagen zu Objektdetektoren

Ein Anwendungsgebiet des *Deep Learnings* beschreiben die Objektdetektoren, die im *Smart Warehouse* Szenario zur Lokalisierung und Klassifizierung von Bestandsobjekten genutzt werden. Im folgenden Kapitel soll demnach der Grundbaustein von Objektdetektoren, das *Convolutional Neural Network*, zunächst genauer betrachtet werden, bevor auf die gängigsten Objektdetektoren, die der *Regional Convolutional Neural Networks* (R-CNN), der *Single Shot MultiBox Detector* und der *You Only Look Once* Ansatz im darauf folgenden Kapitel eingegangen wird. Auch wird die gängiste Metrik zum Vergleich von Objektdetektoren, die *mean Average Precision* eingeführt.

Convolutional Neural Networks

CNNs sind aus drei wesentlichen Bestandteilen aufgebaut. *Convolutional Layer* erzeugen mehrere, übereinander gelegte *Feature Maps*, die durch Anwendung mehrerer Filter, auch *Kernel* genannt, auf ein Bild durch die mathematischen Faltungsoperation entstehen. Daher stammt der „faltungsbedingten“ (engl.: convolutional) Charakter der Schicht. Die Filter werden auf kleinere Bereiche eines Bildes angewandt und sollen dabei Muster im Bild erkennen. Die Filter werden durch Gewichtungen implementiert, die im CNN trainiert werden müssen, was das „Lernen“ im CNN darstellt. Ziel ist es also die optimalen Filter zu finden, die ein Objekt auf Bildern erkennen sollen. Selbstverständlich können auch weitere Filter trainiert werden, um mehrere Objekte zu erkennen. Dadurch, dass die Filter nur auf Teilbereiche des Bildes angewendet werden, ist somit nicht jede LTU mit allen LTUs der vorherigen *Feature Map* verbunden, man spricht hier auch von einem „lokalen Wahrnehmungsfeld“. Im Verlauf des CNNs werden kleinere Muster zu größeren Mustern abstrahiert und dabei das CNN durch sogenannte *Pooling Layer* verkleinert. Ein einfaches *Feed-Forward ANN* mit *Fully-Connected Layer*n am Ende des CNNs trifft anschließend die Klassifikationsaussage des Bildes, also welches Objekt auf dem Bild gezeigt wird [1]. Auch zu CNNs können weitere Informationen im Anhang nachgelesen werden.

Mean Average Precision

Um die Genauigkeit von Objektdetektoren zu messen, wird oft die Metrik *mean Average Precision* (mAP) gewählt. Diese setzt sich aus zwei grundlegenden Größen zusammen:

$$\text{Precision} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}}$$

$$(2.6)$$

$$\text{Recall} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}}$$

Precision sagt also etwas über die Verlässlichkeit einer Klassifikation aus, während *Recall* Aussagen über die Erkennungsfähigkeit eines Objektdetektors trifft. Wichtig ist es hierbei anzumerken, dass mehrfach detektierte Objekte nur einmal als positiver Befund aufgefasst werden, die restlichen Detektionen gehen als *False Positives* ein [14].

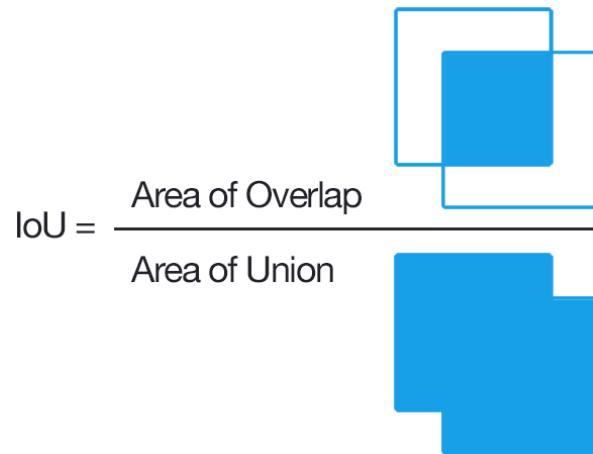


Abbildung 2.5.: Intersection over Union [15]

Die Tatsache, ob eine Bounding Box das gewünschte Objekt enthält und demnach ein positiver Fall vorliegt, wird anhand des *confidence scores* bestimmt. Er berechnet sich aus der Multiplikation der Wahrscheinlichkeit für eine Klasse mit der sogenannten *Intersection over Union* (IoU) (siehe Abbildung 2.5) der jeweiligen ausgewählten Bounding

Box. Die *IoU* beschreibt ein Maß der Überdeckung der detektierten Bounding Box zur wahren Bounding Box. Der *confidence score* sagt also etwas über die Gewissheit der Klassifikation aus. Für den kompletten Datensatz werden nun für unterschiedliche *confidence scores* jeweils *Precision* und *Recall* bestimmt und anschließend in einem Graphen aufgetragen. Meistens werden die *confidence scores* so gewählt, sodass sich eine äquidistante Abstufung in den *Recall* Werten ergibt [16].

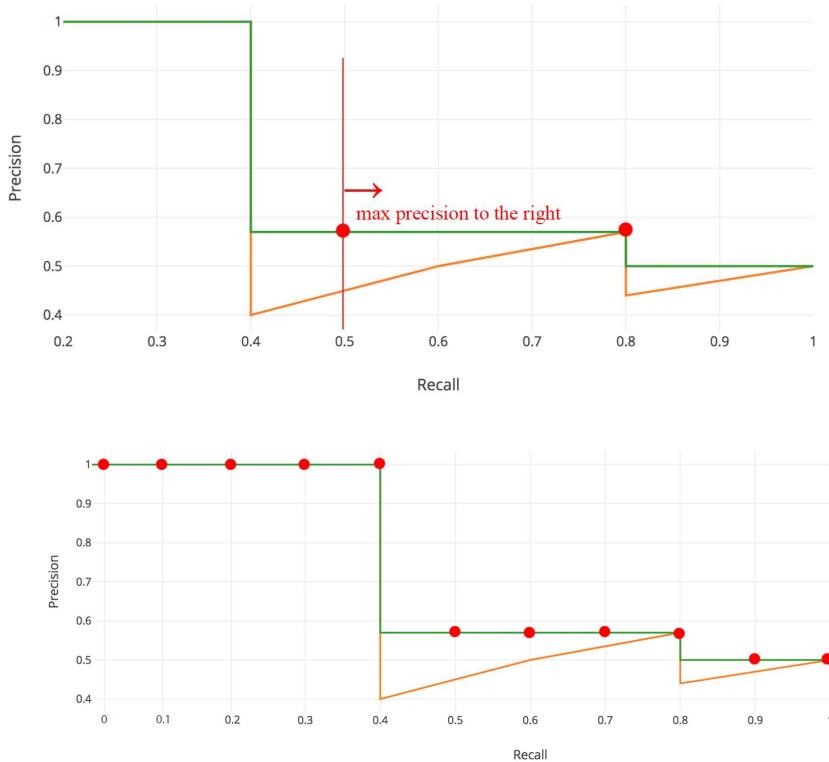


Abbildung 2.6.: Berechnung mAP [17]

Im Graphen ist meist ein klassisches „Zick-Zack“ Muster zu erkennen (siehe Abbildung 2.6). Dieses Muster wird geglättet, indem nach jedem Einbruch für jeden *Recall* Wert der maximale *Precision* Wert rechts des aktuellen *Recalls* übernommen wird. Wird anschließend das diskrete Integral über alle *Recall* Werte gebildet, so ergibt sich der *Average Precision* Wert für eine zu klassifizierende Kategorie. Der Mittelwert der *Average Precisions* über alle Klassifikationskategorien hinweg ergibt letztendlich den *mAP* Wert [17].

2.6. Objektdetektoren

Ein Teilziel der Machbarkeitsstudie ist es, anhand vorbestimmter Kriterien eine ausgewählte Menge von Objektdetektoren zu vergleichen. Um im Laufe der Arbeit zu verstehen, wie diese Auswahl zu Stande kommt und wie sich bestimmte Ergebnisse im Vergleich begründen lassen, ist eine Einführung in die unterschiedlichen Architekturen der Objektdetektoren unumgänglich.

Regional Convolutional Neural Networks

Regional Convolutional Neural Networks (R-CNNs) vertreten den Ansatz, für ein Bild mehrere Lokationsvorschläge für mögliche Objekte zu liefern, sogenannte *Regions of Interest* (RoIs), und diese anschließend zu klassifizieren.

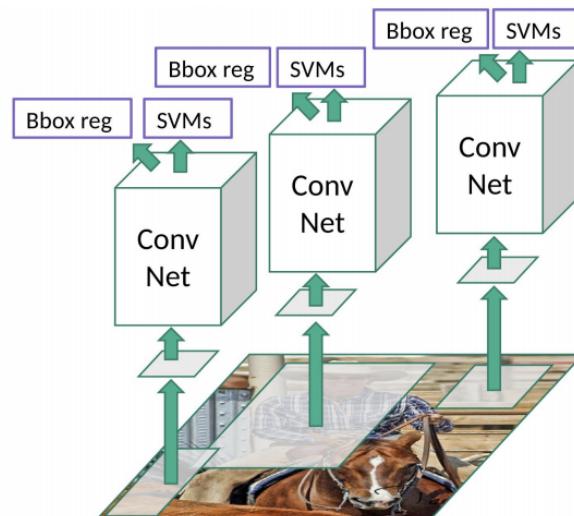


Abbildung 2.7.: R-CNN Architektur [18]

Bei dem klassischen *R-CNN* Detektor werden durch den *Selective Search* Algorithmus 2000 solcher *RoIs* vorgeschlagen. Zur Merkmalsextraktion wird für jede *RoI* anschließend ein CNN eingesetzt. Der resultierende *Feature Vektor* wird zur Klassifikation eines Objektes einer *Support Vector Machine* (SVM) unterzogen.

Um zusätzlich die Bounding Boxen akkurat zu bestimmen, wird der *Feature Vektor* zudem einem *Bounding Box Regressor* unterzogen (siehe Abbildung 2.7) [5].

Da der sogenannte *Region Proposal* Schritt durch den *Selective Search* Algorithmus allerdings viel Zeit in Anspruch nimmt, entstand eine Weiterentwicklung des *R-CNN* Netzes, das *Fast R-CNN* Netz. Dieses tauscht den Schritt des *Selective Search* Algorithmus mit dem Einsatz des CNNs. Außerdem wird das klassische CNN leicht angepasst. Bei *Fast R-CNN* wird ein Bild zunächst einem CNN unterworfen. Bevor eine *Feature Map* durch *Fully-Connected Layer* zu einem einzigen *Feature Vektor* vereinfacht wird, werden aus der *Feature Map* die verschiedenen *RoIs* extrahiert. Dies geschieht wiederum mit dem *Selective Search* Algorithmus, mit dem Unterschied, dass dieser nun nur auf der *Feature Map* operiert und nicht auf dem gesamten Bild. Durch *RoI Pooling Layer* werden die einzelnen entstandenen Regionen in eine feste Größe transformiert und einzeln einer Klassifikation durch *Fully-Connected Layer* und einer Softmax-Funktion unterworfen. Die Komponente mit dem *Bounding Box Regressor* bleibt gleich. Durch den Tausch des CNNs mit dem *Selective Search* Algorithmus werden die mathematischen Faltungsoperationen nur einmal statt 2000 Mal pro Bild ausgeführt, was die Performanz des Detektors gegenüber eines klassischen *R-CNNs* enorm steigert [19].

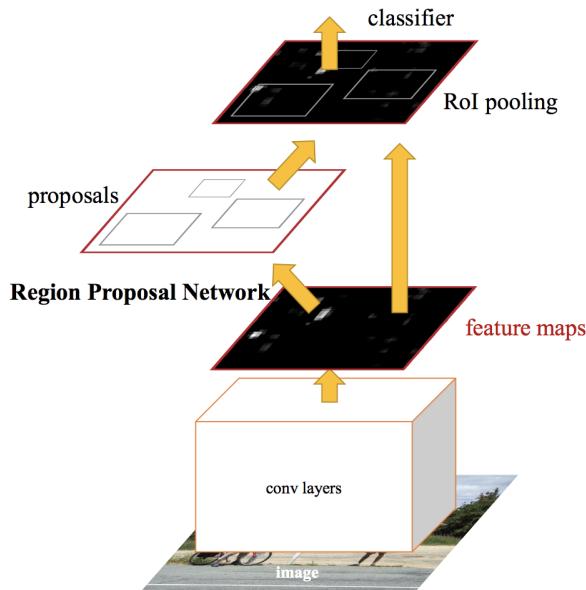


Abbildung 2.8.: Faster R-CNN Architektur [18]

Die letzte Optimierung der *R-CNN* Familie entstand durch das *Faster R-CNN* Netz. Dieses ersetzt den statischen *Selective Search* Algorithmus des *Fast R-CNN* Detektors durch ein eigenes lernfähiges, sogenanntes *Region Proposal Network* (RPN) (siehe Abbildung 2.8) [20].

Neben dem Einsatz von *R-CNN* Detektoren zur Objektdetektion existiert ebenso ein Ansatz zur instanzbasierten Segmentierung, das *Mask R-CNN* Netz. Es nimmt zwei wichtige Anpassungen an der Architektur des *Faster R-CNN* Netzes vor. Da bei Segmentierungsproblemen eine genauere Abgrenzung von Objekt und Hintergrund notwendig ist, wird das *RoI Pooling Layer* durch ein *RoI Align Layer* ausgetauscht. Hierbei wird das Rundungsproblem beim Pooling behoben. Angenommen eine *RoI* von 16x16 Pixeln wird mit einem *MEAN-Pooling Layer* der Schrittweite Drei verarbeitet, so ergibt sich pro Pooling Schritt ein Einzugsgebiet von 5.33 Pixeln. Dieses wurde abgerundet auf 5 Pixel. Bei *RoI Align Layer*n wird durch bilineare Interpolation der Wert des 5.33ten Pixels ermittelt und in das Pooling mit einbezogen. Dies ermöglicht eine genauere Segmentierung an den Grenzen eines Objektes.

Außerdem wird parallel zum RPN ein sogenanntes *Fully-Convolutional Network* (FCN) eingesetzt, einem Netz, dass rein aus *Convolutional Layer*n besteht. Es dient, um für jede existierende Klasse eine pixelbasierte binäre Maske auszugeben, die für jeden Pixel die Zugehörigkeit zu einer Klasse bestimmt. Basierend auf dieser Maske werden die detektierten Objekte anschließend farblich hervorgehoben [21].

Single Shot MultiBox Detector

Zwar liefern die oben genannten Objektdetektoren akkurate Ergebnisse, allerdings sind sie als zu rechenintensiv und langsam einzuordnen, als dass sie für Echtzeit Applikationen eingesetzt werden könnten. Der *Single Shot MultiBox Detector* (SSD) unterscheidet sich von vorhergehenden Modellen, wie beispielsweise den *R-CNN* Detektoren, dahingehend, dass er bewusst auf den Schritt der Generierung von Bounding Box Vorschlägen und des Poolings verzichtet, um wesentlich schneller ablaufen zu können als andere Objektdetektoren. Die Präzision der Klassifikationen bleibt hierbei erhalten, selbst Bilder niedriger Auflösung können weiterhin verarbeitet werden. Dem *SSD* genügt also ein einziges tiefes neuronales Netz zum Lokalisieren und Klassifizieren von Objekten.

Wie der *SSD* aufgebaut ist und welche Ansätze er verfolgt, soll in diesem Unterkapitel erläutert werden [11].

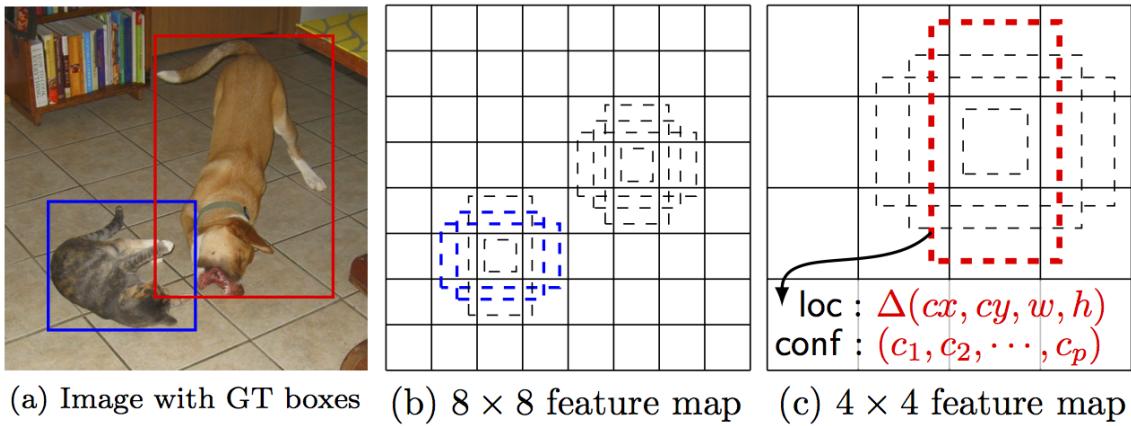


Abbildung 2.9.: SSD Bounding Box Vorschläge - Objekte unterschiedlicher Skalierung, hier Katze (blau) und Hund (rot), werden durch unterschiedlich große Gittereinteilungen und Bounding Box Seitenverhältnissen detektiert [11].

Die Architektur des *SSD* zielt darauf ab, durch unterschiedlich große *Convolutional Layer Feature Maps* unterschiedlicher Skalierung in die Klassifikation mit einfließen zu lassen. Anschaulich kann es sich vorgestellt werden, als werde das Bild in mehrere unterschiedlich große Gitterstrukturen unterteilt und die resultierenden Zellen jeweils einzeln klassifiziert. Dadurch ist es möglich, Objekte unterschiedlicher Größe zu erkennen. Für jede Zelle im Gitter wird eine gleiche Anzahl vordefinierter Bounding Boxen, die unterschiedliche Seitenverhältnisse aufweisen, definiert. Daher entstammt der Name „*MultiBox*“. Abbildung 2.9 zeigt beispielsweise, wie eine Katze (in blau) und ein im Vergleich zur Katze größerer Hund (in rot) durch unterschiedlich große Gittereinteilungen und Bounding Box Seitenverhältnisse detektiert werden. Durch die *MultiBox* Eigenschaft wird ebenso sichergestellt, dass sowohl horizontal als auch vertikal ausgeprägte Objekte in der selben Zelle gleichzeitig erkannt werden können (siehe Abbildung 2.10) [11].

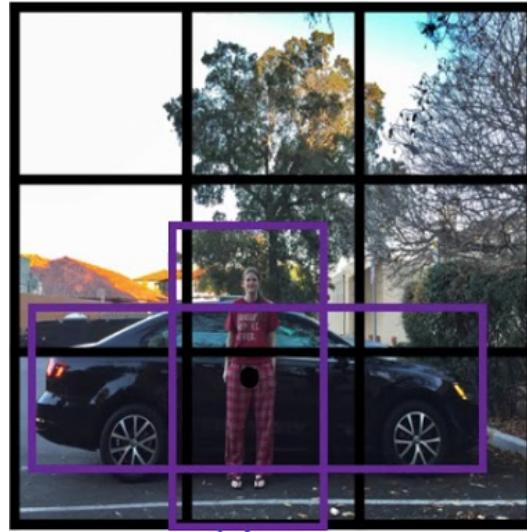


Abbildung 2.10.: Bounding Boxes - In einer Gitterzelle treten Bounding Box Vorschläge mehrerer unterschiedlicher Skalierungen auf, um auch zwei Objekte unterschiedlicher Skalierung, deren Zentrum in der selben Gitterzelle liegt, gleichzeitig detektieren zu können [22].

Für jede dieser Bounding Boxen bestimmt der *SSD* Wahrscheinlichkeiten für Klassen-zugehörigkeiten als auch Verschiebungen der vordefinierten Bounding Box zur wahren Bounding Box des Objekts für jede Klasse. Die Kostenfunktion ist durch die gewichtete Summe des Lokalisationsverlustes und des Klassifikationsverlustes bestimmt. Während der Klassifikationverlust durch eine Softmax-Funktion (2.1) bestimmt werden kann, wird der Lokalisationsverlust über die Smooth L1 Funktion (2.2) bestimmt [11].

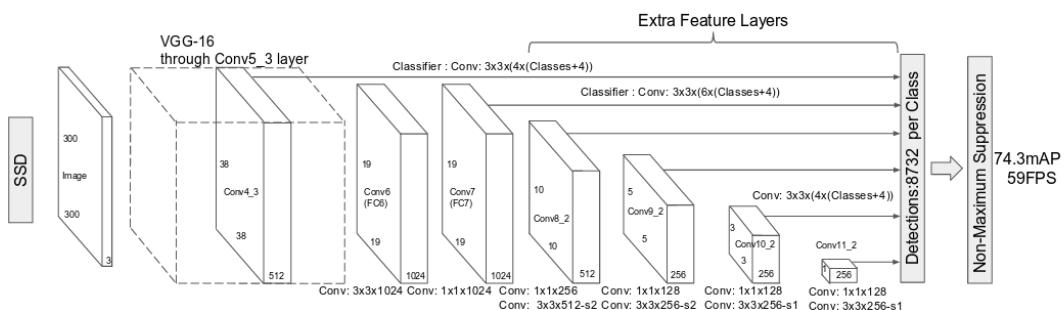


Abbildung 2.11.: SSD Architektur [11]

Technisch basiert der *SSD* auf der Idee eines *Feed-Forward Convolutional Networks* (siehe Abbildung 2.11). Er benutzt nach dem Prinzip des *Transfer Learnings* ein *VGG-16* Basis Netzwerk², dessen *Fully-Connected Layer* am Ende entfernt wurden. Die resultierende *Feature Map* wird nun einer Reihe von ständig kleiner werdenden *Convolutional Layern* unterzogen. Jedes *Convolutional Layer* kann eine feste Anzahl an Detektionen bestimmen. Eine Detektion wird durch eine Klassenangabe und die Lage einer vorhergesagten Bounding Box bestimmt. Eine Bounding Box wird wie bereits erläutert durch den linken oberen Eckpunkt $P(x, y)$ und eine Höhe und Breite bestimmt. Bei c Klassen hat der *Feature Vektor* einer Detektion demnach die Größe $c + 4$. Bei einer *Feature Map* Größe von $m \times n$ und k verschiedenen vordefinierten Bounding Boxen ergeben sich also $m \cdot n \cdot k \cdot (c + 4)$ verschiedene *Feature Vektoren* für eine *Feature Map* [11]. Diese *Feature Vektoren* werden nun an das Ende des Netzes zur Klassifikation weitergeleitet.

Dieser Vorgang wird für alle *Feature Maps* für alle *Convolutional Layer* durchgeführt. Die daraus folgende Menge an Detektionen wird durch ein *Non Maximum Suppression Layer* in ihrer Größe reduziert. Als Maß zur Filterung wird die *IoU* der detektierten Bounding Box zur wahren Bounding Box verwendet. Überschreitet diese einen Wert von 0.5, so ist diese der originalen Bounding Box zugeordnet. Demnach ist es auch möglich, dass eine originale Bounding Box mehreren vordefinierten Bounding Boxen zugeordnet werden kann [11].

Während des Trainingsprozesses des *SSD300*³ wurde eine Lernrate von $\eta = 10^{-3}$ für das *Mini-Batch* Verfahren mit Batchgröße 32 und Moment $\beta = 0.9$ verwendet. Die Gewichtungen wurden *Xavier* initialisiert. Nach 40.000 Iterationen wurde die Lernrate für 10.000 Iterationen auf $\eta = 10^{-4}$ reduziert und schließlich auf $\eta = 10^{-5}$ [11]. Auf Basis der *PascalVOC* Datensätze aus 2007 und 2012 wurde mit dieser Konfiguration eine *mAP* von 74.3% für *SSD300* respektive 76.8% für *SSD512* erreicht.

² *VGG-16* ist ein auf dem Datensatz von *ImageNet* basierendes neuronales Netz, das bis zu 1000 unterschiedliche Kategorien klassifizieren kann [23].

³ *SSD300* verwendet Bilder der Auflösung 300x300 Pixel. Alternativ existiert ebenso *SSD512* für Bilder der Auflösung 512x512 Pixel. Die Bilder können jedoch auch kleiner als die vorgegebene Auflösung gewählt werden.

You Only Look Once

Der Algorithmus *You Only Look Once* (YOLO) ist ein weiterer Objekterkennungsalgorithmus und betrachtet statt separaten Bildregionen das komplette Bild. Er benutzt nur ein neuronales Netz, um Bounding Boxen und Wahrscheinlichkeiten für bestimmte Klassen gleichzeitig vorherzusagen. Daher der Name „You Only Look Once“.

Hierzu wird ein $S \times S$ Gitter über das Bild gelegt. Für jedes Feld im Gitter werden durch ein einziges CNN B Bounding Boxen vorhergesagt, als auch die zur Zelle dazugehörigen Klassenwahrscheinlichkeiten. Für jede Bounding Box wird nun der *confidence score* für alle Klassen berechnet. Basierend auf den *confidence scores* wird ein Regressionsproblem zur richtigen Lokalisation gelöst. Aus der Menge der resultierenden Bounding Boxen werden schließlich mit Hilfe eines festgelegten Schwellwertes für den *confidence score* die Boxen mit lokalisierten Objekten bestimmt (siehe Abbildung 2.12) [24].

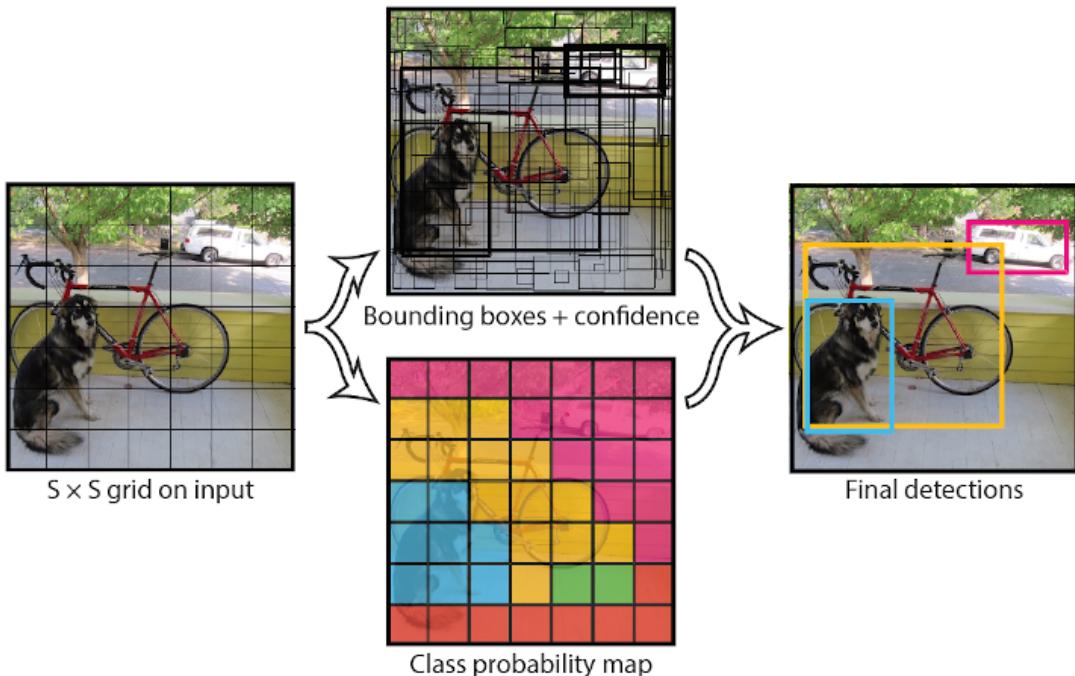


Abbildung 2.12.: Vereinfachte Darstellung des YOLO Algorithmus [24]

Die vorhergesagten Werte werden in einem $S \times S \times (B * 5 + C)$ Tensor kodiert, wobei S und B wie zuvor beschrieben durch das Gitter und die Bounding Boxen festgelegt sind und C die Anzahl der Klassen beschreibt. Abbildung 2.13 zeigt den Aufbau des CNN von *YOLO* für die Detektion. Es besteht aus 24 *Convolutional Layern* zur Feature Extraktion gefolgt von zwei *Fully-Connected Layern*, um die Klassenwahrscheinlichkeiten und Bounding Box Koordinaten zu bestimmen. Für die Genauigkeit bei der Detektion wird die Auflösung des Eingangsbildes verdoppelt [24].

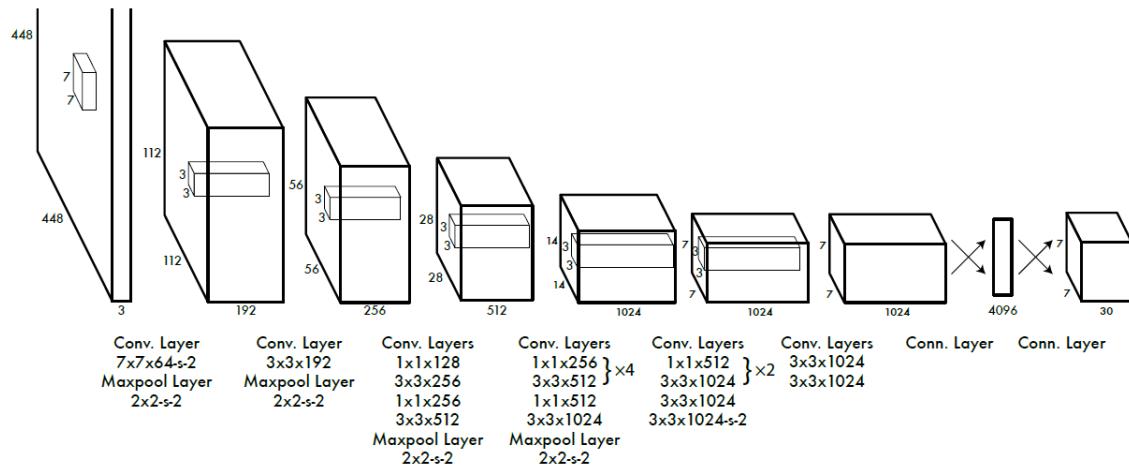


Abbildung 2.13.: YOLO Architektur [24]

In dem Netzwerk in Abbildung 2.13 wird ein Bild mit einer Auflösung von 224×224 Pixeln verwendet und die vorhergesagten Werte im $7 \times 7 \times 30$ Tensor ausgegeben. Im Trainingsprozess wird die Lernrate zu Beginn langsam von 10^{-3} auf 10^{-2} angehoben, um eine Divergenz wegen instabilen Gradienten zu vermeiden. Anschließend wird die Lernrate für 75 Epochen beibehalten, bis sie daraufhin für 30 Epochen auf 10^{-3} gesenkt wird. Zuletzt wird sie für weitere 30 Epochen auf 10^{-4} herabgesetzt [24].

Die mittlerweile dritte und aktuelle Version von *YOLO* weist einige Verbesserungen auf, gerade im Bezug auf die Erkennung von sehr kleinen Objekten wie zum Beispiel einzelnen Vögeln in einem Schwarm. Das in *YOLOv3* verwendete CNN umfasst insgesamt 53 *Convolutional Layer* und entstand aus mehreren Optimierungsschritten des in Abbildung 2.13 dargestellten CNN [25].

2.7. Datensatzformate

Basierend darauf, welcher Objektdetektor trainiert werden soll, muss der zum Training verwendete Datensatz in einem bestimmten Format vorliegen. Zum Trainieren des *SSDs* wird das sogenannte *Pascal Visual Object Classes* (PascalVOC) Format benötigt. Es definiert eine Unterteilung in *JPEGImages*, *Annotations* und *ImageSets*. Während in dem Ordner *JPEGImages* alle Bilder des Datensatzes vorhanden sind, befindet sich die Information über die vorhandenen Objekte in dem Bild im Ordner *Annotations*.

```

1 <annotation>
2   <folder>JPEGImages</folder>
3   <filename>000001.jpg</filename>
4   <path>..\JPEGImages\000001.jpg</path>
5   <source>
6     <database>Unknown</database>
7   </source>
8   <size>
9     <width>4608</width>
10    <height>2112</height>
11    <depth>3</depth>
12  </size>
13  <segmented>0</segmented>
14  <object>
15    <name>Saskia Wasser Groß</name>
16    <pose>Unspecified</pose>
17    <truncated>0</truncated>
18    <difficult>0</difficult>
19    <bndbox>
20      <xmin>1575</xmin>
21      <ymin>95</ymin>
22      <xmax>3163</xmax>
23      <ymax>635</ymax>
24    </bndbox>
25  </object>
26 </annotation>
```

Listing 2.1: PascalVOC Bildannotation

Für jedes Bild des Datensatzes werden die Informationen in einer gleichnamigen XML-Datei abgelegt (siehe Listing 2.1). Neben allgemeinen Metainformationen über das Bild befindet sich hier ebenso eine Liste aller markierten Objekte. Pro Objekt wird die Klassifikationskategorie, die Ausrichtung (z.B. „Frontal“), die Information über vollständiges Erscheinen im Bild, die Information über schwere Erkennbarkeit und die Bounding Box angegeben. Im Ordner *ImageSets/Main* wird eine Unterteilung in Trainings-, Test- und Validierungsdatensatz durch drei Textdateien realisiert, die die Dateinamen der Bilddateien als Auflistung enthalten [13, 26].

Das *YOLO* Format für den *YOLO* Objektdetektor definiert in einer *.names*-Datei alle im Datensatz vorhandenen Kategorien durch simple Auflistung der Bezeichner. Die Bilder werden zusammen mit ihren Annotationen in einem separaten Ordner abgelegt. Die Annotationen folgen hier dem Format:

```
< Kategorie – ID > < Zentrum – X > < Zentrum – Y > < Breite > < Hoehe >
```

Die Unterteilung in Trainings- und Testdatensatz erfolgt durch Referenzierung der Bildpfade in zwei getrennten Textdateien. Schließlich wird in einer *.data*-Datei der Pfad zu den beiden Textdateien und zur *.names*-Datei sowie die Anzahl an Kategorien gespeichert [27].

2.8. Cloud Infrastruktur

Das Trainieren eines *Deep Learning* Modells ist gerade bei großen CNN Architekturen rechenaufwendig. Tensor Operationen wie Matrixmultiplikationen und Konvolutionen erfordern im Rahmen des maschinellen Lernens hohe Parallelisierung und Taktfrequenzen, um in absehbarer Zeit gute Ergebnisse zu liefern. Die Rechenkapazität normaler Desktop-PCs reicht meist nicht aus, um performantes *Deep Learning* betreiben zu können.

Abhilfe bieten Software-as-a-Service (SaaS) bzw. Platform-as-a-Service (PaaS) Angebote wie *Amazon SageMaker*, *Google Cloud Platform Cloud AI* oder *Azure ML Services* oder aber auch Start-ups wie *FloydHub*. Diese bieten Infrastruktur in unterschiedlichen Zonen je nach Standpunkt der Rechenzentren zum Trainieren an sowie eine Plattform zum Verwalten der *Deep Learning* Prozesse.

Trainingshardware

Gerade GPUs bieten sich aufgrund ihres hohen Parallelisierungsvermögens gegenüber herkömmlichen CPUs an. Insbesondere der Hersteller *NVIDIA* nimmt hierbei eine Vorreiterrolle in der Produktion von Server-GPUs ein. Die *Compute Unified Device Architecture* (CUDA) von *NVIDIA* ermöglicht als Programmiermodell und paralleler Computing Plattform das Auslagern von Rechenprozessen auf GPUs. Das *CUDA* Toolkit beinhaltet GPU beschleunigte Bibliotheken, einen Compiler, Entwicklungswerkzeuge sowie die eigentliche *CUDA* Laufzeit und wird von vielen *Deep Learning* Bibliotheken genutzt, wie z.B. *PyTorch* [28, 29]. Auch bietet *NVIDIA* die sogenannte *NVIDIA CUDA Deep Neural Network Library* (cuDNN) an, die hardwareoptimierte Routinen zu beispielsweise Konvolution oder Pooling ermöglicht [30]. Hauptvergleichskriterien zwischen GPUs sind hierbei der Grad der möglichen Parallelisierung und die reine Rechenleistung im Verhältnis zum Stromverbrauch.

Neben GPUs existieren seit 2015 die von Google entwickelten *Tensor Processing Units* (TPUs). Diese sind speziell entwickelte, anwendungsspezifische integrierte Schaltung (engl.: *Application-Specific Integrated Circuit*) (ASIC) für Arbeitslasten im maschinellen Lernen [31]. Sie versprechen eine bis zu zehnmal schnellere Trainingsphase bei Tensor-Float-32 Werten [32].

Eine weitere Steigerung versprechen Microsofts Field Programmable Gate Arrays (FPGAs), die allerdings nicht weiter im Rahmen dieser Arbeit betrachtet werden sollen [33].

Amazon Web Services SageMaker

Amazon Web Services (AWS) bietet mit *SageMaker* eine integrierte Plattform zum Trainieren und Bereitstellen von *Deep Learning* Modellen. Zentrales Alleinstellungsmerkmal ist das einheitliche Toolset, in dem alle Arbeitsprozesse rund um ein *Deep Learning* Modell integriert abgebildet werden können. Es ist somit nicht mehr nötig, unterschiedliche Tools und Arbeitsabläufe zusammenfügen, was zuvor zeitaufwändig und fehleranfällig war [34].

Außerdem bietet *AWS SageMaker* die ersten vollständig integrierte Entwicklungsumgebung für Machine Learning, „*Amazon SageMaker Studio*“. Zum Erstellen der *Deep*

Learning Modellen werden sogenannte *Amazon Sagemaker Notebooks* genutzt, eine Ableitung klassischer *Jupyter Notebooks*. Unterstützte Frameworks sind TensorFlow, PyTorch, Apache MXNet, Chainer, Keras, Gluon, Horovod, Scikit-Learn und Deep Graph Library [34].

AWS SageMaker bietet verschiedene Instanztypen an, die sich je nach Anzahl an vCPUs und GPUs unterscheiden. Auch der vorhandene Arbeitsspeicher, Grafikkartenspeicher und die Netzwerkleistung kann durch die Vielzahl an angebotenen Instanzen nach individuellen Bedürfnissen gewählt werden [35].

Google Cloud Platform AI Platform

AI Platform ist das Konkurrenzprodukt zu *AWS SageMaker* von der *Google Cloud Platform* (GCP). Die Plattform bietet ebenso verschiedene Komponenten für das *Deep Learning* an. Hierzu gehören *AI Platform Notebooks*, ein Dienst mit einer integrierten JupyterLab-Umgebung, *Deep Learning* Virtual Machines (VM) mit vorinstallierten *Deep Learning* Frameworks, verteiltes Training mit automatischer Hyperparameter-Abstimmung durch den *AI Platform Training* Dienst oder *AI Platform Prediction* zum Bereitstellen trainierter Modelle [36].

Hervorzuheben sind insbesondere Googles *Tensor Processing Units* (TPUs). Dies sind spezielle Hardwarebeschleuniger, die speziell für *Deep Learning* Projekte im *TensorFlow* Framework optimiert wurden und für jede Instanz in der GCP mobilisiert werden können. Dabei wird pro TPU-Kern eine Rechenleistung von bis zu 92 TOPS erreicht [37]. Werden 2048 solcher TPU-Kerne zu einem TPU-Pod zusammen geschlossen, so ergibt sich eine Rechenleistung von über 100 PetaFLOPS [38]. Zudem ist die größere Rechenleistung gleichzeitig effizienter als herkömmliche GPUs [31]. TPUs sind also darauf ausgelegt, ein optimales Preis-/Leistungsverhältnis beim Trainieren von *Deep Learning* Modellen zu erreichen [36].

Google Colab

Google Colaboratory, kurz *Google Colab*, ist eine kostenfreie, cloudbasierte *Jupyter Notebook* Umgebung von Google. Dokumente, die in Google Colab erstellt werden, werden

automatisch mit *Google Drive* synchronisiert. Die Laufzeit ist frei konfigurierbar zwischen Python 2 und 3 bzw. zwischen einfachem CPU, GPU oder TPU Computing. Nachteil an dem kostenfreien *Google Colab* ist, dass zugewiesene Hardwareresourcen mit weiteren Nutzern geteilt werden müssen und so nicht die volle Rechenleistung für den individuellen Entwickler zur Verfügung stehen [39].

Auch können keine längerfristigen Trainingsjobs ausgeführt werden, ohne dass nach 90 Minuten der Client von dem zugewiesenen Server getrennt wird [40].

Microsoft Azure

Microsoft Azures Angebot für *Deep Learning* in der Cloud ist zunächst wenig transparent. Sie bieten ebenso wie Amazon und Google das Trainieren und Bereitstellen von *Deep Learning* Modellen an und zudem eine eigene DevOps Landschaft für solche Arbeitsprozesse. Auch werden Frameworks wie *TensorFlow* oder *PyTorch* unterstützt sowie das Programmieren in *Jupyter Notebooks* [41].

FloydHub

FloydHub, ein kalifornisches Start-up, bietet eine Data-Science Plattform zum Trainieren und Bereitstellen von *Deep Learning* Applikationen. FloydHub erlaubt es Anwendern, sich auf reines *Deep Learning* zu konzentrieren, während es die Arbeit rund um den *Deep Learning* Lebenszyklus abnimmt. Hierzu gehört das Bereitstellen der entsprechenden Hardware, das Installieren von Treibern oder das Integrieren verschiedener *Deep Learning* Bibliotheken, wie *TensorFlow*, *PyTorch* oder *Keras* [42].

Mit Hilfe des von FloydHub bereitgestellten Command Line Interfaces (CLI) kann ein lokales Projekt zu einem FloydHub Projekt initialisiert werden. Anschließend können anhand einer Konfigurationsdatei Einstellungen über das Training spezifiziert werden (siehe Listing 2.2). Alternativ können diese auch über das CLI festgelegt werden.

```
1 machine: gpu2
2 env: pytorch-1.4
3 input:
4     - destination: input
```

```
5      source: <username>/datasets/smartwarehousessd/3
6      - <username>/datasets/smartwarehousessd/3:ssd
7 description: Job to train the SSD
8 max_runtime: 3600
9 command: python train.py
```

Listing 2.2: Konfigurationsdatei zum Trainingsjob auf FloydHub

Hierbei kann zwischen der K80 (gpu) oder der V100 (gpu2) GPU für das Training gewählt werden. Auch die *Deep Learning* Laufzeitumgebung muss spezifiziert werden. Bei Bedarf auch zusätzliche Bibliotheken in einer *floyd_requirements.txt*-Datei zur Installation mit angegeben werden. Anschließend muss der Datensatz referenziert werden, mit dem das Modell trainiert werden soll.

Dieser Datensatz wird separat hochgeladen, da sich dieser im Gegensatz zum Programmcode nur selten ändert. FloydHub implementiert auf seiner Plattform eine Art Pfadsystem, unter dem Datensätze und Projekte abgespeichert werden. Diese Pfade werden in der Konfigurations-Datei zur Referenzierung genutzt. Um auch im Programmcode auf den Datensatz zuzugreifen, muss ein Mountname definiert werden. In obigen Beispiel wird dem Datensatz unter Verzeichnis *<username>/datasets/smartwarehousessd/3* der Mountname *ssd* gegeben. Das Verzeichnis zum Einlesen der Daten ist anschließend im Code unter */floyd/input/ssd/* erreichbar.

Mit dem CLI Befehl *floyd run* wird der Programm Code auf die Plattform hochgeladen und der in der Konfigurationsdatei angegebene Befehl ausgeführt. Daraufhin wird ein Job erstellt, versioniert und ausgeführt. Während der Job ausgeführt wird, wird dem Nutzer ein Einblick in die Konsolenausgabe gewährt sowie in Metriken zur Hardwareauslastung. In der Jobhistorie kann im Nachhinein jeder Job mit dem damals aktuellen Programmcode und Datensatz eingesehen werden. Auch Datensätze werden versioniert. Schreibrechte sind auf das Verzeichnis */floyd/home* begrenzt, hier können Zwischenspeicherpunkte des Modells abgelegt werden.

Neben klassischen Trainingsjobs können *Deep Learning* Modelle auch ganz einfach in *Jupyter Notebooks* erstellt werden. Hierzu muss in einem Projekt ein Workspace angelegt werden.

2.9. Drohnen

Was die gesetzlichen Anforderungen zu Drohnen betrifft, so wurden im Juni 2019 von der *European Aviation Safety Agency* (EASA) einheitliche Regeln veröffentlicht, die den Drohnenbetrieb in der Europäischen Union einheitlich regeln sollen. Da den Mitgliedsstaaten ein Zeitraum von einem Jahr zur Umsetzung der Regularien zugesprochen wurde, gelten in Deutschland weiterhin die Vorschriften der deutschen Drohnen-Verordnung von 2017 [43].

Diese schreiben unter anderem vor [44]:

- Eine Kennzeichnungspflicht ab einem Startgewicht von über 250g,
- Eine maximale Flughöhe von 100 Metern über dem Grund,
- Eine Haftpflichtversicherung und
- Flugverbotszonen (Wohngrundstücke, Naturschutzgebiete, etc.).

Analysiert man den Markt auf programmierbare Drohnen mit einem frei zugänglichen *Software Development Kit* (SDK) und integrierter Kamera, so fällt das Angebot sehr gering aus. Im Folgenden soll ein Überblick über die zwei verfügbaren Modelle *Ryze Tech Tello EDU* und *Parrot Bebop 2* gegeben werden [45, 46].

	Ryze Tech Tello EDU	Parrot Bebop 2
Gewicht in Gramm	87	500
Maximale Bildgröße	2592x1936	4096x3072
Video Aufnahme Modi	1280x720 30fps	1920×1080 30fps
Batterie Kapazität in mAh	1100	2700
Max. Flugzeit in Minuten	13	23
Max. Fluggeschwindigkeit in km/h	28.8	60
Flugstabilisierung	Ja	Nein
Preis in €	159	295

Tabelle 2.1.: Technische Daten zu programmierbaren Drohnen mit Kameraintegration

Der technische Vergleich (siehe Tabelle 2.1) bildet die Grundlage für die spätere Auswahl einer Drohne für die Umsetzung des *Smart Warehouse* Szenarios.

3. Konzeption

3.1. Übersicht

Zur Umsetzung des *Smart Warehouse* Szenarios sind einige fundierte, konzeptionelle Überlegungen notwendig, die in diesem Kapitel betrachtet werden sollen. Sie beschränken sich auf sechs übergeordnete Themenbereiche, die in Abbildung 3.1 dargestellt sind.

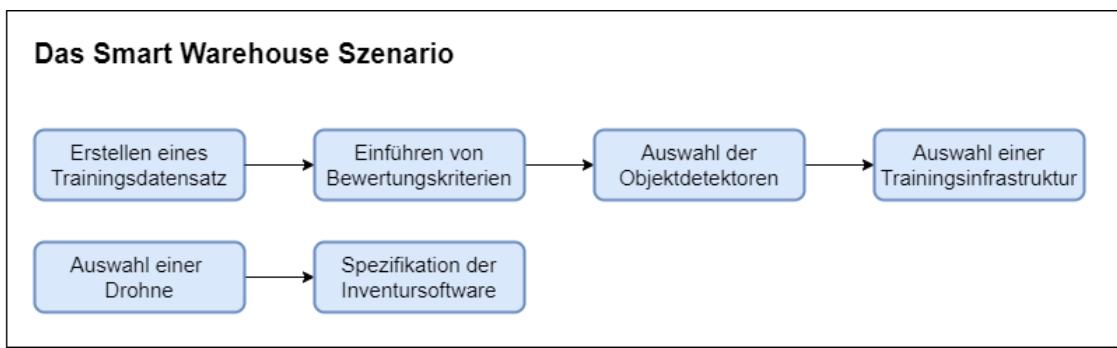


Abbildung 3.1.: Konzeptionelle Schritte

Daraus ergeben sich zwei Neuheitswerte im Rahmen dieser Studienarbeit. Zum einen soll evaluiert werden, wie gut die bestehenden Objektdetektoren für industrielle Anwendungsszenarien grundsätzlich geeignet sind, zum anderen, ob das spezifische Anwendungsszenario zur Durchführung einer Inventur von Warenhäusern mit einer Drohne prototypisch umsetzbar ist.

Für das *Smart Warehouse* Szenario ist demnach zum einen die Entwicklung eines *Deep Learning* Modells notwendig, das auf das Anwendungsszenario einer Inventur von Warenhäusern spezialisiert ist. Hierzu muss ein geeigneter Trainingsdatensatz erstellt werden, auf dem die ausgewählten Objektdetektoren trainiert werden sollen. Um die Objektdetektoren miteinander vergleichen zu können, ist zudem die Einführung von Bewertungskriterien notwendig. Auf Basis dieser Kriterien wird später evaluiert, wie gut die ausgewählten Objektdetektoren sich allgemein für einen industriellen Einsatz eignen.

Welche Objektdetektoren überhaupt im Sinne des *Smart Warehouse* Szenarios miteinander verglichen werden sollen, ist im nächsten Schritt „Auswahl der Objektdetektoren“ zu beschließen. Anschließend stellt sich nur noch die Frage, auf welcher Infrastruktur die Objektdetektoren trainiert werden sollen.

Um das zweite Ziel der Arbeit zu erarbeiten, kann auf den Ergebnissen des ersten Ziels aufgebaut werden. Eine Drohne soll die benötigten Daten zur Inferenz liefern. Welche Drohne dies sein soll, ist im ersten Schritt zu erarbeiten. Außerdem müssen die Daten der Drohne, hinsichtlich des Gedanken eine Inventur durchzuführen, passend verarbeitet und visualisiert werden. Diese *Inventursoftware* ist im letzten Schritt in ihrem Umfang zu spezifizieren.

3.2. Erstellen eines Trainingsdatensatzes



(a) Saskia Wasser Klein



(b) Saskia Wasser Groß



(c) Pepsi Cola Klein



(d) Pepsi Cola Groß



(e) ISO



(f) ACE



(g) Stenger Johannisbeerschorle



(h) Stenger Apfelsaftschorle



(i) Vitamalz Malzbier

Abbildung 3.2.: Die neun Datensatz-Kategorien

Das *Smart Warehouse* lehnt sich an ein großes Warenhaus an, bei dem Produkte nicht in Kartons verpackt, sondern als ganzes auf Regalen angeordnet sind, ähnlich wie bei Warenhäusern wie *Baumarkt* oder *Selgros*. Bei dem Aufbau des Trainingsdatensatzes hat die Machbarkeitsstudie allerdings nicht zum Ziel, ein solches Warenhaus vollständig im Datensatz abzubilden, sondern wesentlich den Datensatz so umfangreich zu wählen, um eine generelle Umsetzbarkeit des *Smart Warehouse* Szenarios zu beweisen. Im Rahmen des Projektes wurde sich deshalb exemplarisch auf Getränkeflaschen eines Warenhauses konzentriert. Dabei wurden neun Kategorien festgelegt (siehe Abbildung 3.2).

Der Datensatz ist 1.45 GB groß und besteht aus 1088 manuell annotierten Bildern. Sie dienen später dazu die ausgewählten Objektdetektoren zu trainieren. Die Bilder besitzen eine Auflösung von 2112x4608 Pixeln mit einer Farbtiefe von 24 Bit. Alle neuen Kategorien sind nahezu gleich häufig im Datensatz vorhanden.

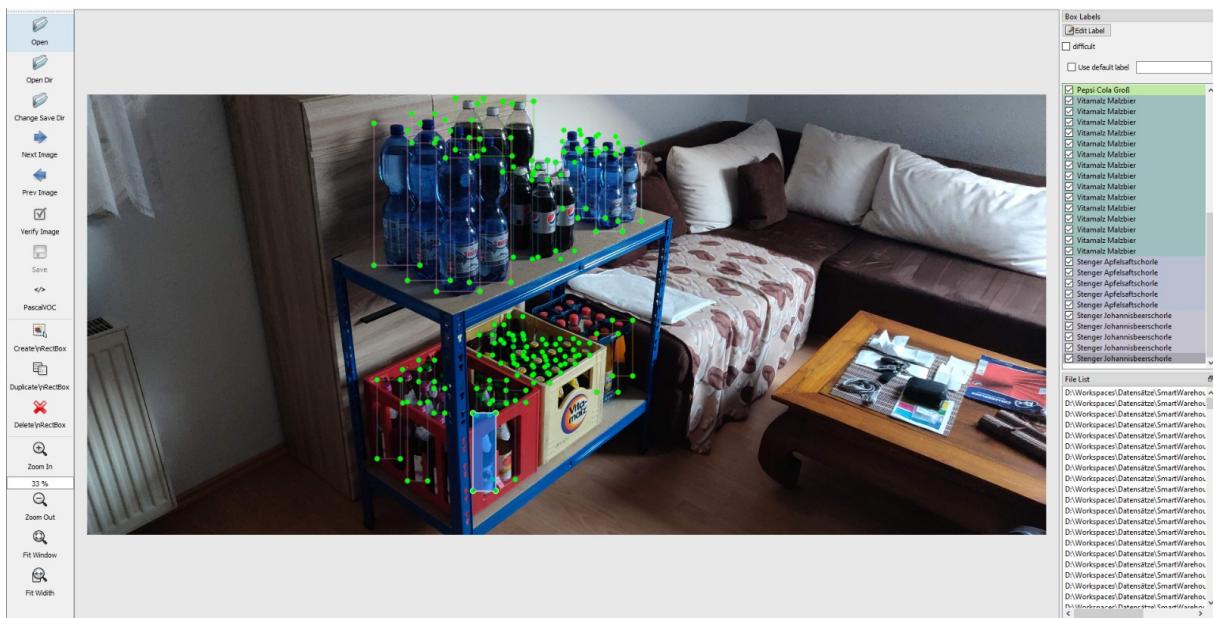


Abbildung 3.3.: Annotieren der Bilddaten mit LabelImg

Durch das Tool *LabelImg* wurden die Daten sowohl für das *PascalVOC*, als auch das *YOLO* Format annotiert (siehe Abbildung 3.3). Im initialen Datensatz sind auf 75% der Bilder die Objekte der jeweiligen Kategorien einzeln und klar erkennbar abgebildet (siehe Abbildung 3.2). Hierdurch wird erhofft, dass Modell zunächst auf die Muster der jeweiligen Objekte zu trainieren.

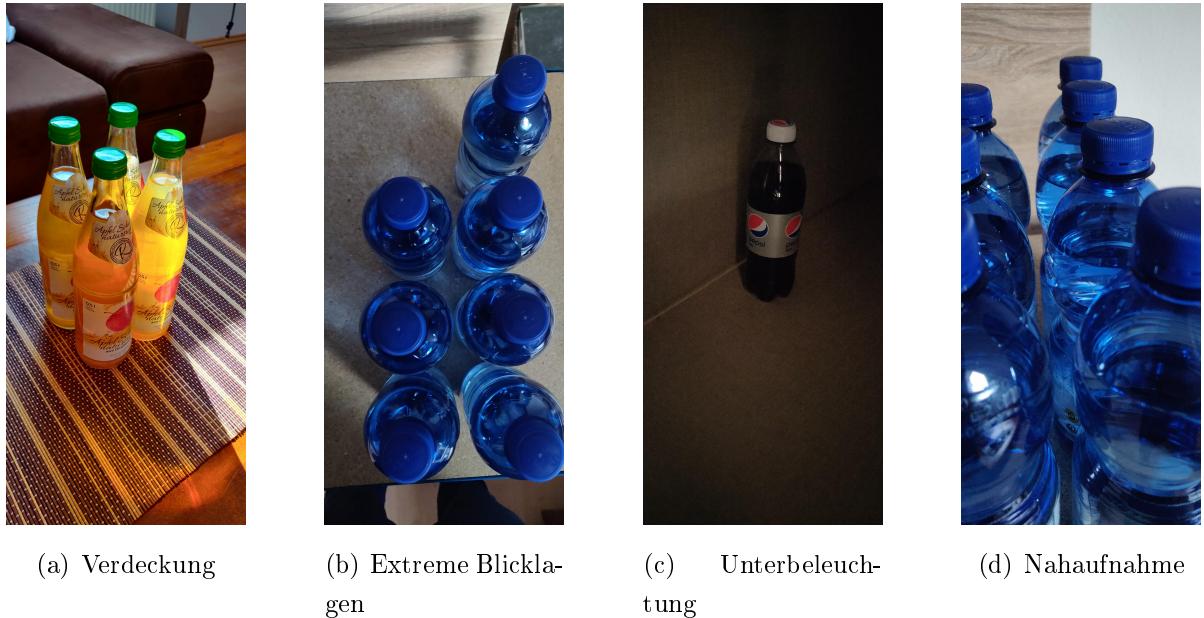


Abbildung 3.4.: Ausschnitt I aus dem Smart Warehouse Datensatz

Hinsichtlich der im folgenden Kapitel eingeführten Bewertungskriterien sind weitere Bilder so ausgewählt, dass ein möglichst gutes Inferenzverhalten, gerade bei schwierigen Aufnahmeständen und möglichen Problemsituationen, gewährleistet werden kann. So sind Verdeckungen von zu detektierenden Objekten zu 13% im Datensatz abgebildet (siehe Abbildung 3.4 (a)), Detektionssituationen aus extremen Blicklagen zu 4% (siehe Abbildung 3.4 (b)) und schwierige Beleuchtungs- & Lichtverhältnisse zu 8% (siehe Abbildung 3.4 (c)). Nur wenige Beispiele (2%) sind zu unterschiedlichen Entfernungen während der Detektion im Datensatz vorhanden (siehe Abbildung 3.4 (d)). Je nach Umgebung wurden Bilder zudem als schwer erkennbar markiert. Auch wurde darauf geachtet, die Hintergründe der zu detektierenden Objekte kontinuierlich zu variieren. Mit der Anreicherung des Datensatzes mit Repräsentationen von solchen erschwerten Aufnahmeständen wird erhofft, die Generalisierungsfähigkeit darauf aufbauender *Deep Learning* Modelle zu steigern.

Um schließlich das Warenhaus zu simulieren, bei dem mehrere Objekte auf einmal detektiert werden sollen, sind in 12,5% aller Bilder die Objekte auf Regalen angeordnet, jeweils hintereinander oder in Getränkekästen. Ein Ausschnitt hierzu kann in Abbildung 3.5 eingesehen werden.



(a) Objekte im Regal



(b) Objekte im Getränkekasten

Abbildung 3.5.: Ausschnitt II aus dem Smart Warehouse Datensatz

Aus dem Gesamtdatensatz von 1088 Bildern wurden 90 Bilder für die spätere Validierung herausgezogen. Die restlichen 998 bestehen aus 200 Bildern Testdaten und 798 Bildern für das Training. In 90% aller Bilder sind nur Objekte einer Klasse abgebildet, da Warenbestände ebenso meist nach den Waren gruppiert sind. Der Datensatz wurde sowohl für das *PascalVOC* Format als auch für das *YOLO* Format auf *Kaggle*¹ veröffentlicht [47].

3.3. Einführen von Bewertungskriterien

Um Objektdetektoren miteinander vergleichbar zu machen und um deren Potential zum industriellen Einsatz zu bewerten, müssen konkrete Bewertungskriterien eingeführt werden.

Präzision

Zur Messung der Präzision wird die Metrik *mAP* verwendet. Dies garantiert eine gute Vergleichbarkeit mit den veröffentlichten Leistungsmerkmalen der Objektdetektoren.

¹Kaggle ist eine Online-Plattform für Datenwissenschaftler. Hier werden regelmäßig neue Datensätze bereitgestellt, oft im Zusammenhang mit bestimmten dazugehörigen Problemstellungen, die es unter der Gemeinschaft zu lösen gilt. Für bestimmte Problemstellungen werden sogar Preisgelder für deren Lösung ausgezahlt.

Reaktionsvermögen

Um eine Verarbeitung in Echtzeit zu ermöglichen, muss gewährleistet sein, dass die Inferenzgeschwindigkeit mit dem Modell mit der eingehenden Bildrate einhergeht. Als Maßstab dafür dient die *Frames Per Second* (FPS) Zahl. Echtzeitfähigkeit in der Machbarkeitsstudie ist so definiert, dass die Inferenz mit dem Modell mindestens so schnell ablaufen muss, dass Änderungen in der Umgebung rechtzeitig von Objektdetektor noch wahrgenommen werden können.

Trainingsverhalten

Unter dem Punkt Trainingsverhalten wird zusammengefasst, wie schnell sich die einzelnen Modelle mit den unterschiedlichen Objektdetektoren trainieren lassen. Hierbei wird besonderer Fokus darauf gelegt, wie viele Trainingsepochen notwendig sind, bis der Gradient der Fehlerfunktion des neuronalen Netzes keine merkenswerten Fortschritte beim Abstieg auf Basis des verwendeten Datensatzes mehr erzielt.

Inferenzverhalten

Im Zuge der Evaluation des Inferenzverhaltens werden vier Kriterien betrachtet.

- Das Verhalten bei besonderen Beleuchtungsverhältnissen wie unterbeleuchteten oder überbeleuchteten Gegenden,
- Das Verhalten bei extremen Blicklagen auf Basis der Entfernung und des Winkels zum detektierenden Objekt,
- Das Verhalten bei nicht vollständig sichtbaren Objekten, z.B. bei Verdeckung und
- Der Umgang mit doppelt detektierten Objekten.

Da es schwierig ist, das Verhalten bei solchen Detektionsbedingungen numerisch zu erfassen, soll lediglich eine Einordnung in die Kategorien *gut* (+), *mittel* (0), und *schlecht* (-) erfolgen.

3.4. Auswahl der Objektdetektoren

Für das *Smart Warehouse* Szenario soll eine Auswahl zwischen den vier Detektoren *Faster R-CNN*, *Mask R-CNN*, *SSD* und *YOLO* getroffen werden. Als Vergleichsbasis dienen die bereits veröffentlichten Benchmarkergebnisse. Zur Evaluation der Machbarkeitsstudie werden die zuvor eingeführten Bewertungskriterien auf die aus dieser Auswahl resultierenden Objektdetektoren angewendet.

Method	mAP	FPS	batch size	# Boxes	Input resolution
Faster R-CNN (VGG16)	73.2	7	1	~ 6000	~ 1000×600
Fast YOLO	52.7	155	1	98	448×448
YOLO (VGG16)	66.4	21	1	98	448×448
SSD300	74.3	46	1	8732	300×300
SSD512	76.8	19	1	24564	512×512
SSD300	74.3	59	8	8732	300×300
SSD512	76.8	22	8	24564	512×512

Abbildung 3.6.: Vergleich SSD auf PascalVOC 2007²[11]

Abbildung 3.6 sind die Referenzergebnisse aus der wissenschaftlichen Veröffentlichung des *SSDs* [11]. Die Ergebnisse zeigen, wie die Objektdetektoren *SSD*, *YOLO*, *Fast YOLO* und *Faster R-CNN* untereinander abschneiden. Jeder der Detektoren besitzt eigene Charakteristika bezüglich der benötigten Auflösung der zu verarbeitenden Bilder, der Anzahl der generierten Bounding Boxen und der Batchgröße während des Trainings. Nach diesen Ergebnissen ist eindeutig festzustellen, dass der *SSD* bezüglich *mAP* mit 74.3% bzw. 76.8% am besten abschneidet. Unter Hinzunahme des *PascalVOC* 2012 und *Common Objects in Context* (COCO) *trainval135k* Datensatzes erreicht der *SSD* sogar das beste Ergebnis aus der ursprünglichen Veröffentlichung mit einer durchschnittlichen Präzision von 81.6%. *Faster-RCNN* kann zwar mit 73.2% bezüglich der *mAP* mithalten, ist allerdings mit nur 7 FPS nicht zur schnellen Inferenz ausgelegt. *YOLO* schneidet in beiden Kategorien schlechter als der *SSD* ab, er erzielt wesentlich eine *mAP* von 66.4% und eine Framerate von 21 FPS. Dem *SSD* gelingt es also, ein gutes Verhältnis zwischen Präzision und Reaktionsvermögen zu bewahren. Durch den Verzicht auf den Schritt der Generierung von Bounding Box Vorschlägen und des Poolings kann *SSD* deutlich schnel-

²Mask R-CNN wird in der wissenschaftlichen Veröffentlichung von SSD nicht aufgeführt. Beim Vergleich von Faster-RCNN (RoI-Align) mit Mask R-CNN ergibt sich eine mAP von 37.3 zu 38.2 auf Basis des COCO Datensatzes. Die FPS Anzahl betrug wesentlich 5 FPS [21].

ler ablaufen als die Vergleichsdetektoren, während durch das Vordefinieren von Bounding Boxen ebenso eine hohe Präzision erzielt werden kann [11].

Allerdings ergibt sich vor allem für kleine Objekte ein erschwertes Detektionsvermögen, da diese in den höherliegenden Convolutional Layern untergehen. Als Lösung hierfür kann eine erhöhte Inputgröße gewählt werden (vgl. *SSD512*) oder *Data Augmentation* für den Lernprozess angewandt werden [11].

Diese Probleme treten bei Netzen der *R-CNN* Familie nicht auf. Wird der *Faster-RCNN* Objektdetektor mit *Mask R-CNN* zur instanzbasierten Segmentierung auf Basis des COCO Datensatzes verglichen, so ergibt sich für *Mask R-CNN* mit 38.2% mAP nur eine geringe Verbesserung gegenüber *Faster R-CNN* mit 37.3% [21]. Für diesen Benchmark wurde wohlbemerkt das *RoI-Pooling Layer* des *Faster R-CNN* mit einem *RoI-Align Layer* zur besseren Vergleichbarkeit mit dem *Mask R-CNN* Objektdetektor ausgetauscht. Dennoch bleibt auch beim *Mask R-CNN* das Problem eines langsameren Inferenzverhaltens gegenüber dem *SSD* oder *YOLO* offen. Für einen generell industriellen Einsatz könnte dieses langsame Inferenzverhalten möglicherweise problematisch sein, doch für das konkrete Szenario von stehenden Getränkeflaschen in der Machbarkeitsstudie ist eine starke Gewichtung der FPS Metrik zunächst mit Vorsicht zu betrachten [48].

Ein weiteres Auswahlkriterium stellt dar, wie gut die Detektoren aufgesetzt und auf eigens erstellte Datensätze umkonfiguriert werden können. Nach Betrachtung mehrerer Repositories ließen sich der *YOLO* und *SSD* Objektdetektor einfach aufsetzen und auf eigene Datensätze anpassen, während bei *Faster R-CNN* und *Mask R-CNN* vermehrt auf Probleme gestoßen wurde. So ist Facebooks Implementierung von *Mask R-CNN* „*Detectron*“ beispielsweise nur auf Linux oder macOS lauffähig. Abgeleitete Repositories sind bereits als *deprecated* deklariert und werden nicht mehr gewartet. Ein manuelles Aufsetzen dieser Implementierungen ist nur unter großem Aufwand möglich und wurde aufgrund der limitierten Zeit nicht weiter fortgeführt. Auch die Referenzimplementierung von *Faster R-CNN* ist bereits als *deprecated* deklariert und verweist auf die *Mask R-CNN* Implementierung *Detectron*. Nebenläufige Implementierungen sind ebenso als *Legacy* Implementierungen vermerkt und nur zeitaufwändig manuell aufsetzbar, sofern sie die Windows-Plattform unterstützen.

Aufgrund dieser Umstände und des schlechteren Abschneidens in der zeitkritischen Modellinferenz wurden *YOLO* und *SSD* als die beiden Detektoren ausgewählt, um exem-

plarisch am *Smart Warehouse Szenario* die Fähigkeit von Objektdetektoren zum industriellen Einsatz zu evaluieren. *YOLO* wird im *Darknet* Framework implementiert, dessen ursprüngliche Variante nicht mehr gewartet wird und zudem keine offizielle Dokumentation für die Verwendung unter Windows bereitsteht. *YOLOv3* besitzt hingegen eine umfassende Dokumentation. *Darknet* lässt sich einfach auf eigene Datensätze anpassen im Gegensatz zur Referenzimplementierung des *SSD* im *Caffe* Frameworks. Aus diesem Grund wurde sich bei *SSD* für eine Custom Implementierung in *PyTorch* entschieden.

3.5. Auswahl der Trainingsinfrastruktur

Bei der Auswahl der Trainingsinfrastruktur wurden zunächst die Cloud PaaS-Angebote in Betracht gezogen. Diese ermöglichen meist eine weit bessere Performance als lokales Training. Wichtig bei der Auswahl war hierbei

- möglichst niedrige Betriebskosten,
- ein diverses Angebot an Hardware-Beschleunigern und
- ein einfaches Aufsetzen der Trainingsinfrastruktur.

Insbesondere sollten die Testversionen der jeweiligen Angebote zu Nutze gemacht werden, um niedrige Betriebskosten zu erreichen. Manche Testversionen bieten hierbei ein Startkontingent, das je nach ausgewählter Hardware unterschiedlich schnell bei Nutzung verbraucht wird, bei anderen Cloud Anbietern wird die Hardwarekonfiguration vorgegeben, die anschließend nur für eine bestimmte Zeitdauer unter Last genutzt werden kann. In Tabelle 3.1 sind die Ergebnisse der Untersuchung dargestellt.

	AWS	GCP	Azure	FloydHub	Colab
Nutzungsrahmen	50 Std.	300\$	200\$	2 Std.	Keine Beschränkung
Hardware-Beschleuniger	Nein	Ja	Ja	Ja	Ja
Setup-Komplexität	Hoch	Mittel	Mittel	Einfach	Einfach

Tabelle 3.1.: Kostenlose SaaS-Angebote der Cloud Anbieter

Amazon SageMaker bietet hierbei für 50 Stunden eine *ml.m4.xlarge* Instanz für Modelltrainingszwecke an [49]. Da diese allerdings nur 4 vCPUs und 16 GiB Arbeitsspeicher

umfasst, also keinerlei Cloud GPU als Hardwarebeschleuniger angeboten wird, wurde das Angebot wieder verworfen [35].

Auf Empfehlung wurde anschließend die *GCP* betrachtet. Diese bietet mit 300\$ Startguthaben für 12 Monate ein lukratives Angebot zum Ausprobieren von beliebigen *GCP* Produkten [50]. Die Benutzung der *Deep Learning VM* bietet zudem eine native Unterstützung des *PyTorch* Frameworks, was von der *SSD* Implementierung genutzt wird, und zugleich eine Auswahl aus vier gängigen Cloud GPUs, der *NVIDIA Tesla K80*, *NVIDIA Tesla P100*, *NVIDIA Tesla T4* und der *NVIDIA Tesla V100*. Um die Konfiguration der *Deep Learning VM* allerdings mit Auswahl einer Cloud GPU abschließen zu können, muss zunächst das mit dem Account verknüpfte Kontingent erhöht werden. Hierzu musste an das *GCP* Support Team ein offizieller Antrag gestellt werden. Aufgrund der geringen Kaufhistorie wurde der Antrag allerdings abgelehnt.

Microsoft Azure bietet für 200\$ bei einer Laufzeit von 30 Tagen Zugang zu allen *Microsoft Azure* Diensten [51]. Darunter gehört eine *NC6* Instanz mit sechs vCPUs und einer *NVIDIA Tesla K80* [52]. Da *Microsoft Azures* Angebot allerdings nur sehr oberflächlich beschrieben wurde, wurde sich letzten Endes auch gegen *Microsoft Azure* entschieden.

Als letzter Anbieter wurde *FloydHub* getestet. Hervorzuheben ist die besonders einfache Vorgehensweise bei der Account Erstellung und dem Aufsetzen der Trainingsinfrastruktur, was bereits in Kapitel 2.8 beschrieben wurde. *FloydHub* bietet 20 Stunden CPU Trainingszeit bzw. 2 Stunden GPU Trainingszeit auf einer *NVIDIA Tesla K80* [53]. Neben einer *NVIDIA Tesla K80* konnte ebenso Trainingszeit auf einer *NVIDIA Tesla V100* erworben werden. Zudem wurde das verwendete *PyTorch* Framework unterstützt. Aufgrund der einfachen Handhabung wurde sich trotz der erhöhten Kosten für *FloydHub* entschieden.

Während des Trainings mit der *NVIDIA Tesla K80* fiel allerdings auf, dass die Wahl dieser GPU keine großen Performance Verbesserungen gegenüber lokalem Training brachte. Während lokal innerhalb einer Stunde 16 Epochen durchlaufen werden konnten, waren dies bei *Floydhub* hingegen nur 8 Epochen. Dies veranlasste eine Gegenüberstellung gängiger Cloud GPUs mit lokalen GPUs, allen voran den bereits vorhandenen Desktop-Grafikkarten *NVIDIA GeForce GTX 1080* und *NVIDIA GeForce RTX 2060* (siehe Tabelle 3.2) [54].

	K80	P100	T4	V100	GTX 1080	RTX 2060
CUDA Cores	2496	3584	2560	5120	2560	2176
Tensor Cores	/	/	320	640	/	272
TeraFLOPS (Single Precision)	4,113	9,526	8,141	14,13	9,784	7,377
Memory Bandwidth (GB/sec)	240,6	732,2	320	897	345,6	448
Suggested Power Supply Unit	700	600	350	600	450	450

Tabelle 3.2.: Vergleich von GPUs nach Rechenleistung

Hierbei fällt auf, dass im Grad der Parallelisierung eine *NVIDIA Tesla K80* zwar mit den vorhandenen lokalen Grafikkarten mithalten kann, in der Anzahl an Rechenoperationen pro Sekunden allerdings weit schlechter abschneidet. Im Vergleich zu einer *NVIDIA GeForce GTX 1080* schneidet die *NVIDIA Tesla K80* bezüglich der Rechenleistung weniger als halb so schnell ab, was auch erklärt, warum nur halb so viele Epochen in einer Stunde auf der *FloydHub* Cloud trainiert werden konnten. Damit sich das Training in der Cloud nach Performance lohnt, muss demnach mindestens eine *NVIDIA Tesla V100* verwendet werden. Da diese allerdings mit 42\$ für zehn Stunden mehr als dreimal so teuer als eine *NVIDIA Tesla K80* für 12\$ ist und zusätzlich zu den GPU Kosten noch monatliche Account-Gebühren berechnet werden³, wurde sich nach nun nach Kosten-Nutzen Abwägung letzten Endes auf lokales Training festgelegt. Dies ist ebenso hinsichtlich des Trainings des *YOLO* Objektdetektors besser, da das sehr spezifische *Darknet* Framework, das in der Implementierung genutzt wird, bisher von noch keinem Cloud Anbieter unterstützt wurde. Das Trainieren des *YOLO* Objektdetektors in der Cloud hätte demnach eine Umentscheidung auf eine Alternativ-Implementierung in beispielsweise *TensorFlow* oder *PyTorch* nötig gemacht. Werden noch andere Anpassungen in der Programmlogik mit einbezogen, z.B. dass ein Zugriff auf das Dateisystem beim Erstellen von Dateien in der *SSD* Implementierung in der Cloud Umgebung nicht möglich ist, so kommen zusätzlich zeitliche Bedenken mit auf. Ein lokales Trainieren bietet unter den genannten Voraussetzungen somit eine weitaus bessere Umgebung.

Auch wurden Überlegungen zum Training in *Google Colab* unternommen, da hier ein einfaches Training mit TPUs ermöglicht werden kann. Da allerdings in *Google Colab* nach 90 Minuten ein Trainingsjob beendet wird und die Rechenressourcen neuen Nutzern

³ Je nach Account kann eine unterschiedliche Anzahl an Projekten erstellt und Speicherplatz verwendet werden. Die *Beginner* Ausstattung von einem Projekt und 10 GB Speicher ist allerdings kostenfrei.

zugewiesen werden, war diese Art des Trainings ebenfalls nicht möglich.

Aus arbeitstechnischen Gründen ist anzumerken, dass die beiden Objektdetektoren *SSD* und *YOLO* lokal jeweils auf unterschiedlicher Hardware trainiert werden, *SSD* auf einer *NVIDIA GeForce GTX 1080* und *YOLO* auf einer *NVIDIA GeForce RTX 2060* GPU.

3.6. Auswahl einer Drohne

Bei der Auswahl der Drohne sind vor allem zwei Bereiche zu betrachten:

- Die gesetzlichen Rahmenbedingungen zur Drohne und
- Die technischen Anforderungen an die Drohne.

Aufgrund der gesetzlichen Auflagen aus Kapitel 2.9 wurde der Beschluss gefasst, dass die auszuwählende Drohne nur innerhalb von geschlossenen Wohnräumen im privaten Betrieb genutzt werden soll und zudem ein Startgewicht von unter 250g besitzen soll. Um eine studentische Machbarkeitsstudie durchführen zu können, ist eine solche eingeschränkte Anwendungsumgebung ausreichend.

Die Programmierbarkeit der Drohne gehört zu der wichtigsten technischen Anforderung an die Drohne. Zudem soll sie eine integrierte Kamera aufweisen, die in der Lage ist, einen Video-Stream zur Laufzeit zur Verarbeitung bereit zu stellen. Hierzu wurden in Kapitel 2.9 bereits die beiden auf dem Markt verfügbaren Modelle gegenübergestellt. Eine Wahl kann demnach nur zwischen den beiden Modellen der *Bebop 2* von der Firma Parrot oder der *Tello EDU* Drohne von der Firma Ryze Tech getroffen werden. Die gesetzlichen Auflagen, speziell die Kennzeichnungspflicht, und die daraus abgeleiteten, projektkrelevanten Bedingungen lassen allerdings nur die *Ryze Tello EDU* Drohne zu, zudem sie mit vertretbaren Budget erworben werden kann. Des Weiteren wird noch eine Garantie zu präzisem Schweben geboten, was gerade für die Objektdetektion vorteilhaft sein wird. Im Vergleich dazu erscheint die *Parrot Bebop 2* vielmehr als eine Drohne, die für den Einsatz im Außenbereich konzipiert wurde.

3.7. Spezifikation der Inventursoftware

Abbildung 3.7 visualisiert das technische Architekturmodell des gesamten *Smart Warehouse* Szenarios.

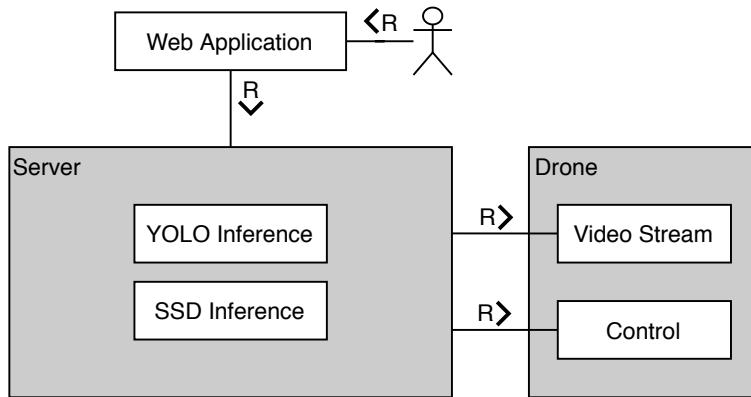


Abbildung 3.7.: Architektur des Smart Warehouse Szenarios

Um die Bilddaten der Drohne für eine Inventur zu verarbeiten, soll die Drohne von einer Client-Anwendung angesprochen werden. Der Server, der als Client agiert und im *Flask* Framework für Python zu schreiben ist, soll sich zur Drohne verbinden und die Flugsignale als Fluganweisungen zur Drohne senden. Da die Drohne keinerlei Sensoren zur Detektion möglicher Hindernisse enthält und es zudem nicht das Ziel ist, die Drohne automatisiert fliegen zu lassen, wird die Flugroute statisch als Flugsequenz festgelegt. Die Drohne soll dabei links oben starten und im Uhrzeigersinn die einzelnen Objektgruppen des Regals abfliegen. Dadurch soll eine möglichst genaue Detektion der einzelnen Objekte ermöglicht werden. Zum Schluss soll die Drohne den Blick auf das gesamte Regal ermöglichen und etwas zurück fliegen, damit auch die Detektion aller Objektklassen gemeinsam aus größerer Distanz getestet werden kann.

Auf dem Server soll zudem das *Deep Learning* Modell zur Inferenz integriert werden. Aufgabe des Servers soll es sein, die von der Drohne empfangenen Video-Stream-Frames mit dem Modell zu inferieren, die detektierten Objekte für die Inventur mit Hilfe eines Zählalgorithmus zu zählen und die Bilddaten mit den eingezeichneten Bounding Boxen anschließend per Livestream an eine Webapplikation zur Visualisierung weiterzugeben.

Eine *REST* Schnittstelle soll Aussagen über die Bestandsdaten des Warenhauses liefern und eine Möglichkeit bieten, die Flugsequenz zu initiieren.

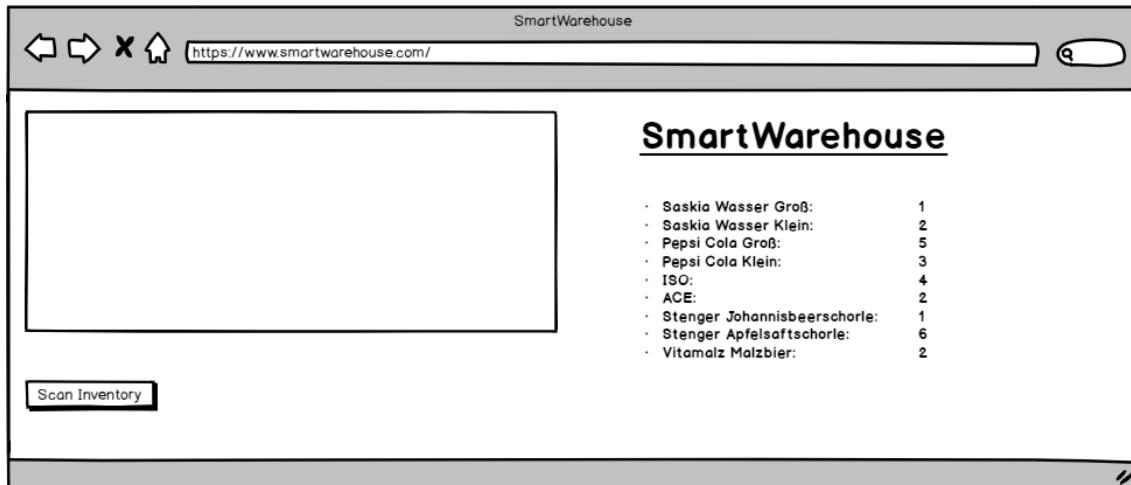


Abbildung 3.8.: Smart Warehouse User Interface Prototyp

Die Client-Anwendung soll das Live-Drohnenbild mit den eingezeichneten, erkannten Objekten zeigen. Zudem wird die Anzahl an erkannten Objekten der spezifischen Klassen rechts daneben dargestellt (siehe Abbildung 3.8).

4. Realisierung

Im folgenden Abschnitt soll auf die Implementierung des *Smart Warehouse* Szenarios eingegangen werden. Insbesondere werden Probleme während der Umsetzung der beiden Objektdetektoren *SSD* und *YOLO* betrachtet, die Realisierung der Dashboard-Webapplikation und des Zählalgorithmus zur Durchführung der Inventur aufgezeigt und letztendlich Herausforderungen im Rahmen der Drohnen Anbindung besprochen.

4.1. Umsetzung der Objektdetektoren

SSD

Die verwendete Custom-Implementierung in *PyTorch* realisiert die *SSD300* Variante des *SSDs*. Neben kleineren Änderungen in der Codebasis zur Erreichung von Kompatibilität mit aktuellen Bibliotheksversionen und weiteren Anpassungen zur Integration eines eigenen Datenbestandes, wurden vor allem drei größere Erweiterungen durchgeführt.

Da der erstellte Datenbestand nur 1088 gelabelte Daten enthält, wurde zusätzlich zur Custom-Implementierung ein sechsfaches Kreuzvalidierungsverfahren realisiert, um ein höheres Abstraktionsvermögen des Modells auf dem geringen Datenbestand zu erreichen. Auch unterstützte die Referenzimplementierung keine Validierung durch zuvor ungesetzte Daten. Die Modellklassen des Datenbestandes und die Validierungsskripte wurden dahingehend angepasst. Zudem fehlte eine Visualisierung der Entwicklung der Verlustkurven während des Trainingsverfahrens.

Die Referenzimplementierung erstellt sich des Weiteren einige Hilfsdateien, in der die Pfade zu Bildern und weitere Datenstrukturen für Trainingszwecke abgespeichert werden. Ohne diese ist kein Training möglich, das Trainingsskript fordert demnach Zugriff auf den Sekundärspeicher.

Um ein lokales Training auf der *NVIDIA GeForce GTX 1080* GPU zu ermöglichen, wurde zudem *CUDA* Version 10.1 verwendet. Trainiert wurde mit folgenden Hyperparametern:

- Batchgröße: 16
- Lernrate: $1.0 \cdot 10^{-3}$
- Momentum: 0.9
- Kreuzvalidierungen: 6 à 22 Epochen
- Epochen: 132
- Gradientenverfahren: Mini-Batch
- Aktivierungsfunktion: ReLU
- Kostenfunktion: Smooth L1

Das Basisnetzwerk des *SSDs* besteht aus einem auf *ImageNet* vortrainierten *VGG16*. Die restlichen *Convolutional Layer* sind *Xavier* initialisiert.

Die Hyperparameter sind nahezu gleich zu denen in der ursprünglichen wissenschaftlichen Veröffentlichung. Die Batchgröße im *Mini-Batch* Verfahren wurde für größere Stabilität von 32 auf 16 heruntergesetzt. Auch in der Evaluierung wurde die Batchgröße von 64 auf 48 verkleinert, da die Eingangsdaten eine weitaus höhere Auflösung als die ursprünglich im *PascalVOC* verwendeten Daten haben. Andernfalls wird Gefahr gelaufen, einen Speicherüberlauf zu verursachen. Die Lernrate bleibt während des gesamten Trainingsverlaufs gleich. Zudem wurde der Datensatz in fünf Untermengen unterteilt, in jedem Kreuzvalidierungsschritt dient jeweils eine Untermenge als Testdatensatz.

YOLO

Für *YOLO* wurde die neuste Version *YOLOv3* implementiert. Hierbei kommt das *Darknet* Framework zum Einsatz, eine Implementierung in C. Vor der initialen Kompilierung müssen einige Konfigurationsschritte unternommen werden, weil auch der Betrieb auf einer normalen CPU oder einer GPU mit Tensor Cores möglich ist. Für das Training kommt eine *NVIDIA GeForce RTX 2060 SUPER* GPU auf Basis von *CUDA* Version 10.0 zum Einsatz, deren enthaltenen Tensor Cores mitbenutzt werden. Zudem ermöglicht das *Darknet* Framework die Nutzung von *cuDNN* Version 7.4 für hardwarebeschleunigte Matrizenoperationen. Die benötigten Hyperparameter wurden dabei wie folgt gesetzt:

- Batchgröße: 64
- Subdivisions: 16
- Lernrate: $1.0 \cdot 10^{-3}$
- Momentum: 0.9
- Maximale Anzahl Batches: 18.000
- Gradientenverfahren: Mini-Batch
- Aktivierungsfunktion: LReLU
- Kostenfunktion: Summe der Fehlerquadrate

Im Unterschied zum *SSD* kann *YOLO* die in den GPU Speicher zu ladende Datenmenge eines Batches durch sogenannte *Subdivisions* festgelegt. Hierbei werden nur noch *Batchgroesse ÷ Subdivisions* Bilder gleichzeitig in die GPU geladen, um einen Speicherüberlauf zu vermeiden.

Im Gegensatz zu *SSD* wird kein Maximum für die Anzahl an Epochen, sondern ein Maximalwert für die zu durchlaufenden Batches gesetzt. Dieser Wert wird abhängig von der Menge an gelabelten Klassen gesetzt und kann als Empfehlung aus der Dokumentation für das *Darknet* Framework entnommen werden. Die Lernrate wird auf Basis dieses Maximalwertes an zu durchlaufenden Batches festgelegt. Zu Beginn beträgt sie 10^{-3} , nachdem 80% (14.400) aller Batches durchlaufen sind, wird sie auf 10^{-4} und schließlich nach 90% (16.200) aller Batches auf 10^{-5} herabgesetzt.

4.2. Dashboard Entwicklung

Der Server wurde mit dem *Flask* Framework in Python implementiert und läuft auf dem *Web Server Gateway Interface* (WSGI) Server *Waitress*. Er führt den Inferenzalgorithmus des *SSDs* bzw. des *YOLO* Objektdetektors für jeden Frame des empfangenen Video-Streams aus und streamt die inferierten Bilder mit den Bounding Boxen an jeden Client.

```

1 | def gen():
2 |     global counter, detected_objects
3 |     while True:

```

```

4     frame = tello.read()
5     if frame is not None:
6         frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
7         fps = FPS().start()
8         frame, new_detected_objects, counter =
9             YOLO.infer(frame, detected_objects, counter)
10        if len(new_detected_objects) != 0:
11            detected_objects = new_detected_objects
12        fps.update()
13        fps.stop()
14        , encodedImage = cv2.imencode('.jpg', frame)
15        print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
16        yield (b'--frame\r\n'
17               b'Content-Type: image/jpeg\r\n\r\n' + bytearray(encodedImage) + b'\r\n')
18
19
20 @app.route('/video_feed')
21 def video_feed():
22     return Response(gen(), mimetype='multipart/x-mixed-replace; boundary=frame')

```

Listing 4.1: Streaming der inferierten Bilddaten

Der Client wurde mit dem *Bootstrap* Framework grafisch gestaltet (siehe Abbildung 4.1).

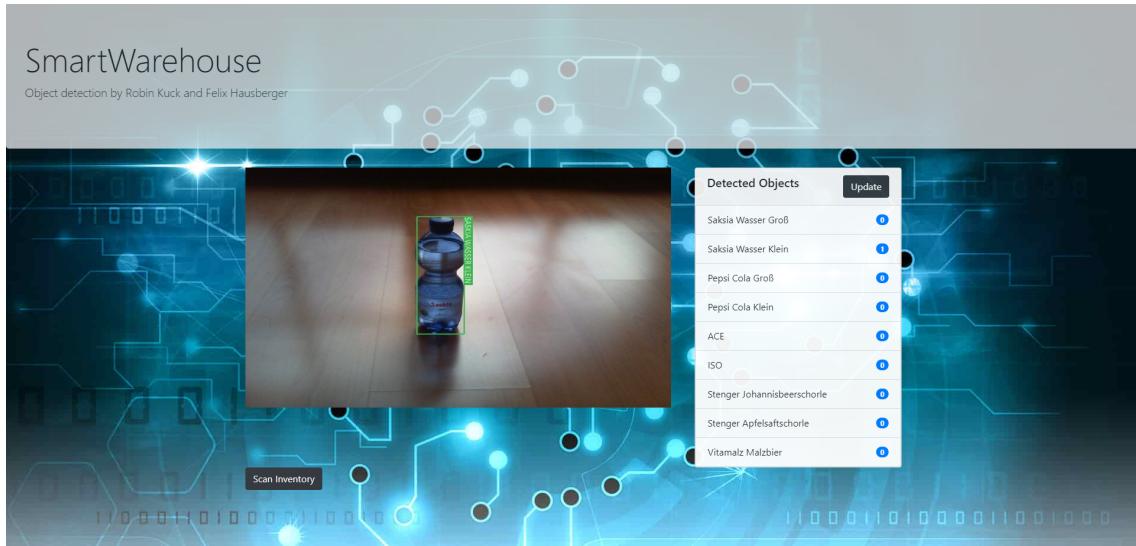


Abbildung 4.1.: Webapplikation Smart Warehouse

Auf Anfrage eines Client kann dieser die Flugsequenz initiieren und die aktuell gezählten Objekte vom Server abfragen. Der Zählalgorithmus hierzu wird im folgenden Unterkapitel erklärt.

4.3. Zählalgorithmus

```

1  definiere zähleNeueObjekte(Zähler, zuletzt detektierte Objekte):
2      Für alle detektierten Objekte o_neu:
3          Gefunden = falsch
4          Für alle zuletzt detektierten Objekte o_alt:
5              Wenn Label(o_neu) gleich Label(o_alt) UND
6                  Abstand der Bounding Boxen unter Schwellwert:
7                      Gefunden = wahr
8                  Abbruch
9
10         Wenn nicht gefunden:
11             Erhöhe Zähler für detektiertes Objekt

```

Listing 4.2: Zählalgorithmus zum Zählen der detektierten Objekte

Der Algorithmus 4.2 wird momentan dazu verwendet, um für das Industrieszenario einer Inventur einzelne detektierte Objekte zu zählen. Nach jedem Inferenzvorgang werden die detektierten Objekte abgespeichert, sodass der Algorithmus bei Aufruf sowohl auf die zuletzt detektierten Objekte *o_alt*, als auch auf die neu detektierten Objekte *o_neu* zugreifen kann. Für alle detektierten Objekte im aktuellen Durchlauf durchsucht er alle zuletzt detektierten Objekte, um herauszufinden, ob ein Objekt bereits im vorherigen Bild vorhanden war. Ausschlaggebend hierfür ist das gleiche *Label* als auch die Distanz der Bounding Boxen des aktuell detektierten Objekts zum Objekt der Vorrunde. Nur falls das Objekt nach diesen Kriterien nicht bereits in der Vorrunde vorhanden war, wird es gezählt. Das Problem, das der Algorithmus zu lösen versucht, enthält allerdings eine weitere Komplexitätsstufe. Dasselbe Objekt kann im Laufe der Inventur erneut auftreten und dessen relative Position im Bild ist ebenso variabel. In diesem Falle werden Objekte doppelt gezählt.

4.4. Drohnen Anbindung

Für die Steuerung sowie den Zugriff auf den Video-Stream bietet die *Ryze Tello EDU* ein eigenes WiFi Netz an, zu dem sich das Steuergerät verbinden muss. Die Kommunikation erfolgt über das *UDP* Protokoll¹ und besteht aus mehreren Kommunikationskanälen. Das Senden von Flugkommandos und das Abfragen des Video-Streams läuft über eine bidirektionale Socket Verbindung zwischen Client Applikation und der *Ryze Tello*

¹UDP steht für User Datagram Protocol. Es ist ein Transportprotokoll, welches im Gegensatz zu TCP verbindungslos und nicht zuverlässig ist.

EDU. Das liegt daran, dass die *Ryze Tello EDU* auf jede erhaltene Nachricht auch eine Antwort zurücksendet. Damit das Warten auf die Antworten von zuvor gesendeten Fluganweisungen und auf den Empfang des Video-Streams nicht den Programmablauf blockiert, werden in der Modellklasse der *Ryze Tello EDU* die Rückroutinen in einen eigenen Thread ausgelagert.

Neben Befehlen zur Steuerung existieren ebenso Befehle zum Auslesen von Informationen wie zum Beispiel der Geschwindigkeit der Drohne oder des Ladezustands der Batterie [55].

Zur Kodierung der Videodaten wird der H.264 Codec verwendet. Um die Rohdaten beim Datenempfang wieder zu dekodieren, wird ein spezieller Decoder benötigt, der allerdings als externe Bibliothek nicht in Python, sondern in C++ geschrieben wurde. Beim Compilieren mit *CMake* wird eine *.pyd* Datei erstellt, die als Bibliothek in die Python Umgebung eingebunden werden kann. An dieser Stelle sei vermerkt, dass zum Compilieren des Decoders veraltete Bibliotheken von *FFmpeg*² und *Boost.Python*³ benötigt werden, die nicht mehr verfügbar sind. Das Repository der *Ryze Tello EDU* enthält allerdings exklusiv für Windows eine vorkompilierte Variante des Decoders. Die generierte Python Bibliothek ist allerdings nur in Python 2.7 lauffähig.

Flugsequenz

Für die Flugsequenz bietet die Kommunikation durch eine Socket⁴ Verbindung die ideale Lösung. Python bietet ein *socket* Modul an, welches verschiedene Typen von Sockets unterstützt. Das nachfolgende Listing zeigt die Initialisierung einer Instanz der Socket Klasse:

```
1 | self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
2 | self.socket.bind(('localhost', 9000))
```

Listing 4.3: Initialisierung der Socket Klasse

Die Methode *socket()* erzeugt eine neue Instanz der Socket Klasse und benötigt zwei Parameter für die Adressfamilie und den Verbindungstyp. *AF_INET* steht für Internet-

²Multimedia Framework zum Verarbeiten von Video- und Audiomaterial

³Framework, das eine Schnittstelle zwischen Python und C++ bietet

⁴Mit Sockets können Daten bidirektional zwischen Programmen ausgetauscht werden, die entweder auf dem gleichen oder auf einem anderen durch eine Netzwerkadresse erreichbaren PC laufen.

Socket-Adresse, die sich aus IP-Adresse und Portnummer zusammensetzt. Der Parameter *SOCK_DGRAM* gibt an, dass die Socket Verbindung über das UDP-Protokoll aufgebaut wird. Für das Empfangen von Antworten wird die Instanz zusätzlich an eine Adresse (hier *localhost:8999*) gebunden.

Die Flugkommandos werden als UTF-8 kodierte Zeichenketten an die Drohne über den Aufruf *socket.sendto(msg, adress)* an den Socket übermittelt. Dieser erhält neben dem Befehl auch die Zieladresse, die als Tupel bestehend aus IP-Adresse und Port übergeben wird. Im nächsten Schritt wird mittels einer Schleife auf das Erreichen der Antwort gewartet, die abbricht, sobald eine gewissen Zeitgrenze übertreten wurde. Die blockierende Schleife stellt sicher, dass alle Nachrichten synchron und in der gleichen Reihenfolge an die Drohne gesendet werden.



Abbildung 4.2.: Drohnenflugsequenz

In der Flugsequenz fliegt die Drohne zunächst nah vor einem Regal mit zwei Böden her, um alle Objekte aus der Nahaufnahme detektieren zu können, danach fliegt die Drohne zurück, um das gesamte Regal im Blickwinkel zu haben (siehe Abbildung 4.2). Die finale Befehlskette wird wie folgt festgelegt:

- Command (Übernahme der Steuerung)
- streamon (Einschalten von Video-Stream)
- up 150 (Aufsteigen um 150 cm)
- right 100 (Nach rechts fliegen um 100 cm)
- down 100 (Absteigen um 100 cm)

- left 100 (Nach links fliegen um 100 cm)
- up 100 (Nach oben fliegen um 100 cm)
- right 50 (Nach rechts fliegen um 50 cm)
- back 150 (Nach hinten fliegen um 150 cm)
- streamoff (Ausschalten von Video-Stream)
- land (Landung)

Modellinferenz

Die Befehlssequenz zur Steuerung der Drohne wird parallel zur Inferenz der Bilder ausgeführt, wie durch Abbildung 4.3 verdeutlicht.

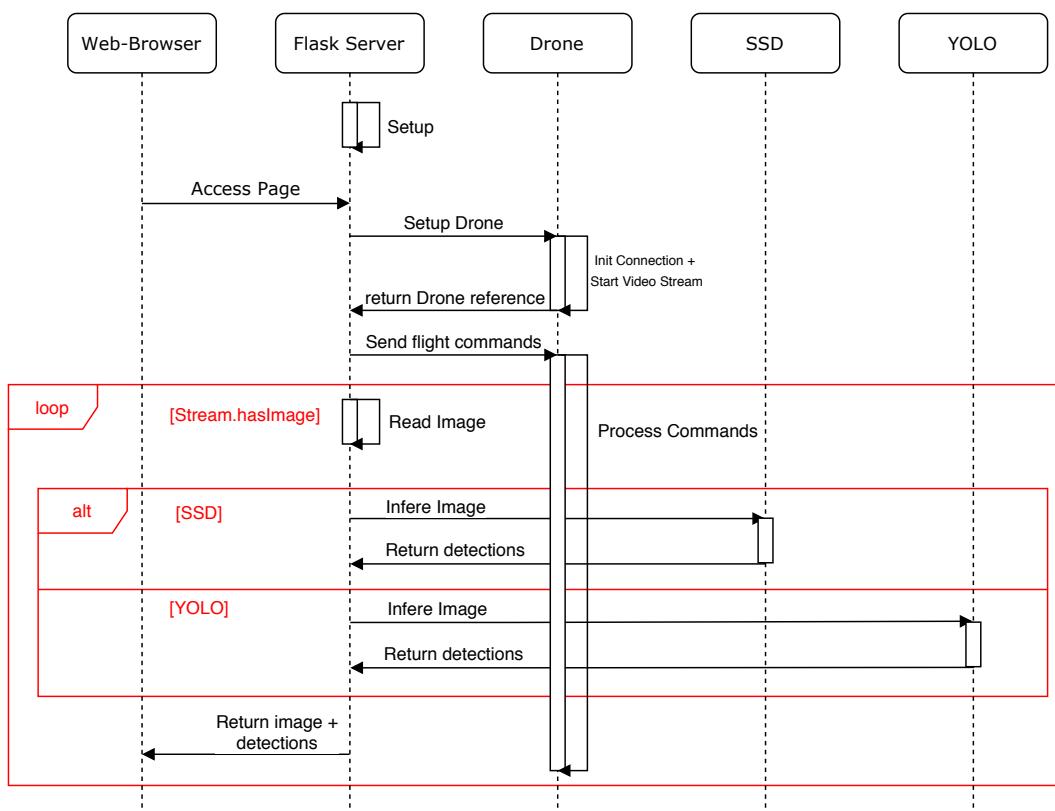


Abbildung 4.3.: Ablaufsequenz für die Verarbeitung von Drohnenbefehlen und Inferenz von Bildern

Der Video-Stream der Drohne kann auf dem zentralen Server durch die *Tello* Modellklasse Frame für Frame abgefragt werden. Dadurch, dass die *Ryze Tello EDU* Drohne einen stabilen Flugmodus anbietet, entstehen keine Probleme bei der Bildaufnahme. Auch sind die Bilder nicht verrauscht (siehe Abbildung 4.4).



Abbildung 4.4.: Bild der Video-Kamera der Ryze Tello EDU Drohne

Bei der Inferenz stellt sich nun ein zentrales Problem heraus. Da der H.264 Decoder nur auf Python Version 2.7 lauffähig ist und das verwendete *PyTorch* Framework für den *SSD* bzw. das *Darknet* Framework für *YOLO* nur auf Python Version 3 ausgeführt werden können, war es nicht möglich, eine Inferenz von Live-Video Daten der Drohne zu ermöglichen. Python Version 3 ist grundsätzlich nicht abwärtskompatibel zu früheren Versionen, demnach war der vom *Ryze Tello EDU* SDK geforderte H.264 Decoder nicht mit der bestehenden Infrastruktur vereinbar. Aufgrund dessen wurde sich dazu entscheiden, die Problemstellung zu vereinfachen, indem statt Bilder der Drohne Bilder einer Webcam inferiert werden. Hierzu wird eine *Microsoft LifeCam HD-3000* verwendet, die für bessere Vergleichbarkeit Bilder gleicher Auflösung und gleicher Bildrate liefert. Durch die *VideoCapture* Klasse von *OpenCV* ist das Integrieren der Webcam nahezu problemlos umsetzbar.

Da die verwendete Implementierung von *SSD* ebenso in Python geschrieben ist, konnte eine einfache Integration des *Deep Learning* Modells zur Inferenz auf den Server er-

möglich werden. Da das *Darknet* Framework für *YOLO* hingegen allerdings wie zuvor angemerkt in C implementiert ist, kann keine nahtlose Inferenz auf dem Python Server wie bei *SSD* umgesetzt werden. Hierfür muss auf die Kompilierung einer *Dynamic Link Library* (DLL) zurückgegriffen werden. Eine *DLL* ist eine ausführbare Datei, die Funktionen und Ressourcen als geteilte Bibliothek bereitstellt. Programme, die in verschiedenen Programmiersprachen implementiert wurden, können dadurch die gleiche DLL-Funktion aufrufen und die Inferenz mit *YOLO* kann somit durch das Python Programm aufgerufen werden [56].

5. Ergebnisse

In diesem Kapitel werden die Ergebnisse des Trainings und des Einsatzes der Objekt detektoren nach den in Kapitel 3.3 definierten Kriterien dargestellt. Es beinhaltet die Ergebnisse zur Präzision, zum Inferenzverhalten, zum Reaktionsvermögen und zum Trainingsverhalten der Detektoren.

5.1. Präzision und Inferenzverhalten

Ursprünglich wurden 500 Epochen für das Training von *SSD* vorgesehen. Da allerdings beim Training schon nach knapp über hundert Epochen sich der Gradient der Kostenfunktion nur träge veränderte, wurde im Sinne des *Early Stoppings* nach 121 Epochen das Training vorzeitig beendet, um *Overfitting* zu vermeiden. Abbildung 5.1 zeigt den Verlauf der Trainingsverlustkurve und der Testverlustkurve während des Kreuzvalidierungsverfahrens.

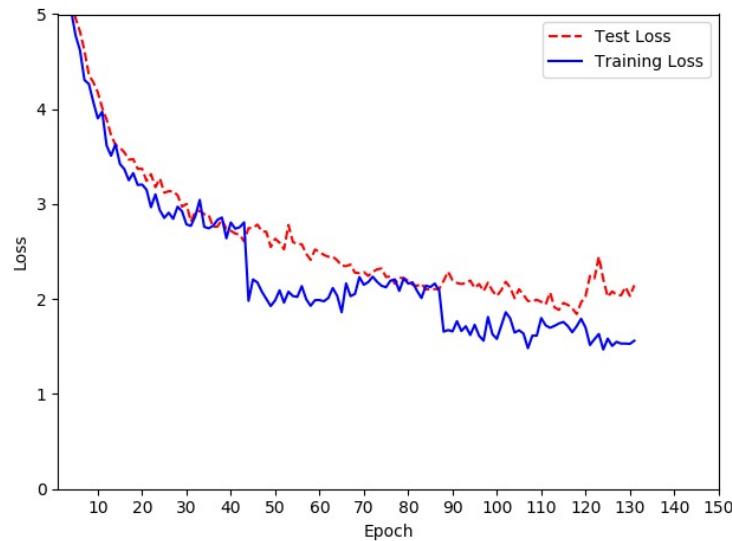


Abbildung 5.1.: Entwicklung der SSD Trainings- und Testverlustkurve im Training

Die Entscheidung zum *Early Stopping* basiert auf dem Anstieg der Differenz zwischen Trainings- und Testverlustkurve ab Epoche 122, was auf *Overfitting* schließen lässt. Zur Epoche davor war die Differenz der beiden Kurven am niedrigsten bei vergleichsweise geringem Verlust im Trainingsverfahren. Auffällig ist ebenfalls der große Abfall der Trainingsverlustkurve bei Epoche 45 und Epoche 89. Hier findet ein Wechsel im Kreuzvalidierungsverfahren statt. Die bessere Generalisierungsfähigkeit des Modells bei neuen Testdaten zeigt Ausschlag, indem die Klassifikationsergebnisse schlagartig besser werden und zu niedrigeren Kosten führen. Auch in Epoche 67 ist einer dieser Ausschläge zu sehen, der allerdings kleiner im Vergleich zu den anderen ausfällt. Zur Epoche 121 betrug das Ergebnis der Kostenfunktion 1.7. Es ergab eine *mAP* von 83.1%, leicht über den Referenzergebnissen von *SSD* zu *PascalVOC* (siehe Abbildung 3.6).

Das Training von *YOLO* ergab eine *mAP* von 80.36%, wobei nach bereits etwa 4500 Batches eine *mAP* von 80% erreicht wurde (siehe Abbildung 5.2). Da sich das Modell ab diesem Zeitpunkt nicht mehr maßgeblich verbessert hat, wurde das Training vor Erreichung der durch die Dokumentation nahegelegten maximalen Anzahl an Batches abgebrochen. Das frühe Erreichen eines sehr guten Ergebnis kann darauf zurückgeführt werden, dass es sich bei den gelabelten Klassen um Objekte des gleichen Typs handelt und das Modell dementsprechend leichter trainiert werden kann.

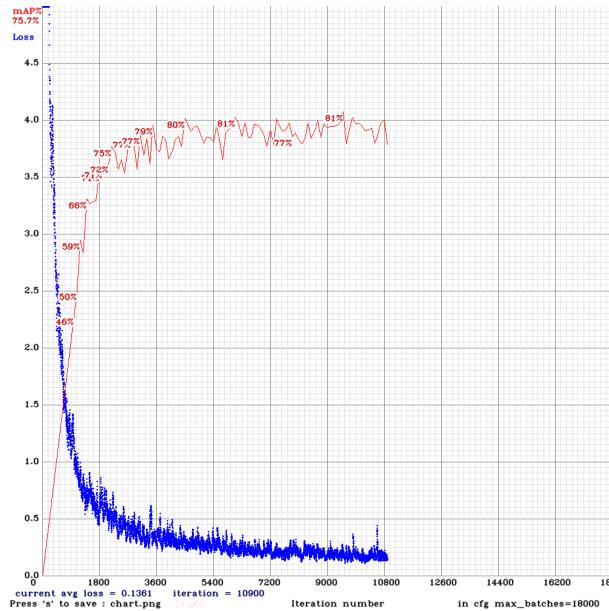


Abbildung 5.2.: Entwicklung der YOLO Testverlustkurve und der *mAP* im Training

Die Ergebnisse zu den einzelnen Klassen sind in folgender Tabelle dargestellt:

Klasse	mAP SSD	mAP YOLO
Saskia Wasser Groß	77.62%	76.69%
Saskia Wasser Klein	75.96%	80.63%
Pepsi Cola Groß	94.94%	73.14%
Pepsi Cola Klein	86.38%	75.24%
ISO	86.37%	90.28%
ACE	85.43%	86.69%
Stenger Johannisbeerschorle	69.47%	72.50%
Stenger Apfelsaftschorle	82.48 %	78.05%
Vitamalz Malzbier	76.24	81.94%%

Tabelle 5.1.: Validierungsergebnisse von SSD und YOLO

Werden nun die trainierten Modelle auf echte Daten angewendet, so fällt auf, dass manche Objekte doppelt detektiert werden. Um dieses Problem zu lösen, gibt es zwei Möglichkeiten.

Als erstes kann bei der Detektion der minimale *confidence score* angegeben werden, ab wann eine Detektion offiziell als solche wahrgenommen wird. Hier liegt die Herausforderung darin, einen optimalen Wert zu finden, sodass verdeckte Objekte noch als solche erkannt werden, aber doppelt erkannte Objekte nicht mehr auftreten. Der *confidence score* für *SSD* wurde nach mehrmaligem Iterieren auf 0.7 gesetzt. Bei *YOLO* ist der *confidence score* standardmäßig auf 0.25 festgelegt. Nach mehreren Durchläufen stellt sich heraus, dass dieser Wert nicht verändert werden sollte. Ein zu geringer Wert sorgt zwar dafür, dass möglicherweise mehr Objekte erkannt werden, allerdings steigt damit auch die Rate an falsch oder doppelt erkannter Objekte.

Die zweite Möglichkeit besteht darin, die maximale Überlappung zweier Bounding Boxen festzulegen. Somit werden doppelte Bounding Boxen, die sich flächenmäßig über einem gewissen Grenzwert überlappen, auf eine Bounding Box reduziert. Es stellte sich ein Wert von 0.45 beim *SSD* als geeignet heraus. Bei *YOLO* hingegen existiert dieser Parameter nicht. Außerdem wurden Inkonsistenzen im Detektionsverhalten festgestellt.

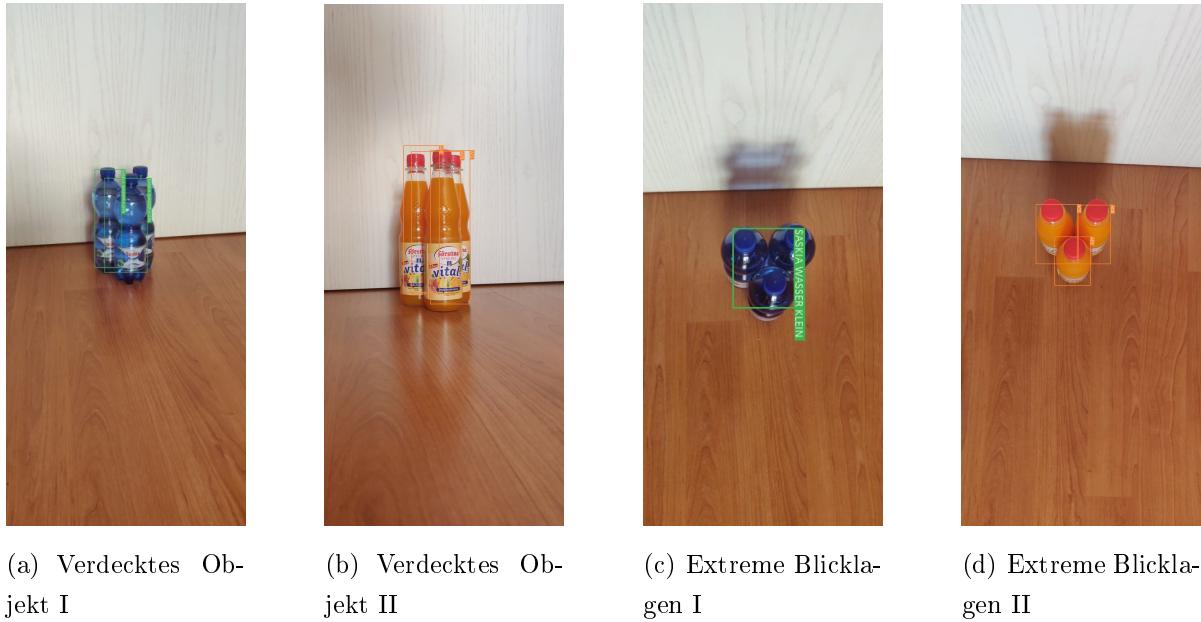


Abbildung 5.3.: Detektionsverhalten von SSD bei extremen Blicklagen

So sind von anderen Objekten verdeckte Objekte bei *SSD* nur schwer zu erkennen. So wird beispielsweise in Abbildung 5.3 (a) das rechte hintere Objekt selbst bei einem sehr niedrigen *confidence score* Schwellwert nicht erkannt. In Abbildung 5.3 (b) hingegen ist eine Detektion ab einem *confidence score* von 0.53 möglich, was allerdings wieder weitere ungenaue und ungewünschte Detektionen verursacht. Ähnliche Probleme ergeben sich für die Detektion von Objekten aus extremen Blicklagen. In Abbildung 5.3 (c) werden die Objekte beispielsweise erst ab einem *confidence score* von 0.54 erkannt, allerdings nicht als drei einzelne Objekte, sondern als ein großes Gesamtobjekt. Dies gilt allerdings nicht für alle Klassen. Bei Anwendung dieses Problemfalls auf eine andere Klasse ergeben sich bei vergleichbarem *confidence score* von 0.56 sehnswertere Ergebnisse (siehe Abbildung 5.3 (d)). Demnach könnte man darauf schließen, dass durch Ausbau des Datensatzes eine bessere Inferenz für solche Fälle ermöglicht werden könnte.

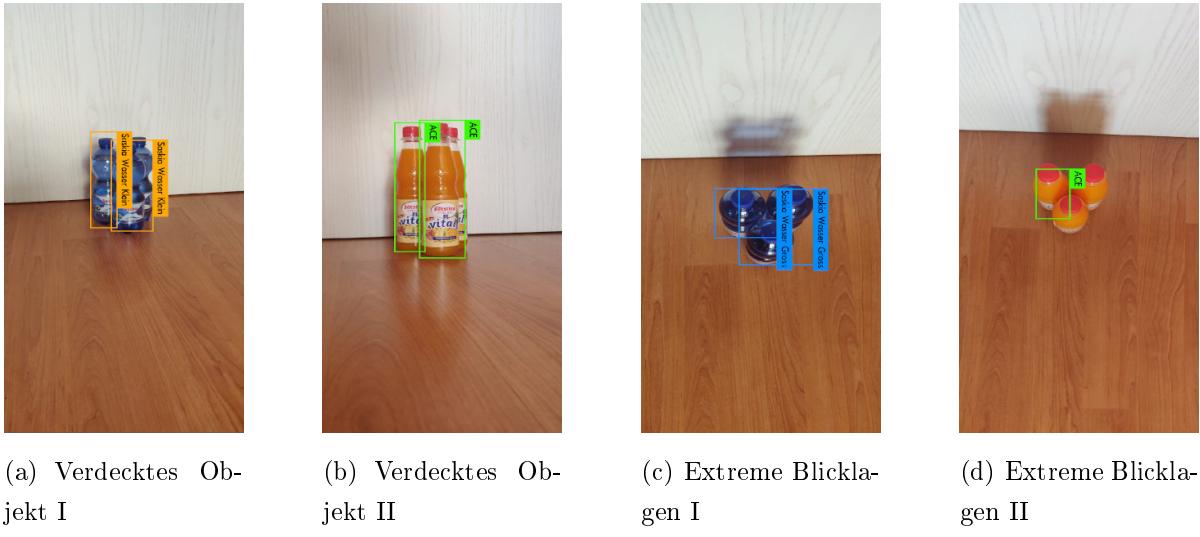


Abbildung 5.4.: Detektionsverhalten von YOLO bei extremen Blicklagen

Auch bei *YOLO* können verdeckte Objekte nur schwer erkannt werden. In den beiden Beispielen (Abbildung 5.4 (a) und (b)) wurden jeweils nur die beiden vorderen Flaschen detektiert, da der *confidence score* für die zwei stark verdeckten Flaschen bei nur 0.15 und 0.08 liegt. Bei einer extremen Blicklage fällt das Ergebnis noch schlechter als bei *SSD* aus. Hier werden im Beispiel von Abbildung 5.4 (c) beide gefundenen Flaschen mit einem *confidence score* von 0.97 und 0.53 sogar falsch klassifiziert, wobei wie bei *SSD* die gleichen zwei Flaschen mit einer Box umschlossen sind.

Die zwei genannten Problemfälle wurden im Datensatz zwar zu 12.5% abgedeckt, scheinen allerdings nur wenig Auswirkung auf besseres Detektionsverhalten für solche Fälle geliefert zu haben. Es lässt sich auch keine Aussage darüber treffen, ob eine Erweiterung des Datensatzes mit weiteren solchen Extrempfällen eine Abhilfe für dieses Problem hätte liefern können.



(a) 1 Meter



(b) 2 Meter



(c) 3 Meter

Abbildung 5.5.: Detektionsverhalten von SSD bei unterschiedlichen Entfernungen

Auch die Entfernung zum zu detektierenden Objekt besitzt eine Auswirkung auf das Detektionsverhalten. In Abbildung 5.5 wird gezeigt, dass ab einer Entfernung von drei Metern eine zuverlässige Detektion bei *SSD* nicht mehr möglich ist. In diesem Beispiel ist erst ab einer niedrigeren *confidence score* Grenze von 0.31 eine Detektion des Objekts wieder erfolgt, bei anderen Beispielen sogar erst ab einem Wert von 0.09.

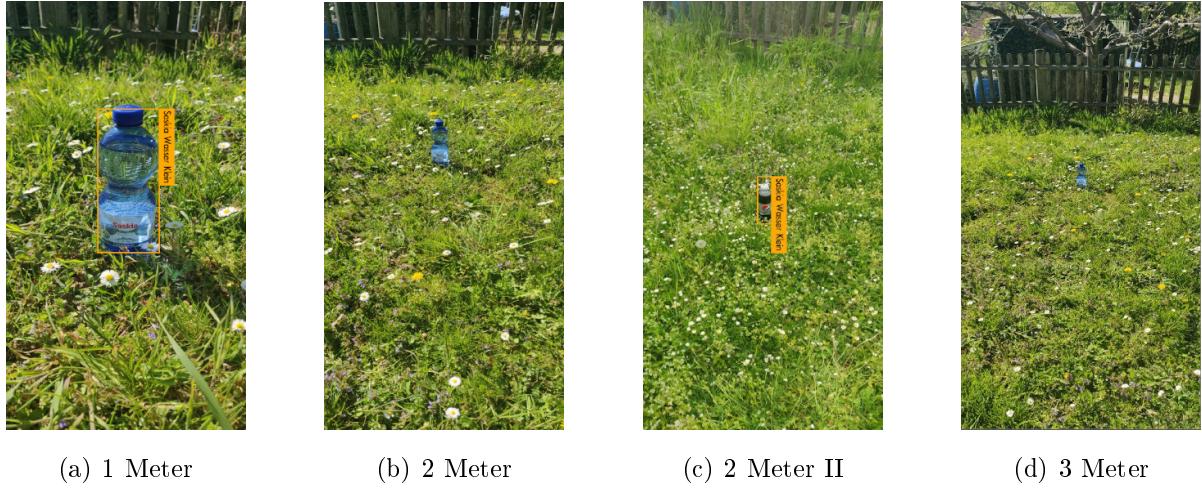


Abbildung 5.6.: Detektionsverhalten von YOLO bei unterschiedlichen Entfernungen

Bei größeren Entfernungen scheint *YOLO* anfälliger zu sein als *SSD*. Nach bereits zwei Metern Entfernung erfolgt keine zuverlässige Detektion mehr (siehe Abbildung 5.6 (b)). In dem Beispiel wird lediglich ein *confidence score* von 0.14 erzielt. Bei anderen Klassen werden Objekte sogar falsch klassifiziert (siehe Abbildung 5.6 (c)).

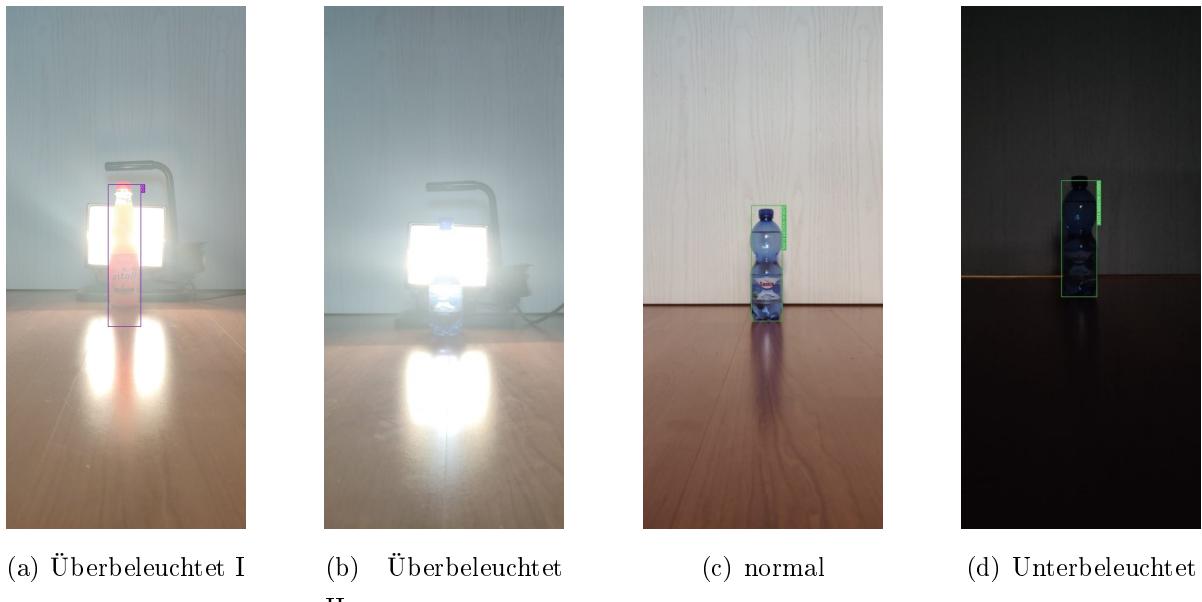


Abbildung 5.7.: Detektionsverhalten von SSD bei unterschiedlichen Beleuchtungsverhältnissen

Des Weiteren haben unterschiedliche Beleuchtungsgrade eine Auswirkung auf das Detektionsverhalten. In Abbildung 5.7 wird unterschieden zwischen dem Detektionsverhalten bei überbeleuchteten, normalen und unterbeleuchteten Sichtverhältnissen. Überbeleuchtete Umgebungsverhältnisse erschweren die Objektdetektion. Zum einen können bei *SSD* bestimmte Objekte gar nicht mehr erkannt werden (siehe Abbildung 5.7 (b)) oder die dadurch verursachte Lichtabsorption und anschließende Emission durch die zu detektierenden Objekte führt zu einer Falschdetektion. In Abbildung 5.7 (a) wird so beispielsweise mit einem *confidence score* von 0.91 das Objekt der Klasse *ACE* falsch als *ISO* klassifiziert.

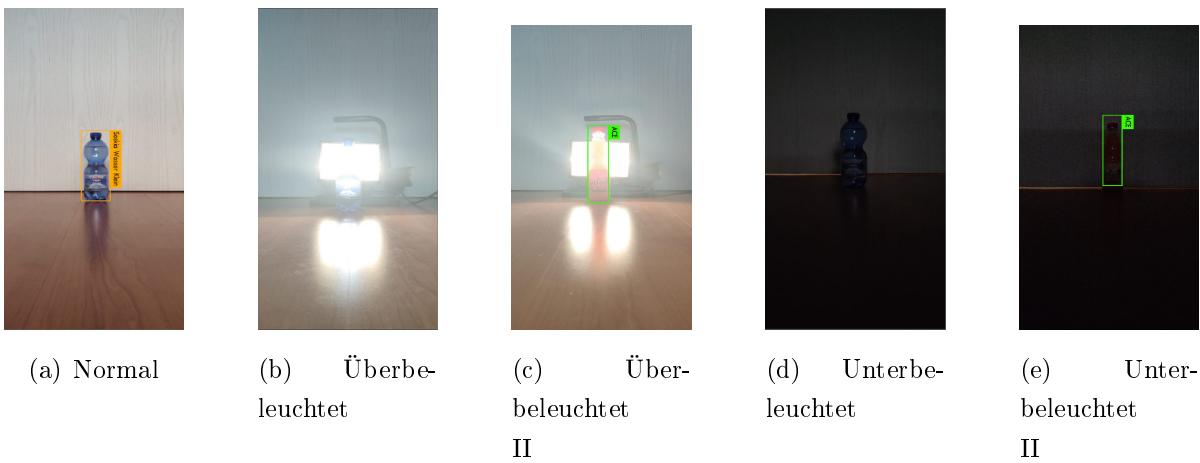


Abbildung 5.8.: Detektionsverhalten von YOLO bei unterschiedlichen Beleuchtungsverhältnissen

YOLO verhält sich bei den verschiedenen Beleuchtungsverhältnissen ähnlich zu *SSD*. Bei Abbildung 5.8 (b) und (c) sowie (d) und (e) fällt auf, dass die Detektion je nach Klasse unterschiedlich gut ausfällt. Im Beispiel wird eine Flasche der Klasse *ACE* mit einem *confidence score* von 0.61 in einer überbeleuchteten und mit 0.56 in einer unterbeleuchteten Umgebung detektiert, wohingegen die Klasse *Saskia Wasser Klein* in der jeweils gleichen Beleuchtungsumgebung bei nur 0.01 in überbeleuchteter und 0.06 in unterbeleuchteter Umgebung liegt.

Alle Detektionen beider Modelle reagierten allerdings invariant gegenüber unterschiedlichen Hintergründen oder Bildauflösungen.

5.2. Reaktionsvermögen

Bei der Inferenz fällt allerdings entgegen der Erwartungen auf, dass die Inferenz überdurchschnittlich langsam verläuft. Das Problem lässt sich auf die synchrone Arbeitsweise der bisherigen Detektionsalgorithmen zurück führen, bei dem erst ein neuer Frame des Video-Streams angefragt wird, sobald das aktuelle Bild durch die Vorverarbeitung gelaufen ist und durch das Modell inferiert wurde.

Um dem entgegen zu wirken, wurde ein Bufferkonzept in einem parallelen Thread realisiert, der einzelne Frames zeitgleich zur Inferenz anfragt und zwischenspeichert. Ist der Buffer voll, so werden nach dem *First In First Out* Verfahren die älteren Frames verworfen. Dadurch konnte die FPS Anzahl von *SSD* von maximal 18 auf die vollen 30 und bei *YOLO* von 20 auf 28 gesteigert werden. Dadurch sollten sämtliche Änderungen in der Umgebung rechtzeitig vom Objektdetektor erkannt werden.

Ein weiteres Problem beschreibt die initiale Latenz zwischen der Inferenz und der Bildaufnahme. Die Inferenz kann erst gestartet werden, sobald die Gewichtungen des Modells initialisiert und geladen wurden. Bei *YOLO* benötigt die Initialisierung etwa drei Sekunden, bei *SSD* vier Sekunden. Das lässt sich umgehen, indem entweder der Thread zur Bildaufnahme verzögern gestartet wird oder dessen Buffer kleiner gewählt wird.

5.3. Trainingsverhalten

Detektor	Hardware	Batchgröße	Dauer Epoche	Dauer Insgesamt
SSD	GTX 1080	16	9 Min.	16.5 Std.
YOLO	RTX 2060	64	1.5 Min.	8 Std.

Tabelle 5.2.: Trainingsverhalten von *SSD* und *YOLO*

Tabelle 5.2 zeigt die Ergebnisse des Trainingsverhaltens von *SSD* und *YOLO*. Pro Epoche wurden 998 Bilder durchlaufen. Die Dauer der beiden Trainingsdurchläufe lässt sich allerdings nur schwer vergleichen. Zum einen wird für das Training von *YOLO* eine andere GPU verwendet, deren Tensor Cores den Trainingsprozess merklich beschleunigen¹.

¹Nach einem bekannten Fork von *YOLO* kann durch Unterstützung von Tensor Cores eine doppelt so schnelle Trainingszeit erreicht werden [57]

Zum anderen werden die beiden Objektdetektoren in zwei verschiedenen Frameworks implementiert, wodurch ein anderes Trainingsverhalten von Grund auf gegeben ist und Unterschiede in der Performance auftreten können. So unterstützt das *Darknet* Framework beispielsweise die Nutzung der *cuDNN* Bibliothek für hardwarebeschleunigte Matrizenoperationen. Auch die Batchgröße wurde unterschiedlich gewählt, was sich auf die Häufigkeit des Gradientenabstiegs auswirkt. Bei beiden Verfahren kann das Training durch die Ablage von sogenannten Modell-Checkpoints zu einem späteren Zeitpunkt fortgeführt werden. Das ist vor allem dann von Vorteil, wenn nachträglich Parameter oder Trainingsdaten angepasst werden und kein kompletter Neustart des Trainings erforderlich ist.

6. Diskussion

6.1. Einsatz der Objektdetektoren in der Industrie

Die Bewertung der beiden Detektoren für den industriellen Einsatz baut auf den in Kapitel 3.3 eingeführten Bewertungskriterien auf.

Hinsichtlich der Präzision fallen beide Objektdetektoren besser als die ursprünglichen Referenzergebnisse aus den wissenschaftlichen Veröffentlichungen aus. Es ist aber zu bemerken, dass die Ergebnisse nicht gut mit denen der wissenschaftlichen Veröffentlichungen vergleichbar sind, da sie auf unterschiedlichen, deutlich einfacheren Daten trainiert wurden. Der Vergleichsdatensatz ist hierbei *PascalVOC 2007*, der 20 Klassen besitzt und damit für das Modell komplexer zu modellieren ist, als der *Smart Warehouse* Datensatz mit nur 9 Klassen. Mit 83,1% fällt *SSD300* um 8,8% besser aus, während *YOLOv3* sogar eine Verbesserung von 16.7% verzeichnet. Dennoch lässt sich darauf schließen, dass beide Implementierungen bezüglich Präzision in einer Größenordnung skalieren, die dem Niveau der Vergleichsergebnisse aus *PascalVOC 2007* nachkommt. Die leicht erhöhte Präzision von *SSD300* gegenüber *YOLOv3* liegt daran, dass *SSD300* im Gegensatz zu *YOLOv3* Bounding Box Vorschläge unterschiedlicher Seitenverhältnisse zulässt und die Unterteilung in Gitterstrukturen ebenfalls für mehrere Skalierungen durchgeführt wird. Dies wird vor allem erkenntlich beim Vergleich der *mAP* bei den Klassen *Pepsi Cola Groß* und *Pepsi Cola Klein*. Objekte der Klassen sehen grundsätzlich gleich aus, nur die Größe ändert sich, sind also unterschiedlich skaliert. Lässt man den Effekt bei unterschiedlichen Skalierungen allerdings außer Auge, so schneidet *YOLOv3* dennoch besser ab als *SSD300* auf Basis des *Smart Warehouse* Datensatzes (siehe Tabelle 5.1).

Ein weiteres Bewertungskriterium ist das Reaktionsvermögen. Hier schneidet *SSD300* mit durchschnittlich 30 FPS leicht besser ab als *YOLOv3* mit 28 FPS, wobei dieser Unterschied kaum als wahres Entscheidungskriterium gesehen werden sollte, *SSD300* *YOLOv3* vorzuziehen. Auf einem leichten Datensatz wie *Smart Warehouse* gibt es also kaum Unterschiede im Reaktionsvermögen als im Vergleich zu größeren und komplexeren Datensätzen wie *PascalVOC 2007* (siehe Abbildung 3.6). Laut wissenschaftlichen

Referenzergebnissen scheidet hier *SSD300* um 25 FPS besser ab als *YOLO*.

Bezüglich Trainingsverhalten könnte *SSD300* das Training mit leistungsfähigeren Grafikkarten wie der *NVIDIA Tesla V100* von 16,5 Stunden auf umgerechnet 5,2 Stunden geschätzt rund drei Mal schneller vollziehen. Die Rechnung legt die Tabelle 3.1 zugrunde. Pro Sekunde kann eine *NVIDIA GeForce GTX 1080* hierbei $2560 \cdot 9,784 = 25.047,04 \text{TeraFLOPs}$ vollziehen im Vergleich zu $5120 \cdot 14,13 = 72.345,6 \text{TeraFLOPs}$ pro Sekunde bei einer *NVIDIA Tesla V100*. Da die *NVIDIA Tesla V100* zusätzlich noch über Tensor Cores verfügt, kann der Zeitgewinn sogar noch höher eingeschätzt werden. Bei acht *NVIDIA Tesla V100* betragen die Trainingskosten so beispielsweise nur noch ca. 39 Minuten. Somit lässt sich festhalten, dass sich *SSD300* durchaus mit aktuellen Cloud Grafikkarten effizient trainieren lassen lässt. Da die Custom-Implementierung allerdings Hilfsstrukturen auf dem Sekundärspeicher erstellt, können hierbei beim Trainieren auf der Cloud Probleme auftreten. Meistens wird der Sekundärspeicher auf virtuellen Maschinen in der Cloud zum Schutz vor Angriffen nur *read-only* bereitgestellt.

YOLOv3 ließ sich mit acht Stunden deutlich schneller trainieren, allerdings unter vollkommen anderen Voraussetzungen. Nicht nur die verwendeten *Deep Learning* Frameworks sind unterschiedlich, sondern wie bereits erläutert aus Gründen der Arbeitsteilung ebenso die Trainingshardware. Als Grafikkarte wurde bei *YOLOv3* eine *NVIDIA GeForce RTX 2060* verwendet, die zusätzlich über 272 Tensor Cores verfügt und somit die Rechenleistung erhöht. Auch musste bei *YOLOv3* nicht die Batchgröße erniedrigt werden wie bei *SSD300*, um einem GPU Speicherüberlauf entgegen zu wirken. *YOLO* löst dieses Problem allgemein durch das *Subdivision* Konzept, welches bei *SSD* nicht existiert. Bei *SSD300* mussten somit durch die geringere Batchgröße zwangsmäßig mehr Gradientenabstiege vollzogen werden, was zwar auch auf die größere Stabilität hinweist, aber sich als Seiteneffekt auch auf die Trainingslänge auswirkt. Ein Training in der Cloud ist allerdings mit hoher Wahrscheinlichkeit immer noch schneller, z.B. auf Basis einer *NVIDIA Tesla V100*, deren technische Daten zur Rechenleistung rund doppelt so gut sind wie die zur *NVIDIA GeForce RTX 2060* (siehe Tabelle 3.1).

Zuletzt muss das Inferenzverhalten betrachtet werden. Hierzu kann folgende Tabelle 6.1 herangezogen werden:

Kriterium	SSD300	YOLOv3
Beleuchtung	0	0
Extreme Blicklagen	0	0
Entfernung	0	0
Verdeckung	-	-
Doppelte Erkennung	+	+
Hintergrund	+	+
Bildauflösung	+	+

Tabelle 6.1.: Inferenzverhalten SSD300 und YOLOv3

Hinsichtlich der Beleuchtung lässt sich nur eine Hypothese aufstellen, dass gerade sehr transparente Objekte durch die starke Überbeleuchtung nur schwer durch Mustererkennung detektierbar sind. Sowohl bei *SSD* als auch bei *YOLO* konnte so beispielsweise die Klasse *Saksia Wasser Klein* nicht detektiert werden. Welche Features letztendlich das neuronale Netz zur Detektion heranzieht, lässt sich nicht sagen und damit die Hypothese nicht beweisen. Während *SSD* im Falle der Überbeleuchtung zudem das Problem von Falschdetektionen aufweist, werden bei *YOLO* Objekte erst ab einer niedrigeren *confidence score* Grenze erkannt. Beim Test auf Unterbeleuchtung konnte *SSD300* hingegen Objekte besser erkennen als *YOLOv3*. Das Verhältnis im Detektionsvermögen bei variierenden Beleuchtungsverhältnissen ist somit unter beiden eher ausgeglichen. Die Ergebnisse stellen allerdings auch die Frage auf, ob bei Erweiterung des Datensatzes um weitere Bilder mit unterschiedlichen Beleuchtungsverhältnissen ein besseres Detektionsvermögen ermöglicht werden könnte.

Gleiches gilt bei extremen Blicklagen. Hier entsteht bei *YOLO* das Problem von Falschdetektionen, bei beiden Detektoren, dass mehrere Objekte als ein Objekt erkannt werden. Abbildung 5.3 (d) ist hingegen ein gutes Beispiel für die Machbarkeit der Detektion in extremen Blicklagen. Auch hier lässt sich vermuten, dass eine Erweiterung des Datensatzes Abhilfe leisten könnte.

Auch scheint die Detektion ab gewissen Entfernungen und Verdeckungsgraden nur erschwert möglich zu sein, bei *SSD* ab drei Metern, bei *YOLO* hingegen schon ab zwei Metern.

Eine Ausweitung des Trainingsdatensatzes erscheint demnach nicht abwegig. Im *Smart Warehouse* Datensatz sind die behandelten Fälle mit 12.5% nur unterbesetzt abgebildet. Neben einer Erweiterung des Datensatzes wäre ebenso das Anwenden von *Data Augmentation* oder *Transfer Learning* eine Alternative, mit der der Generalisierungsgrad des Modells gesteigert werden könnte. Das Problem der Detektion von verdeckten Objekten scheint aber bei beiden Objektdetektoren vorzuliegen. Auch hier könnte die Erweiterung des Datensatzes dazu dienen, dass die Modelle neue Merkmale gerade von teilweise verdeckten Objekten erlernen könnten. Ob das Erkennen von Teilmustern allerdings ausreicht, um einen genügend hohen *confidence score* zu entwickeln, bleibt offen. Das Problem doppelter Erkennung, wird von beiden allerdings gut gelöst, durch das Festlegen eines Schwellwertes im *confidence score*. Auch scheinen keine Merkmale aus Hintergrundinformationen zur Detektion herangezogen worden sein, da das Detektionsverhalten von *SSD300* als auch von *YOLOv3* invariant zu unterschiedlichen Hintergründen war. Auch die Bildauflösung hat keine Auswirkung auf das Detektionsvermögen.

Kriterium	SSD300	YOLOv3
Präzision	+	+
Reaktionsvermögen	+	+
Trainingsverhalten	0	+
Inferenzverhalten	0	0

Tabelle 6.2.: Gesamtbewertung SSD300 und YOLOv3

Insgesamt lässt sich schließen, dass sowohl *SSD300* als auch *YOLOv3* Objektdetektoren für den industriellen Einsatz darstellen. Die zuvor besprochenen Bewertungskriterien sind in Tabelle 6.2 abschließend zusammengefasst. Welcher der beiden Detektoren präferiert genutzt werden soll, lässt sich generell nicht aussagen. Beide Detektoren haben ihre Stärken in unterschiedlichen Anwendungsbereichen.

Falls viele Objekte unterschiedlicher Skalierung in der Detektionsumgebung erkannt werden sollen, so ist *SSD300* *YOLOv3* vorzuziehen. Falls dies allerdings nicht der Fall ist und bei einfachen Datensätzen wie *Smart Warehouse* mehr Wert auf Genauigkeit gelegt wird, so ist *YOLOv3* die bessere Wahl. Im Reaktionsvermögen sind beide bei einfachen Datensätzen circa gleich schnell, ein Unterschied lässt sich anscheinend erst bei komplexeren Datensätzen erkennen. Bezuglich des Trainingsverhaltens ist *YOLOv3* *SSD300*

vorzuziehen, nicht nur aufgrund der Möglichkeit *Subdivisions* zu definieren, sondern auch aufgrund der einfacheren Aufsetzbarkeit. Während *YOLO* die Menge an in den Grafikspeicher zu ladenden Bildern eines Batches durch die *Subdivisions* partitionieren kann, um einen Speicherüberlauf zu vermeiden, muss bei *SSD* hingegen immer die gesamte Batchgröße an Bildern in den GPU Speicher geladen werden. Andernfalls kann bei *SSD* nur die Batchgröße reduziert werden, was hinsichtlich des Gradientenverfahren womöglich nicht gewünscht sein kann. Außerdem ist es nicht zu vergessen, dass aufgrund der mangelnden Anpassbarkeit auf eigene Datensätze eine Custom-Implementierung von *SSD300* gewählt wurde und diese ebenso an vielen Stellen noch angepasst werden musste. Mit dem Inferenzverhalten als letztes Bewertungskriterium lässt sich aussagen, dass bei den behandelten Detektionssituationen des Öfteren bei beiden Detektoren die gleichen Probleme auftreten, die vermutlich, bis auf das Detektieren von verdeckten Objekten, durch eine Erweiterung des Datensatzes behoben werden könnten.

6.2. Machbarkeit des Smart Warehouse Szenarios

Nachdem sich in Kapitel 4 und 5 ausgiebig mit der Implementierung des *Smart Warehouse* Szenarios und der Darstellung der Ergebnisse bezüglich der Objektdetektoren beschäftigt wurde, ergibt sich nun ein klares Bild über die Machbarkeit des *Smart Warehouse* Szenarios. Es ergeben sich drei Problematiken, die nicht gelöst werden konnten:

- Das Inferieren des Video-Streams der Drohne, aufgrund der Inkompatibilität der Laufzeitumgebungen für den H.264 Decoder und der *Deep Learning* Modelle,
- Das Detektieren von verdeckten Objekten und
- Das eindeutige Zählen von Objekten während der Inventur.

Um wegen erste Problematik trotzdem eine abgewandelte Variante des *Smart Warehouse* Szenarios ermöglichen zu können, wurde anstelle eines Drohnen Video-Streams der Video-Stream einer Webcam verwendet. Aufgrund der oben ausgewerteten Ergebnisse wurde sich für eine Nutzung des *YOLOv3* Modells für das *Smart Warehouse* Szenario entschieden. Ein großer Aspekt bei der Wahl von *YOLOv3* stellt hierbei die Erweiterbarkeit des Szenarios dar. Momentan ist das Szenario allein auf Getränkeflaschen exemplarisch ausgelegt. Kommt man zur eigentlichen Idee zurück, in großen Warenhäusern mit

vielen verschiedenen Produkten Inventuren durchzuführen, so fällt der Aspekt des besseren Präzision der Detektion für gleiche Objekte unterschiedlicher Skalierung bei *SSD300* eher in den Hintergrund. Vielmehr ist es wichtig ein zuverlässiges Detektionsverhalten zu ermöglichen. Dies ist insbesondere dann wichtig, wenn Objekte durch andere Objekte verdeckt sind und trotzdem erkannt werden sollen. Die Flugsequenz der Drohne muss für solche Szenarien hingehend optimiert werden alternative Betrachtungswinkel in Erwägung ziehen. Erweitert man zusätzlich den Datensatz auf ein solch größeres Szenario, so könnten gerade solch klassische Probleme, also Objektdetektion bei größere Entfernung, Verdeckungsgraden und Blicklagen besser bewältigt werden. Da davon auszugehen ist, dass Warenhäuser eher gut belichtet sind, sollten extreme Beleuchtungseinflüsse ausgeschlossen werden können. Auch sind Objekte in Warenhäusern wohl kaum in Bewegung, weshalb auch die schnellere Inferenz von *SSD300* bei komplexeren Datensätzen vernachlässigt werden kann. Letztendlich ist es beim Ausbau eines solchen Szenarios ebenso wichtig, schnell das Detektions-Modell auszubauen, was auch für *YOLOv3* im Gegensatz zu *SSD300* spricht.

Mit der in den vorhergegangenen Kapiteln aufgezeigten Infrastruktur auf Basis einer Webcam anstelle einer Drohne und der Wahl von *YOLOv3*, lässt sich nun zunächst das *Smart Warehouse* Szenario einfach realisieren. Durch die Anbindung der Webcam an den *Flask* Server werden kontinuierlich einzelne Bilder an den Server zur Inferenz weitergeleitet (siehe Abbildung 6.1).



Abbildung 6.1.: Bildauszug aus Video-Stream nach der Inferenz mit YOLO

Problem bei der Inferenz stellt wie bereits erläutert nach wie vor die Detektion verdeckter Objekte dar. Beispielsweise sind Flaschen in Getränkekästen nur schwer detektierbar. Die Teilmerkmale verdeckter Objekte scheinen nicht ausreichend genug für eine zuverlässliche Detektion zu sein. Bei Herabsenken des *confidence scores* zur Detektion gerade solcher Objekte werden wiederum zuvor doppelt detektierte Objekte erneut doppelt erkannt. Es lässt sich hierbei kein gutes Gleichgewicht einstellen.

Die letzte offene Problematik beruht nicht auf technischer Natur, sondern auf der Auswahl des Verfahrens. Der Zählalgorithmus kann das selbe Objekt bei größeren zeitlichen Intervallen oder bei unterschiedlichen absoluten Positionen auf dem Bild nicht als das selbe Objekt identifizieren. Demnach werden Objekte in solchen Szenarien doppelt gezählt. Solange allerdings Objekte nur durch das Bild „gleiten“ und nicht erneut nach einem gewissen Intervall im Bild zu finden sind, lassen sich Objekte präzise zählen. Das Problem der doppelten Detektion lässt sich also nicht komplett mit dem Verfahren der Objektdetektion lösen, sondern nur auf konventionelle Verfahren mittels Scanning von Barcodes oder RFID-Chips.

Das *Smart Warehouse* Szenario ist somit nach den getroffenen Entscheidungen und mit den gegebenen Rahmenbedingungen nicht umsetzbar.

7. Zusammenfassung und Ausblick

Ziel der Arbeit war es zum einen zu evaluieren, wie gut die bestehenden Objektdetektoren für industrielle Anwendungsszenarien grundsätzlich geeignet sind, zum anderen, ob das spezifische Anwendungsszenario zur Durchführung einer Inventur von Warenhäusern mit einer Drohne prototypisch umsetzbar ist. In der Machbarkeitsstudie *Smart Warehouse* wurden schließlich folgende Ergebnisse erzielt:

- Erstellung eines exemplarischen Trainingsdatensatzes am Beispiel von Getränkeflaschen für die Detektion von Waren in einem Warenhaus,
- Architektur-Analyse und Vergleich gängiger Objektdetektoren nach ausgewählten Kriterien,
- Analyse und Vergleich von lokalen und cloudbasierten Trainingsinfrastrukturen nach ausgewählten Kriterien,
- Auswahl einer programmierbaren Drohne mit Video-Stream Anbindung,
- Trainieren zweier Objektdetektoren auf dem erstellten Trainingsdatensatz mit anschließendem Vergleich der entstandenen Modelle nach ausgewählten Kriterien,
- Programmierung einer Flugsequenz für eine Drohne sowie Decodierung des empfangenen Video-Streams der Drohne,
- Entwicklung einer Client-Server Anwendung mit serverseitigem Live-Stream von inferierten Bilddaten,
- Serverseitige Inferenz von Bilddaten einer Webcam in Echtzeit und
- Entwicklung eines Konzepts zur Durchführung einer kostengünstigen, zeitsparenden und ressourcenschonenden Inventur eines Warenhauses

Nach diesen Ergebnissen der Arbeit sind folgende Erkenntnisse festzuhalten:

Objektdetektoren wie *SSD* und *YOLO* weisen durchaus das Potential auf, industriell genutzt zu werden. Nicht nur hinsichtlich ihrer Präzision im Detektionsvermögen weisen sie zuversichtliche Ergebnisse auf, sondern auch in der Schnelligkeit ihrer Detektion. Beide haben bei speziellen Detektionsszenarien ihre Stärken und Schwächen. Gerade die Detektion von teilweise verdeckten Objekten stellt nach wie vor ein Problem dar, für das alternative Lösungswege gefunden werden müssen.

Daneben stellt sich die Machbarkeitsstudie *Smart Warehouse* basierend auf den gegebenen Rahmenbedingungen und den getroffenen Entscheidungen als nicht machbar heraus. Es wurde eine Drohne programmiert, die ein Miniaturmodell eines Warenhauses durchfliegt. Ihr Drohnenstream konnte zwar abgefragt werden, allerdings wegen Kompatibilitätsproblemen der Laufzeitumgebungen nicht durch ein *Deep Learning* Modell inferiert werden. Anstelle eines Video-Streams der Drohne wurde ein Video-Stream einer Webcam an einen Server zur Inferenz mit dem *YOLO* Objektdetektionsmodell gesendet und die daraus resultierenden Objekte gezählt. Das Ergebnis und der Live-Stream der Inferenz wurden auf einer Webapplikation dargestellt. Offen bleibt das Problem einer doppelten Detektion von Objekten bei mehrfachen Erscheinen im Bild nach gewissen zeitlichen Abständen, da keine eindeutige Identifizierung von Objekten mit reiner Objektdetektion möglich ist. Damit wäre das Zählen von gleichen Objekten an unterschiedlichen Lagerplätzen mit Hilfe eines Objektdetektors nicht umsetzbar, zumindest nicht ohne weitere Zuhilfenahme von zum Beispiel einem GPS Sensor. Auch das Detektieren von verdeckten Objekten bleibt eine weitere Problematik.

Um das *Smart Warehouse* Szenario weiter auszubauen, soll als nächstes der Datensatz vom simplen Beispiel von Getränkeflaschen auf eine echte Warenhausumgebung erweitert werden. Dadurch wird ebenfalls erhofft, die Präzision des Modells weiter zu verbessern und Inkonsistenzen bei unterschiedlichen Beleuchtungsverhältnissen, extremen Blickwinkeln, größeren Entfernung und unterschiedlichen Verdeckungsgraden zu beheben. Zudem muss die Flugsequenz der Drohne dahingehend optimiert werden, bei Stellen starker Verdeckungsgrade alternative Betrachtungswinkel zu wählen.

Als nächstes muss ein eigener H.264 Decoder implementiert werden, der die neuste Python Version unterstützt und im Einklang mit den verwendeten *Deep Learning* Frameworks steht. Nach diesem Schritt könnten anschließend auch Live-Bilder des Video-Streams der Drohne inferiert werden.

Um abschließend spezielle Rahmenbedingungen bei der Zählung durch Objektdetektion auszuschließen, könnte das Szenario ebenso zur eindeutigen Identifizierung von Objekten mit konventionellen RFID Chips erweitert werden. Dies ermöglicht ebenso neue Szenarien zu entwickeln, in denen Objektdetektion nur Suche von Objekten einer bestimmten Klasse genutzt wird und deren Identität anschließend mit RFID Chips sichergestellt wird.

Bei Berücksichtigung dieser Verbesserungsvorschläge, lässt sich somit durchaus auf eine Umsetzbarkeit des *Smart Warehouse* Szenarios in der Zukunft hoffen.

Literaturverzeichnis

- [1] Aurélien Géron. *Machine Learning mit Scikit-Learn & TensorFlow: Konzepte, Tools und Techniken für intelligente Systeme: Übersetzung von Kristian Rother.* 1. Aufl. Heidelberg: dpunkt.verlag GmbH, 2018.
- [2] Tao Wang, David J. Wu, Adam Coates, Andrew Y. Ng. „End-to-End Text Recognition with Convolutional Neural Networks“. Diss. Stanford Artificial Intelligence Laboratory, 2012-07-16. URL: <https://ai.stanford.edu/~ang/papers/ICPR12-TextRecognitionConvNeuralNets.pdf> (Einsichtnahme: 18.05.2020).
- [3] dok.s. innovation. *inventAIRyX.* dok.s. innovation Homepage, 2019. URL: <https://www.doks-innovation.com/inventory-x/> (Einsichtnahme: 26.10.2019).
- [4] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. „ImageNet Classification with Deep Convolutional Neural Networks“. Diss. Neural Information Processing Systems Conference, 2012. URL: <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf> (Einsichtnahme: 18.05.2020).
- [5] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik. „Rich feature hierarchies for accurate object detection and semantic segmentation: Tech report (v5)“. Diss. UC Berkeley, 2014-10-22. URL: <https://arxiv.org/pdf/1311.2524.pdf> (Einsichtnahme: 18.05.2020).
- [6] Farhana Sultana, Abu Sufian, Paramartha Dutta. „Evolution of Image Segmentation using Deep Convolutional Neural Network: A Survey“. Diss. University of Gour Banga, 2020. URL: <https://arxiv.org/pdf/2001.04074.pdf> (Einsichtnahme: 18.05.2020).
- [7] Priya Dwivedi. *Semantic Segmentation — Popular Architectures.* Towards Data Science, 2019. URL: <https://towardsdatascience.com/semantic-segmentation-popular-architectures-dff0a75f39d0> (Einsichtnahme: 07.03.2020).
- [8] Xavier Glorot, Y. B. „Understanding the difficulty of training deep feedforward neural networks“. Diss. Montréal: Universite de Montréal, 2010. URL: <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf> (Einsichtnahme: 26.10.2019).

- [9] Imad Dabbura. *Gradient Descent Algorithm and Its Variants*. Towards Data Science, 2017. URL: <https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3> (Einsichtnahme: 26. 10. 2019).
- [10] Danqing Liu. *A Practical Guide to ReLU*. Medium, 2017. URL: <https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7> (Einsichtnahme: 26. 10. 2019).
- [11] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Chang-Yang Fu, Alexander C. Berg. „SSD: Single Shot MultiBox Detector“. Diss. University of North Carolina at Chapel Hill, University of Michigan, Ann Arbor, 2016. URL: <https://arxiv.org/pdf/1512.02325.pdf> (Einsichtnahme: 02. 11. 2019).
- [12] Mark Everingham, J. W. *The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Development Kit*. 2007. URL: <http://host.robots.ox.ac.uk/pascal/VOC/voc2007/html/doc/voc.html#SECTION00030000000000000000> (Einsichtnahme: 08. 02. 2020).
- [13] Mark Everingham, J. W. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Development Kit*. 2012. URL: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/html/doc/devkit_doc.html#SECTION00030000000000000000 (Einsichtnahme: 08. 02. 2020).
- [14] Paul Henderson, V. F. „End-to-end training of object class detectors for mean average precision“. Diss. University of Edinburgh, 2017. URL: <https://arxiv.org/pdf/1607.03476.pdf> (Einsichtnahme: 18. 05. 2020).
- [15] Adrian Rosebrock. *Intersection over Union (IoU) for object detection*. pyimagesearch, 2016. URL: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/> (Einsichtnahme: 02. 11. 2019).
- [16] Dingfu Zhou, Jin Fang, Xibin Song, Chenye Guan, Junbo Yin, Yuchao Dai, Rui-gang Yang. „IoU Loss for 2D/3D Object Detection“. Diss. Northwestern Polytechnical University, Xi'an, China, 2019. URL: <https://arxiv.org/pdf/1908.03851.pdf> (Einsichtnahme: 18. 05. 2020).
- [17] Jonathan Hui. *mAP (mean Average Precision) for Object Detection*. Medium, 2018. URL: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173 (Einsichtnahme: 18. 05. 2020).

- [18] Rohith Gandhi. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms: Understanding object detection algorithms.* Towards Data Science, 2018. URL: <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e> (Einsichtnahme: 18. 05. 2020).
- [19] Ross Girshick. „Fast R-CNN“. Diss. 2015. URL: <https://arxiv.org/pdf/1504.08083.pdf> (Einsichtnahme: 18. 05. 2020).
- [20] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun. „Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks“. Diss. 2016. URL: <https://arxiv.org/pdf/1506.01497.pdf> (Einsichtnahme: 18. 05. 2020).
- [21] Kaiming He, Georgia Gkioxari, Piotr Dollar, Ross Girshick. „Mask R-CNN“. Diss. 2018. URL: <https://arxiv.org/pdf/1703.06870.pdf> (Einsichtnahme: 12. 04. 2020).
- [22] Andrew Ng. *Anchor Boxes*. Coursera, 2019. URL: <https://www.coursera.org/lecture/convolutional-neural-networks/anchor-boxes-yNwO0> (Einsichtnahme: 02. 11. 2019).
- [23] Karen Simonyan, A. Z. „Very Deep Convolutional Networks For Large-Scale Image Recognition“. Diss. University of Oxford, 2015. URL: <https://arxiv.org/pdf/1409.1556.pdf> (Einsichtnahme: 18. 05. 2020).
- [24] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, Ali Farhadi. „You Only Look Once: Unified, Real-Time Object Detection“. Diss. University of Washington, 2016. URL: <https://arxiv.org/pdf/1506.02640.pdf> (Einsichtnahme: 10. 11. 2019).
- [25] Joseph Redmon, A. F. „YOLOv3: An Incremental Improvement“. Diss. University of Washington, 2018. URL: <https://arxiv.org/pdf/1804.02767.pdf> (Einsichtnahme: 04. 02. 2020).
- [26] Mark Everingham, S. M. Ali Eslami, · Luc Van Gool, Christopher K. I. Williams, John Winn, Andrew Zisserman. „The PASCAL Visual Object Classes Challenge: A Retrospective“. Diss. University of Leeds, Microsoft Research, ETH, Zurich, University of Edinburgh, University of Oxford, KU Leuven, 2014. URL: <http://host.robots.ox.ac.uk/pascal/VOC/pubs/everingham15.pdf> (Einsichtnahme: 18. 05. 2020).

- [27] Arun Ponnusamy. *Preparing Custom Dataset for Training YOLO Object Detector*. Arun Ponnusamy Homepage, 2019. URL: <https://www.arunponnusamy.com/preparing-custom-dataset-for-training-yolo-object-detector.html> (Einsichtnahme: 02.02.2020).
- [28] NVIDIA. *TRAIN MODELS FASTER*. NVIDIA Homepage, 2020. URL: <https://developer.nvidia.com/cuda-zone> (Einsichtnahme: 09.02.2020).
- [29] PyTorch. *GET STARTED*. PyTorch Homepage, 2020. URL: <https://pytorch.org/get-started/locally/> (Einsichtnahme: 09.02.2020).
- [30] NVIDIA. *NVIDIA cuDNN*. NVIDIA Homepage, 2020. URL: <https://developer.nvidia.com/cudnn> (Einsichtnahme: 20.05.2020).
- [31] Google Cloud. *Cloud TPU*. Google Cloud Homepage, 2020. URL: <https://cloud.google.com/tpu/?hl=de> (Einsichtnahme: 09.02.2020).
- [32] NVIDIA. *NVIDIA TENSOR CORES*. NVIDIA Homepage, 2020. URL: <https://www.nvidia.com/en-us/data-center/tensor-cores/> (Einsichtnahme: 24.05.2020).
- [33] Karl Freund. *Microsoft: FPGA Wins Versus Google TPUs For AI*. Forbes, 2017. URL: <https://www.forbes.com/sites/moorinsights/2017/08/28/microsoft-fpga-wins-versus-google-tpus-for-ai/#781e2bf53904> (Einsichtnahme: 09.02.2020).
- [34] Amazon Web Services. *Amazon SageMaker*. AWS Homepage, 2020. URL: <https://aws.amazon.com/de/sagemaker/> (Einsichtnahme: 12.03.2020).
- [35] Amazon Web Services. *Amazon SageMaker-ML-Instance-Typen*. AWS Homepage, 2020. URL: <https://aws.amazon.com/de/sagemaker/pricing/instance-types/> (Einsichtnahme: 14.03.2020).
- [36] Google Cloud Platform. *Produkte für künstliche Intelligenz und maschinelles Lernen: AI Platform*. GCP Homepage, 2020. URL: <https://cloud.google.com/products/ai> (Einsichtnahme: 14.03.2020).
- [37] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson. „In-Datacenter Performance Analysis of a Tensor Processing Unit“. Diss. 2017. URL: <https://arxiv.org/ftp/arxiv/papers/1704/1704.04760.pdf> (Einsichtnahme: 18.05.2020).
- [38] Google Cloud. *Systemarchitektur*. Google Cloud Dokumentation, 2020. URL: <https://cloud.google.com/tpu/docs/system-architecture> (Einsichtnahme: 09.02.2020).

- [39] Google Colaboratory. *Welcome to Colaboratory!* Google Colaboratory Homepage, 2020. URL: <https://colab.research.google.com/notebooks/welcome.ipynb> (Einsichtnahme: 14.03.2020).
- [40] Google Cloud. *Colaboratory-Notebooks*. Google Cloud Dokumentation, 2020. URL: <https://cloud.google.com/automl-tables/docs/notebooks> (Einsichtnahme: 14.03.2020).
- [41] Microsoft Azure. *Azure Machine: Machine-Learning-Dienst für Unternehmen zur schnelleren Erstellung und Bereitstellung von Modellen*. Microsoft Azure Homepage, 2020. URL: <https://azure.microsoft.com/de-de/services/machine-learning/> (Einsichtnahme: 14.03.2020).
- [42] FloydHub. *Home*. FloydHub Documentation, 2020. URL: <https://docs.floydhub.com/> (Einsichtnahme: 15.02.2020).
- [43] EASA. *EU wide rules on drones published*. EASA Homepage, 2019. URL: <https://www.easa.europa.eu/newsroom-and-events/press-releases/eu-wide-rules-drones-published> (Einsichtnahme: 28.03.2020).
- [44] Drohnen.de. *Vorschriften, Genehmigungen für die Nutzung von Drohnen und Multicoptern*. Drohnen.de Homepage, 2020. URL: <https://www.drohnen.de/vorschriften-genehmigungen-fuer-die-nutzung-von-drohnen-und-multicoptern/> (Einsichtnahme: 28.03.2020).
- [45] RyzeRobotics. *Tello EDU*. RyzeRobotics Homepage, 2020. URL: <https://www.ryzerobotics.com/de/tello-edu?from=store-product-page> (Einsichtnahme: 28.03.2020).
- [46] Parrot. *PARROT BEBOP 2*. Parrot Homepage, 2020. URL: <https://www.parrot.com/de/drohnen/parrot-bebop-2> (Einsichtnahme: 20.05.2020).
- [47] Felix Hausberger, R. K. *Datasets for SmartWarehouse*. Kaggle Homepage, 2020. URL: <https://www.kaggle.com/fidsusj/datasets> (Einsichtnahme: 03.05.2020).
- [48] Intan Purnamasar. *Vergleich von Algorithmen zur Objekterkennung für die Anwendung in Abbildungen komplexer Energiesystem*. RWTH Aachen University, 2018. URL: https://www.matse.itc.rwth-aachen.de/dienste/public/show_document.php?id=18753 (Einsichtnahme: 14.03.2020).
- [49] Amazon Web Services. *Kostenloses Kontingent für AWS*. AWS Homepage, 2020. URL: <https://aws.amazon.com/de/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc> (Einsichtnahme: 14.03.2020).

- [50] Google Cloud Platform. *Kostenlose Stufe der Google Cloud Platform*. GCP Homepage, 2020. URL: <https://cloud.google.com/free> (Einsichtnahme: 14.03.2020).
- [51] Microsoft Azure. *Create your Azure free account today: Get started with 12 months of free services*. Microsoft Azure Homepage, 2020. URL: <https://azure.microsoft.com/en-us/free/> (Einsichtnahme: 14.03.2020).
- [52] Microsoft Azure. *Virtuelle Windows-Computer – Preise*. Microsoft Azure Homepage, 2020. URL: <https://azure.microsoft.com/de-de/pricing/details/virtual-machines/windows/> (Einsichtnahme: 12.03.2020).
- [53] FloydHub. *Plans*. FloydHub Documentation, 2020. URL: <https://docs.floydhub.com/faqs/plans/#what-is-in-the-trial-plan> (Einsichtnahme: 25.01.2020).
- [54] TechPowerUp. *GPU Specs Database*. TechPowerUp Homepage, 2020. URL: <https://www.techpowerup.com/gpu-specs/> (Einsichtnahme: 09.02.2020).
- [55] Ryze Tech. *TELLO SDK 2.0 User Guide*. Ryze Tech Homepage, 2018. URL: <https://dlcdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf> (Einsichtnahme: 24.04.2020).
- [56] Microsoft Corporation, Hrsg. *CC++ DLLs in Visual Studio*. 2020. URL: <https://docs.microsoft.com/de-de/cpp/build/dlls-in-visual-cpp?view=vs-2019> (Einsichtnahme: 11.04.2020).
- [57] AlexeyAB. *Issue #704*. GitHub Darknet Repository, 2018. URL: <https://github.com/pjreddie/darknet/issues/704> (Einsichtnahme: 24.05.2020).
- [58] Wikipedia. *Deep Learning*. Wikipedia, 2019. URL: https://de.wikipedia.org/wiki/Deep_Learning (Einsichtnahme: 27.01.2019).
- [59] David E. Rumelhart/ Geoffrey E. Hinton/ Ronald J. Williams. *Learning Internal Representations by Error Propagation*. Hrsg. von University of California, San Diego. 09/1985. URL: <https://apps.dtic.mil/dtic/tr/fulltext/u2/a164453.pdf> (Einsichtnahme: 26.10.2019).
- [60] Daphne Cornelisse. *An intuitive guide to Convolutional Neural Networks*. freeCodeCamp Homepage, 2018. URL: <https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050> (Einsichtnahme: 26.10.2019).

- [61] Abhineet Saxena. *Convolutional Neural Networks (CNNs): An Illustrated Explanation*. XRDS Crossroads - The ACM Magazine for Students, 2016. URL: <https://blog.xrds.acm.org/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/> (Einsichtnahme: 26. 10. 2019).
- [62] Leonadro Araujo Santos. *Pooling Layer: Introduction*. GitBook, 2018. URL: https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/pooling_layer.html (Einsichtnahme: 26. 10. 2019).

A. Anhang

Das Perzepron

Der Aufbau eines typischen Perzeprons besteht aus einer oder mehreren Schichten so genannter *Linear Threshold Units* (LTU) wie in Abbildung A.1 dargestellt.

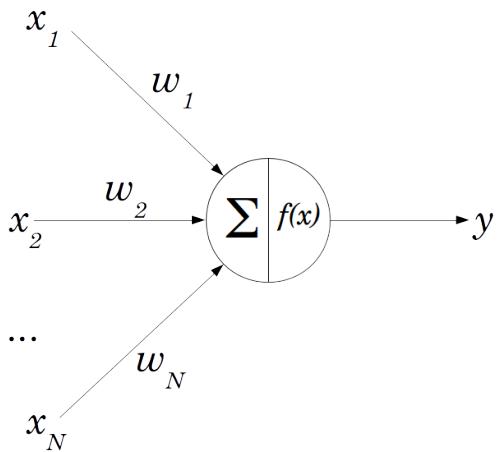


Abbildung A.1.: Linear Threshold Unit [1]

Es besteht aus n Eingängen mit $x_i \in \mathbb{Q}$, die im Inputvektor \mathbf{x} zusammengefasst werden. Jeder Eingang wird mit einem Gewicht w_i aus dem Gewichtsvektor \mathbf{w} versehen. Die LTU berechnet das Skalarprodukt $\mathbf{w}^T \circ \mathbf{x}$ aller Eingänge \mathbf{x} mit ihren Gewichten \mathbf{w} und wendet anschließend auf das Ergebnis z eine Aktivierungsfunktion an. Das Ergebnis $h_w(\mathbf{x})$ kann anschließend als Eingabe für ein weiteres Perzepron dienen.

Die einfachste Aktivierungsfunktion für ANNs ist die *Heaviside-Funktion* [1]:

$$h_w(x) = s(\mathbf{w}^T \circ \mathbf{x}) = s(z) =$$

$$\begin{pmatrix} w_1 & w_2 & \dots & w_n \end{pmatrix} \circ \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{cases} 1 & \text{wenn } z \geq 0 \\ 0 & \text{wenn } z < 0. \end{cases} \quad (\text{A.1})$$

Falls eine Klassifizierung mit Wahrscheinlichkeiten vorliegen soll, so ist die letzte Schicht eines Perzeptrons meist mit der *Softmax-Funktion*

$$h_w(x) = \sigma(z)_j = \frac{e^{z_j}}{\sum_{i=0}^n e^{z_i}}$$

implementiert, die den Wert des j -ten LTUs einer Schicht mit allen anderen n Werten der LTUs derselben Schicht ins Verhältnis setzt [1]. Es gibt eine Vielzahl an möglichen Aktivierungsfunktionen, die im Kapitel *Hyperparameter* betrachtet werden.

Die Aktivierung einer LTU hängt zusätzlich von einem Schwellwert θ ab, der durch einen sogenannten *Bias* festgelegt wird. Dies ist die Gewichtung des letzten Eingangs, der standardmäßig den Wert 1 liefert. Wird die Gewichtung negativ gewählt, so ist es schwieriger die LTU zu aktivieren, während eine positive Gewichtung die Aktivierung vereinfacht [1].

Nun bilden ein oder mehrere Schichten solcher LTUs ein Perzepron. Jede einzelne LTU ist dabei mit allen LTUs der vorherigen Schicht verbunden (siehe Abbildung A.2). Hier wird auch von sogenannten vollständig verbundenen Schichten (engl.: *Fully-Connected Layer*) gesprochen. Die beiden LTUs zur Ausgabe können dabei Aussagen über eine Klassifikation von Daten anhand der Eingangsdaten treffen, während die LTUs im Input Layer wesentlich Daten weiter reichen. Die Verbindungen zur ersten Schicht des Hidden Layer sind stets mit Eins belegt. Existiert keine verborgene Schicht, so wird das ANN als einschichtiges Perzepron bezeichnet, ab einer oder mehr verborgenen Schichten wird bereits von einem *Multi-Layer Perzepron* (MLP), einem mehrschichtigen Perzepron, gesprochen. Ist das neuronale Netz optimal trainiert, so ist am Ende nur eines der LTUs zur Ausgabe aktiviert. Das folgende ANN ist zudem ein Beispiel für ein sogenanntes

Feed-Forward Network, bei dem die Auswertung der Daten von einer Schicht zur nächsten weitergereicht wird, ohne zu bereits besuchten Schichten zurückzukehren [1].

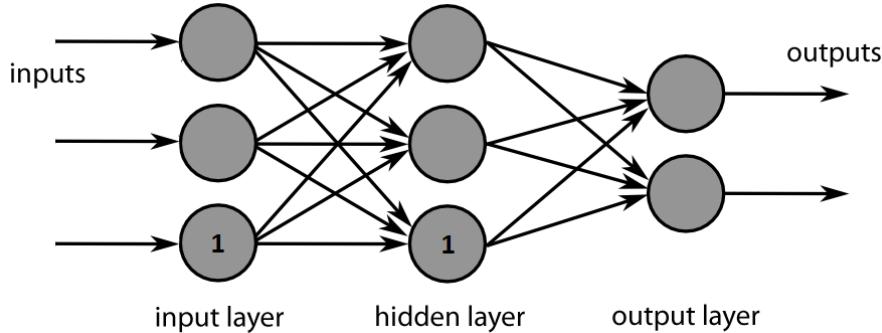


Abbildung A.2.: Das einschichtige Perzeptron [58]

Lernmethoden

Im Wesentlichen existieren vier Methoden, mit deren Hilfe neuronale Netze trainiert werden können. Beim *überwachten Lernen* werden den Trainingsdaten Lösungen, sogenannte *Labels*, hinzugefügt. Klassifikationsprobleme stellen eine typische Problemstellung für überwachte Lernverfahren dar. Auch die später eingeführten Objektdetektoren werden mittels überwachtem Lernen trainiert [1].

Beim *unüberwachten Lernen* werden den Trainingsdaten keinerlei Lösungen hinzugefügt, der Algorithmus muss selbstständig Klassifikationsaussagen treffen können. Ein Beispiel hierzu wäre *K-Means Clustering* [1].

Deep Belief Networks (DBNs) bestehend aus einzelnen *Restricted Boltzmann Machines* (RBMs) werden zunächst unüberwacht trainiert, bevor im *Fine Tuning* das Gesamtnetzwerk mit überwachten Lerntechniken fertiggestellt wird. Es wird hier von *halbüberwachtem Lernen* gesprochen.

Als letztes ist das sogenannte *Reinforcement Learning* zu nennen. Hierbei wird ein neuronales Netz durch das Erteilen von Belohnungen bzw. Bestrafungen so konditioniert, dass es zukünftig basierend auf der wahrgenommenen Umwelt selbstsicher richtige Aktionen auswählen kann.

Gradientenverfahren und Backpropagation

Um zu verstehen, wie ein neuronales Netz durch überwachtes Lernen „lernt“, muss zunächst der Begriff der Kostenfunktion (engl.: *cost function*) eingeführt werden. Die Kostenfunktion ist ein Qualitätsmaß dafür, wie weit die Ausgabe einer LTU vom erwarteten Wert abweicht [1]. Angenommen dem neuronalen Netz wird ein Datensatz zur Klassifikation übergeben, so ist am Ende meist nicht nur eine LTU zu Ausgabe aktiviert, was auf eine eindeutige Klassifikation schließen würde, sondern meist mehrere zu einem frühen Stadium des neuronalen Netzes.

Eine oft genutzte Kostenfunktion ist die *Root Mean Squared Error Funktion* (RMSE):

$$E(\mathbf{z}, \mathbf{o}) = \sqrt{\frac{1}{n} \sum_{k=0}^n (z_k - o_k)^2}. \quad (\text{A.2})$$

Eine weitere Kostenfunktion für das überwachte Lernen ist die *Smooth L1* Funktion:

$$SM_{L1}(\mathbf{z}, \mathbf{o}) = \begin{cases} \sum_{k=0}^n 0.5(z_k - o_k)^2 & \text{wenn } |x| < 1 \\ \sum_{k=0}^n |z_k - o_k| - 0.5 & \text{sonst} \end{cases}$$

Bei beiden ist z der erwartete Ausgabevektor des Perzeptrons, während o die momentane Ausgabe darstellt. Den Fehler der Abweichung dieser beiden Werte gilt es nun schrittweise zu minimieren. Um dies zu erreichen können die drei Parameter

1. Gewichtung der Verbindungen zum Perzepron
2. Bias zur Aktivierung der LTUs des Perzepron und
3. Stärke der Aktivierung des vorherigen Perzepron

angepasst werden [1].

Hierbei wird das sogenannte *Gradientenverfahren* eingesetzt. Es berechnet in einem iterativen Prozess über mehrere Testdaten das globale Minimum der Kostenfunktion nach den Gewichtungen der Verbindungen und damit auch nach den Bias Werten, die natürlich ebenso Gewichtungen darstellen. Ergebnis eines Durchlaufs im Gradientenverfahren

(siehe Formel A.3) ist die Gewichtungsmatrix, die die Änderung der Gewichtung jeder einzelnen Verbindung eines Perzeptrons zu jeder LTU des Folgeperzeptrons angibt [1]:

$$w_{ijt} = w_{ijt-1} - \eta \frac{\partial E}{\partial w_{ij}}. \quad (\text{A.3})$$

Das Gradientenverfahren eignet sich allerdings nur für stetig differenzierbare Funktionen ohne Plateaus. Somit können beispielsweise bei der Heaviside-Funktion als Aktivierungsfunktion Probleme auftreten, da eine Ableitung der Kostenfunktion stets Null betragen würde, wohingegen bei anderen Aktivierungsfunktionen wie beispielsweise der Sigmoid-Funktion mit

$$\text{sig}(x) = \frac{1}{1 + e^{-x}} \quad (\text{A.4})$$

im gesamten Definitionsbereich immer kleine Änderungen der Gewichtungen zu verzeichnen wären [1].

Nun stellt sich auch der Vorteil von MSE als Kostenfunktion gegenüber anderen, durchaus komplexeren Kostenfunktionen heraus. Während MSE genau ein Minimum, das zugleich das globale Minimum der Funktion darstellt, besitzt, haben andere Kostenfunktionen im Gradientenverfahren das Problem, dass anstelle des globalen Minimums auch nur lokale Minima erreicht werden können. Dies hat zur Folge, dass mehrere iterative Durchläufe mit mehreren Testdatensätzen nötig werden, um durch unterschiedliche Startkonfigurationen die unterschiedlichen Minima miteinander vergleichen zu können und damit das globale Minimum herauszustellen [1].

Durch das Gradientenverfahren werden somit nur diejenigen Verbindungen verstärkt, die zum richtigen Ergebnis führen.

Nun bleibt nur noch die dritte Möglichkeit zur Minimierung der Kostenfunktion übrig, die Anpassung der Stärke der Aktivierung des vorherigen Perzeptrons. Zu diesem Problem veröffentlichten David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams 1985 den sogenannten *Backpropagation-Algorithmus* [59]. Dieser berechnet mit Hilfe des Gradientenverfahrens welchen Anteil am Fehler der Ausgabe jede LTU des letzten Perze-

trons hat und anschließend welcher Anteil davon wiederum auf das vorherige Perzeptron der vergorenen Schicht zurück zu führen ist. Das Gradientenverfahren wird solange wiederholt, bis die Eingangsschicht erreicht wurde, es berechnet also für jede LTU deren Anteil am Fehler des Ergebnisses [1].

Mit Hilfe des Gradientenverfahren im Backpropagation Algorithmus wird nun also das neuronale Netz durch mehrere iterative Durchläufe trainiert, wobei das Training als Anpassung der Gewichtungen einzelner Verbindungen zu verstehen ist.

Convolutional Neural Networks

Ein CNN besteht größtenteils aus drei grundlegenden Bausteinen, den sogenannten *Convolutional Layern*, *Pooling Layern* und den bereits bekannten *Fully-Connected Layern*.

Ein *Convolutional Layer* zeichnen sich unter anderem dadurch aus, dass jede LTU dieser Schicht nicht mit allen vorherigen LTUs der vorgegangenen Schicht verbunden ist, sondern nur mit einer festen, beschränkten Anzahl. Es ist also kein vollständig verbundenes neuronales Netz. Dieser „lokale Wahrnehmungsbereich“ macht es möglich, dass örtliche Informationen und Merkmale im Bild erhalten bleiben. Auch können große Bilder klassifiziert werden, ohne dass die Anzahl an nötigen Verbindungen im ANN unüberschaubar groß wächst. Die folgenden LTUs der zweiten Schicht sind ebenfalls wiederum nur mit einem Ausschnitt vorangegangener Neuronen verbunden und fassen die erkannten, kleinteiligen Merkmale der ersten Schicht zu übergeordneten, zusammengesetzten und komplexeren Merkmalen zusammen [1].

Um allerdings *Convolutional Layer* genauer zu verstehen, ist anstelle einer eindimensionalen Darstellung eines Layers eine dreidimensionale Darstellung besser geeignet.

Zunächst kann ein zweidimensionale Bild als Matrix dargestellt werden, bei der jedes Element der Matrix den Grauwert eines Pixels zwischen 0 und 255 trägt. Die dadurch entstandene zweidimensionale Schicht bildet den Input-Layer mit einer LTU pro Pixel. Anschließend werden auf diese Schicht nacheinander mehrere Filter angewandt, die die Gewichte des CNNs tragen und Muster aus dem Bild extrahieren. Die Stellen, die dem Muster ähnlich sind sollen verstärkt werden, während Stellen, die nicht dem Muster entsprechen durch eine Nullgewichtung ausgelöscht werden sollen. Der Lernprozess bei

der Bilderkennung beruht also darauf, die bestmöglichen Filter für die gegebene Aufgabe zu finden und diese für eine Erkennung komplexer Mustern zusammen setzen zu können. In Abbildung A.3 ist ein 3x3 Pixel Filter mit dessen Anwendung dargestellt. Ein Filter besitzt die Größe des künstlichen Wahrnehmungsbereiches einer LTU [1].

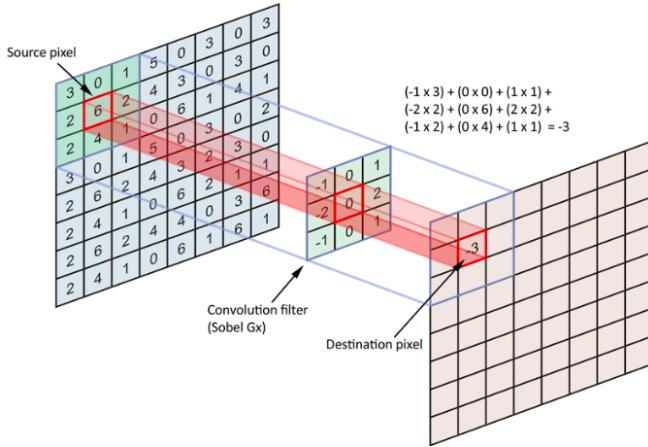


Abbildung A.3.: Convolutional Layer [60]

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Abbildung A.4.: Zero-Padding [61]

Ein Filter k wird dazu verwendet, jeden Pixel der Eingabeschicht $I(x, y)$ nach der mathematischen Faltungsoperation

$$I(x, y) = \sum_{i=1}^n \sum_{j=1}^n I(x-i, y-j)k(i, j) \quad (\text{A.5})$$

auf die folgende Schicht abzubilden. Um keine Informationen zu verlieren und den Filter ebenso auf Randbereich anwendbar zu machen, wird oft ein sogenanntes *Zero-Padding* auf eine Schicht angewandt, bei dem die Randbereiche mit LTUs des Wertes 0 aufgefüllt werden (siehe Abbildung A.4) [1].

Falls eine gleich große folgende Schicht gewünscht ist, wird eine Schrittweite (engl.: *stride*) von 1 gewählt. Dies dient vor allem dazu kleinere Strukturen noch zu erkennen. Der Filter wird von einem Pixel zum direkt benachbarten Pixel bewegt und angewandt. In tieferen, fortgeschritteneren Schichten kann die Schrittweite auch größer als 1 gewählt werden, da hier bereits nach dem Anwenden mehrerer Filter feinere Muster erkannt wurden und diese nun zu größeren zusammengesetzt werden. Dabei verkleinert sich die resultierende Schicht [1].

Das Ergebnis der Anwendung eines Filters wird als *Feature Map* bezeichnet. Da mehrere Filter auf die gleiche Schicht angewandt werden, entstehen ebenso mehrere *Feature Maps* der Schicht. Werden diese *Feature Maps* übereinander gelagert vorgestellt, so entsteht der dreidimensionale, „faltungsbedingte“ (engl.: convolutional) Charakter eines *Convolutional Layers*. Eine Schicht eines *Convolutional Layers* ist mit den entsprechenden Wahrnehmungsbereichen aller vorhergehenden *Feature Maps* des vorhergehenden *Convolutional Layers* verbunden (siehe Abbildung A.5) [1].

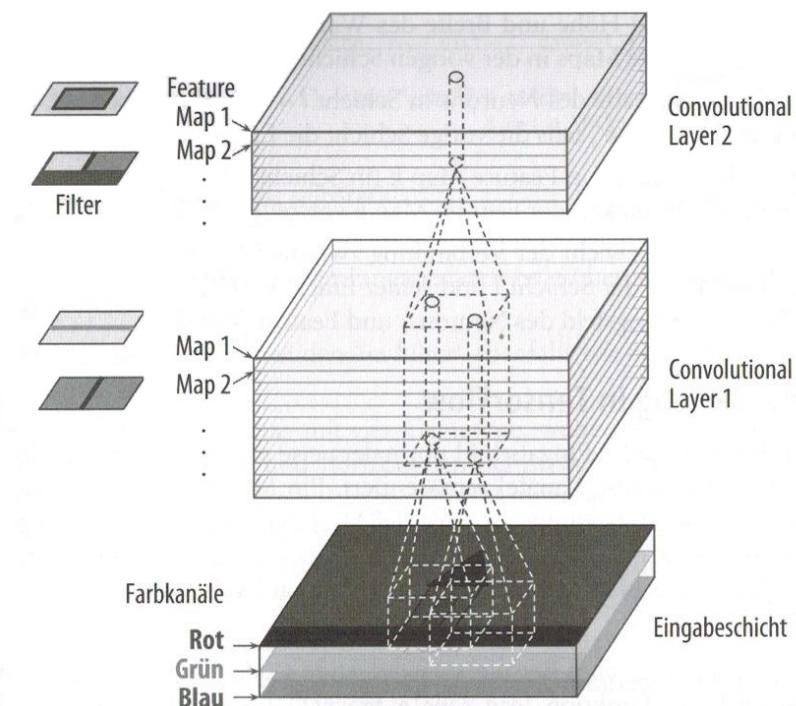


Abbildung A.5.: Veranschaulichung von Feature Maps [1]

Falls zusätzlich eine Farberkennung gewünscht ist, besitzt der Input Layer für jeden der drei Farbkanäle des RGB-Schemas eine Schicht, die Werte zwischen 0 und 255 in ihren LTUs tragen und den Stärken des Rot-, Grün- und Blaukanals entsprechen [1].

Der zweite Grundbaustein eines CNN sind *Pooling Layer*. Ähnlich zu den *Convolutional Layern* ist auch hier jede LTU nur mit einer begrenzten Anzahl an LTUs des vorhergegangenen Layers verbunden, also nur mit dem lokalen Wahrnehmungsbereich. Der Hauptunterschied liegt aber darin, dass keine Filter existieren, die die Werte vorhergehender LTUs unterschiedlich gewichten und dabei Muster erkennen. Statt den Filtern

werden Aggregatfunktionen wie $MAX()$ oder $MEAN()$ dazu verwendet, um die Eingaben in nachfolgende Schichten zu verkleinern. So wird beispielsweise bei einem *MAX-Pooling Layer* mit Schrittweite größer als 1 der jeweils größte Wert des lokalen Wahrnehmungsreiches weitergereicht und damit die Eingabe in nachfolgende Schichten verkleinert (siehe Abbildung A.6), was mit einem Informationsverlust verbunden ist. Diese Verkleinerung des Bildes ist ein wesentlicher Schritt, um bei der Mustererkennung weiter Informationen und Merkmale abstrahieren zu können [1].

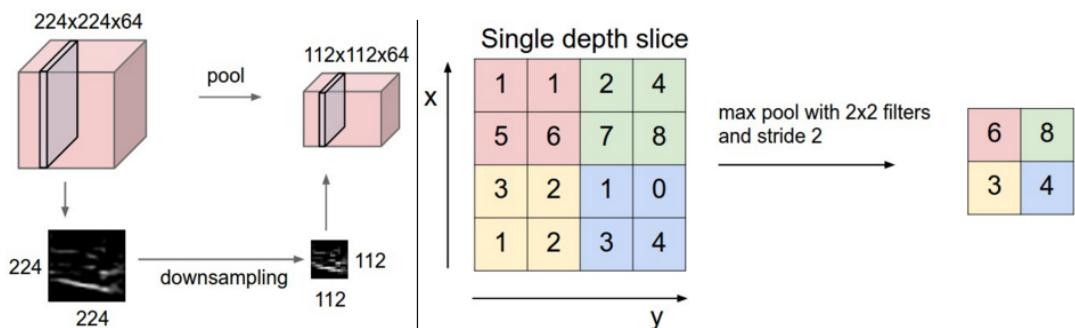


Abbildung A.6.: Pooling Layer [62]

Daneben ist ein Pooling über die Tiefe der *Feature Maps* möglich. Hier bleibt die Größe der resultierenden *Feature Maps* gleich, die Anzahl verringert sich allerdings. Die unterschiedlichen Farbkanäle werden somit nach und nach abstrahiert [1].

Nachdem nun beide Grundbausteine eines CNNs genauer erläutert wurden, lassen sich diese nun kombinieren um ein vollständiges CNN zu bauen. Hierbei gibt es unterschiedlichste Architekturen, größtenteils äußerst komplexe. Im Rahmen dieser Arbeit genügt es allerdings, die grundlegende Architektur zu erläutern.

Diese beginnt mit einigen *Convolutional Layern*, die aufeinander folgen und am Ende durch eine ReLU-Funktion nochmals gefiltert und durch ein *Pooling Layer* abgeschlossen werden. Dies wird je nach Komplexität der zu erkennenden Muster und der Größe der Bilder einige Male wiederholt. Das ursprüngliche Bild wird durch die *Pooling Layer* zwar immer kleiner, allerdings auch durch die *Convolutional Layer* immer tiefer. Das CNN schließt mit einem normalen *Feed-Forward ANN* mit *Fully-Connected Layern* ab, generiert dabei einen *Feature Vektor* und trifft durch eine Softmax-Funktion eine Klassifizierungsaussage des Bildes [1].

Diese Architektur ermöglicht ebenso die Wiederverwendbarkeit einzelner Schichten und Gewichtungen für ähnliche Klassifikationsprobleme, bei denen gleiche Muster vorzufinden sind [1].