DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

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 - Mai 2019

-Sperrvermerk-

Abgabedatum:	18. Mai 2020
Bearbeitungszeitraum:	30.09.2019 - 18.05.2020
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 versicheren 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 versicheren zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 15. Februar 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 and economically reasonable. In addition, the focus is also on the object detectors themselves and their differences in architecture and behavior in the *Smart Warehouse* environment.

Abstract

- Deutsch -

In dieser Arbeit werden die Objektdetektoren *You Only Look Once* und *Single Shot MultiBox Detector* nach Präzision, Reaktionsvermögen, Trainings- und Interaktionsverhalten 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 sowie wirtschaftlich sinnvoll ist. Zusätzlich liegt der Fokus ebenso auf den Objektdetektoren selbst und deren Unterschiede hinsichtlich Architektur und Verhalten im *Smart Warehouse* Umfeld.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
COCO	Common Objects in Context
CLI	Command Line Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
ELU	Exponential Linear Unit
FLOPS	Floating Point Operations Per Second
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
IoU	Intersection over Union
LReLU	Leaky Rectified Linear Unit
mAP	mean Average Precision
PReLU	Parametric Rectified Linear Unit
PascalVOC	Pascal Visual Object Classes
PaaS	Platform-as-a-Service
ReLU	Rectified Linear Unit
SaaS	Software-as-a-Service
SSD	Single Shot MultiBox Detector
TOPS	Tera Operations Per Second
TPU	Tensor Processing Unit
XML	Extensible Markup Language
YOLO	You Only Look Once

Abbildungsverzeichnis

Formelverzeichnis

Listenverzeichnis

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 [1]. 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 [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 *You Only Look Once* (YOLO) oder *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 Be-

trieb 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 klassifizieren.

Neben der Frage, ob ein solches Industrieszenario überhaupt umsetzbar und wirtschaftlich sinnvoll 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 Interaktionsverhalten untergliedern und werden später genauer eingeführt. Dadurch lassen sich Aussagen darüber treffen, ob 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

Zunächst muss sich mit den theoretischen Grundlagen von CNNs und Objektdetektoren auseinander gesetzt werden. Hierzu ist zunächst eine Einführung in neuronale Netz erforderlich, darunter zu Perzeptronen, dem Gradientenverfahren, dem Backpropagation Algorithmus und Hyperparametern zum Trainieren eines neuronalen Netzes.

Nachdem kurz auf den Grundbaustein moderner Objektdetektoren eingegangen wird, den CNNs, können anschließend die Funktionsweisen und Architekturen der zwei miteinander verglichenen Objektdetektoren *YOLO* und *SSD* erläutert werden. Bei *YOLO* ist zu bemerken, dass unterschiedliche Evolutionsstufen der drei Detektoren zu betrachten sind.

Um weitere Grundlagen zum Umfeld während des Trainierens von neuronalen Netzen einzuführen, wird anschließend über die Anforderungen eines Datensatzes gesprochen, bevor Trainingsinfrastruktur in der Cloud und das verwendete *Deep Learning* Framework näher gebracht wird.

In der Konzeptionsphase sollen zunächst die Vergleichskriterien eingeführt werden und deren Metriken anschließend für initiale Benchmarkdatensätze für jeden Objektdetektor ermittelt werden. Hierzu wird auf die Datensätze *Pascal Visual Object Classes* (Pascal-VOC), *Common Objects in Context* (COCO) und ImageNet zurückgegriffen. Anschließend wird der Datensatz für das *Smart Warehouse* Szenario eingeführt.

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 wird Bestandteil dieses Kapitels sein. Die Ergebnisse der Realisierungsphase werden 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. Auch wirtschaftliche Gesichtspunkte werden in diesem Kapitel nicht außer Acht gelassen.

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

Im folgenden Kapitel soll sich speziell mit den Architekturen der unterschiedlichen Objektdetektoren auseinander gesetzt werden und wie sich diese voneinander abgrenzen. Außerdem wird grundlegendes Wissen über neuronale Netze und wie diese „lernen“ vermittelt, um die späteren Optimierungsverfahren an den Objektdetektoren zu verstehen. Zuletzt werden technische Grundlagen für die Programmierung des *Smart Warehouse* Szenarios vermittelt.

2.1. Neuronale Netze

Ein neuronales Netz bildet die Grundlage des *Deep Learnings* [1]. Zunächst soll die einfachste Architektur eines neuronalen Netzes, das Perzeptron [1], exemplarisch erklärt werden als auch der Lernprozess eines maschinellen Lernmodells an sich, um darauf basierend die Auswirkungen von Hyperparametern auf den Lernprozess des Modells zu erklären.

Das Perzeptron

Der Aufbau eines typischen Perzeptrons besteht aus einer oder mehreren Schichten sogenannter *Linear Threshold Units* (LTU) wie in Abbildung 2.1 dargestellt.

Es besteht aus n Eingängen mit $x_i \in \mathbb{Q}$, die im Inputvektor $\boldsymbol{x}$ zusammengefasst werden. Jeder Eingang wird mit einem Gewicht w_i aus dem Gewichtsvektor $\boldsymbol{w}$ versehen [1]. Die LTU berechnet das Skalarprodukt $\boldsymbol{w}^T \circ \boldsymbol{x}$ aller Eingänge $\boldsymbol{x}$ mit ihren Gewichten $\boldsymbol{w}$ und wendet anschließend auf das Ergebnis z eine Aktivierungsfunktion an [1]. Das Ergebnis $h_w(x)$ kann anschließend als Eingabe für ein weiteres Perzeptron dienen. Die einfachste Aktivierungsfunkion für ANNs ist die *Heaviside-Funktion* (2.1) [1]. Falls eine Klassifizierung mit Wahrscheinlichkeiten vorliegen soll, so ist die letzte Schicht eines Perzeptrons meist mit der *Softmax-Funktion* (2.2) implementiert, die den Wert des j-ten LTUs einer Schicht mit allen anderen n Werten der LTUs derselben Schicht ins

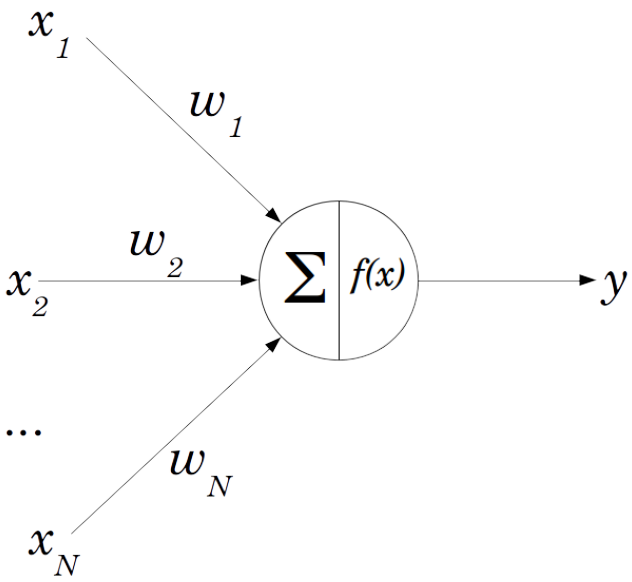


Abbildung 2.1.: Linear Threshold Unit [4]

Verhältnis setzt [1]. Es gibt eine Vielzahl an möglichen Aktivierungsfunktionen, die im darauffolgenden Unterkapitel *Hyperparameter* betrachtet werden.

$$h_w(x) = s(\boldsymbol{w}^T \circ \boldsymbol{x}) = s(z) = \begin{pmatrix} w_1 & w_2 & \dots & w_n \end{pmatrix} \circ \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}) = \begin{cases} 1 & \text{wenn} z \geq 0 \\ 0 & \text{wenn} z < 0 \end{cases} \tag{2.1}$$

$$h_w(x) = \sigma(z)_j = \frac{e^{z_j}}{\sum_{i=0}^{n} e^{z_i}} \tag{2.2}$$

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. Wählt man die Gewichtung negativ, 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 Perzeptron. Jede einzelne LTU ist dabei mit allen LTUs der vorherigen Schicht verbunden (siehe Abbildung 2.2) [1]. Die beiden LTUs zur Ausgabe können dabei Aussagen über eine Klassifikation von Daten anhand der Eingangsdaten treffen, während die LTUs im Input Layer wesentlich die Daten weiter reichen. Die Verbindungen zur ersten Schicht des Hidden Layer sind stets mit Eins belegt. Existiert keine verborgene Schicht, so bezeichnet man das ANN als einschichtiges Perzeptron, ab einer oder mehr verborgenen Schichten spricht man bereits von einem *Multi-Layer Perzeptron* (MLP), einem mehrschichtigen Perzeptron [1]. 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 Schichte zurückzukehren [1].

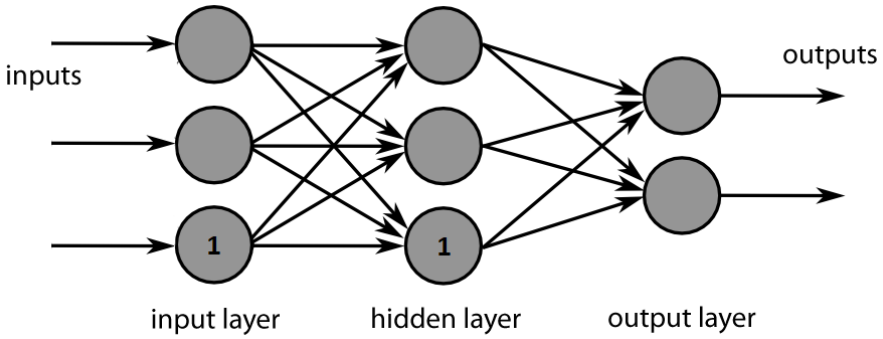


Abbildung 2.2.: Das einschichtige Perzeptron [5]

Gradientenverfahren und Backpropagation

Um zu verstehen, wie ein neuronales Netz durch Training 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) (2.3) [1]

$$E(\boldsymbol{z}, \boldsymbol{o}) = \sqrt{\frac{1}{n}\sum_{k=0}^{n}\|z_k - o_k\|^2} = \sqrt{\frac{1}{n}\sum_{k=0}^{n}(z_k - o_k)^2}. \tag{2.3}$$

Hierbei ist z der erwartete Ausgabevektor des Perzeptrons, während o die momentane Ausgabe darstellt. Da der erwartete Ausgabewert bekannt ist, spricht man auch von sogenanntem überwachtem Lernen [1]. 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 Perzeptron
2. Bias zur Aktivierung der LTUs des Perzeptrons und
3. Stärke der Aktivierung des vorherigen Perzeptrons

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 (2.4) ist die Gewichtungsmatrix, die die Änderung der Gewichtung jeder einzelnen Verbindung eines Perzeptrons zu jeder LTU des Folgeperzeptrons angibt [1].

$$w_{ijt} = w_{ijt-1} - \eta\frac{\partial E}{\partial w_{ij}} \qquad [4] \tag{2.4}$$

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 der später eingeführten Sigmoid-Funktion im gesamten Definitionsbereich immer kleine Änderung 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 [1]. Dies hat zur Folge, dass mehrere iterative Durchlaufe mit mehreren Testdatensätzen nötig werden, um durch unterschiedliche Startkonfigurationen die unterschiedlichen Minima miteinander vergleichen zu können und damit das globale Minimum herauszustellen.

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* [6]. Dieser berechnet mit Hilfe des Gradientenverfahren welchen Anteil am Fehler der Ausgabe jede LTU des letzten Perzeptrons 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.

2.2. Hyperparameter

Hyperparameter sind die Parameter, die zur anfänglichen Konfiguration des neuronalen Netzes als auch zur Konfiguration des Lernprozesses heran gezogen 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 (2.5) gilt.

$$Var(X + Y) = Var(X) + Var(Y) \tag{2.5}$$

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 (siehe Abbildung 2.4) gegen Null konvergiert [1].

Eine *Xavier Initialisierung* umgeht das Problem der sogenannten *schwindenden Gradienten*, indem die Gewichte nach 2.6 gleichverteilt werden, wobei nj die Anzahl an LTUs

der $j-ten$ Schicht sind [7].

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}] \tag{2.6}$$

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 zu erreichen und damit alleinig eine richtige Erkennung der Trainingsdatensätze zu ermöglichen.

Lernrate

Die Lernrate η (2.4) gibt an, wie groß die Sprünge zum globalen Minimum sein sollen und damit indirekt wie viele Iterationen benötigt werden, um das globale 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 (2.7) allerdings nicht zu groß wird, beschränkt der Hyperparameter $\beta \in [0, 1]$ die Größe des Momentums [1].

$$\begin{aligned} m_x &= \beta \cdot m_{x-1} + \eta \frac{\partial E}{\partial w_{ij}} \\ w_{ijt} &= w_{ijt-1} - m \end{aligned} \tag{2.7}$$

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 unterscheidet man zwischen drei verschiedenen Arten das Gradientenverfahren durchzuführen (siehe Abbildung 2.3):

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 Mimima wieder zu überwinden [1].

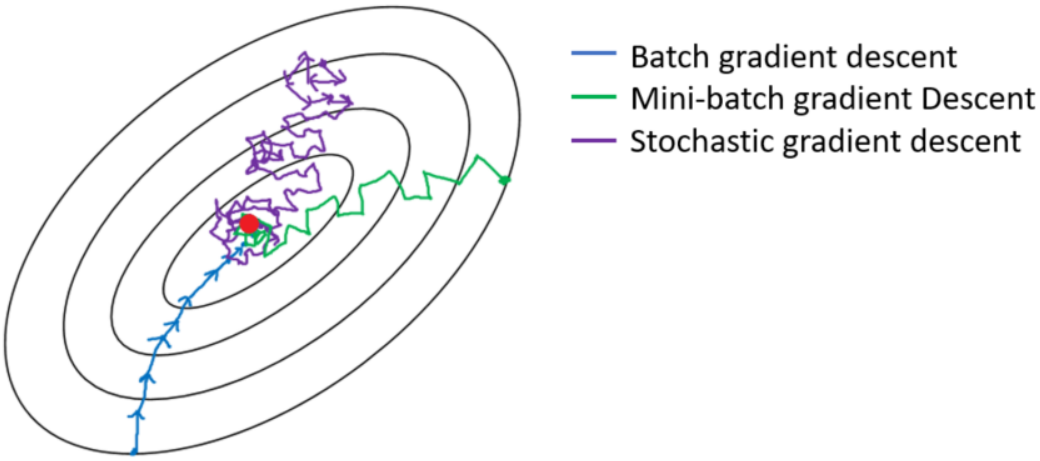


Abbildung 2.3.: Gradientenverfahren [8]

Ein Kompromiss der beiden Verfahren bietet das *Mini-Batch* Verfahren, bei dem wiederholt Teilmengen des gesamten Datensatzes für einen Gradientenabstieg verwendet werden. 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 [1].

Aktivierungsfunktionen

Zwei bekannte und ähnliche Aktivierungsfunktionen sind die *Sigmoid-Funktion* und die *Tangens Hyperbolicus* Funktion (siehe Abbildung 2.4).

Da diese allerdings anfällig für das Problem *schwindender Gradienten* sind [1], wird die *Rectified Linear Unit* (ReLU) bzw. *Parametric/Leaky Rectified Linear Unit* (PReLU/L-RelU) Aktivierungsfunktion bevorzugt (siehe Abbildung 2.5).

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

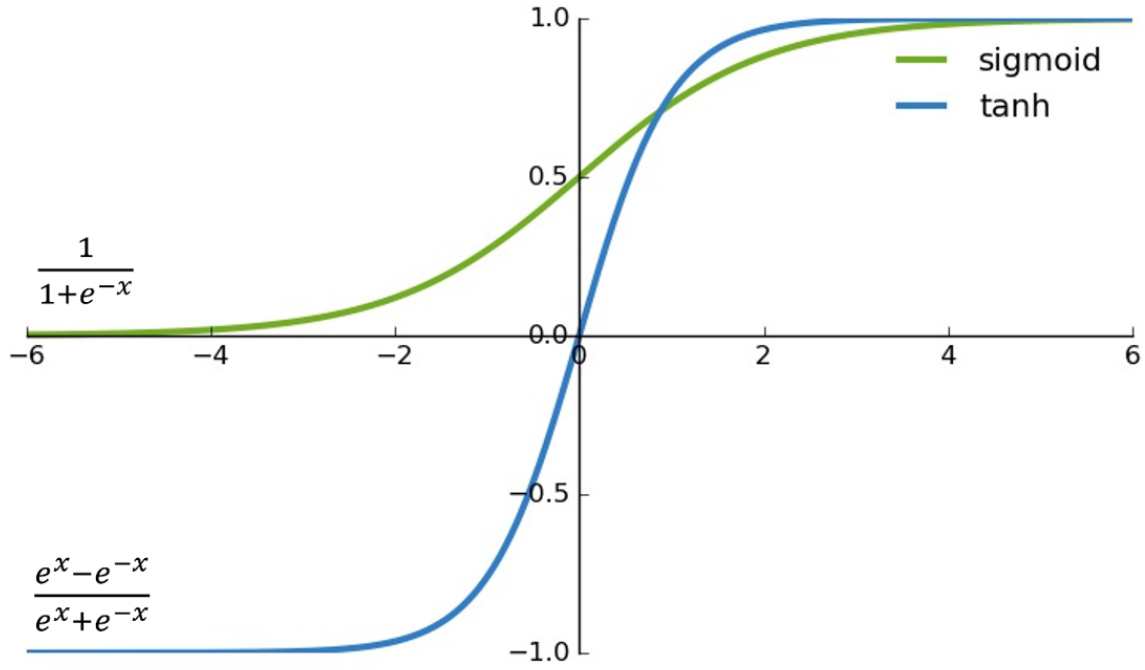
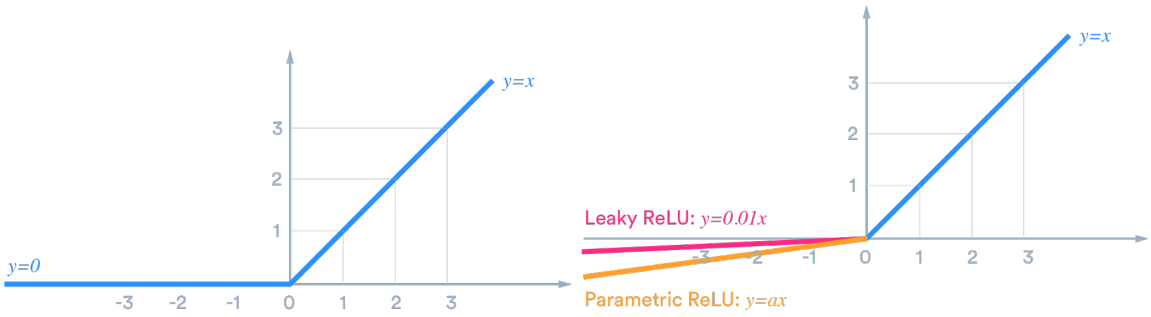


Abbildung 2.4.: Sigmoid und Tangens Hyperbolicus [9]

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 2.6).

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



(a) RELU (b) PReLU/LReLU

Abbildung 2.5.: ReLU-Aktivierungsfunktionen [10]

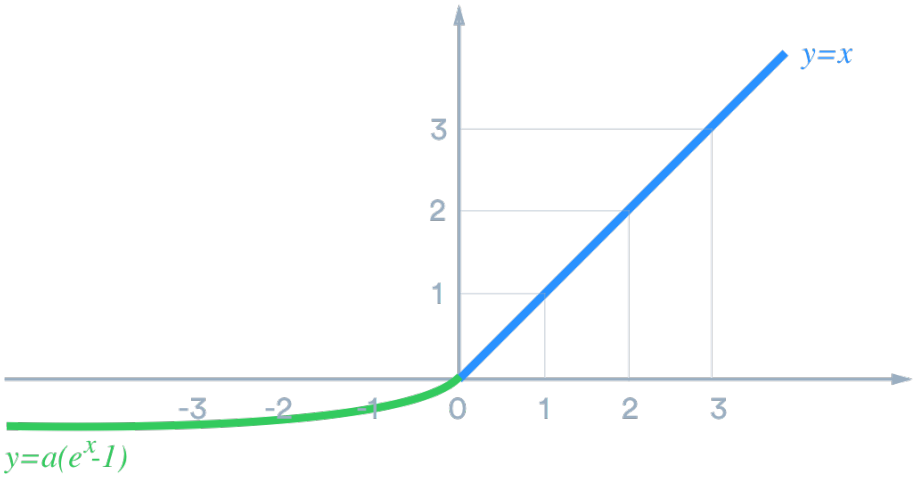


Abbildung 2.6.: ELU [10]

2.3. 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 drei gängigsten Objektdetektoren, das *Mask Regional Convolutional Neural Network*, der *Single Shot MultiBox Detector* und der *You Only Look Once* Ansatz eingegangen wird.

Convolutional Neural Networks

Ein CNN besteht aus zwei grundlegenden Bausteinen, den sogenannten *Convolutional Layern* und *Pooling 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. Dies macht es möglich, dass auch große Bilder klassifiziert werden können, 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 mehrere Filter angewandt, die die Gewichte des CNNs tragen und Muster aus dem Bild extrahieren. Die Stellen, die dem Muster ähnlich sind werden verstärkt, während Stellen, die nicht dem Muster entsprechen durch eine Nullgewichtung ausgelöscht werden. In Abbildung 2.7 ist ein 3x3 Pixel Filter mit dessen Anwendung dargestellt. Ein Filter besitzt die Größe des künstlichen Wahrnehmungsfeldes einer LTU [1].

Ein Filter wird dazu verwendet, jeden Pixel der Eingabeschicht 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 2.8) [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,

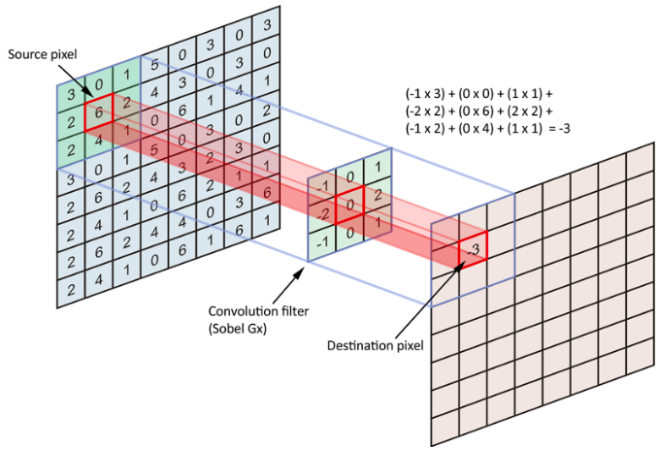


Abbildung 2.7.: Convolutional Layer [11]

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 2.8.: Zero-Padding [12]

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. Stellt man sich diese Feature Maps übereinander gelagert vor, 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 2.9) [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 Lernprozess bei der Bilderkennung beruht nun darauf, die optimalsten Filter für die gegebene Aufgabe zu finden und diese zu komplexen Mustern zusammen setzen zu können [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 Wahrnehmungsfeld. Der Haupt-

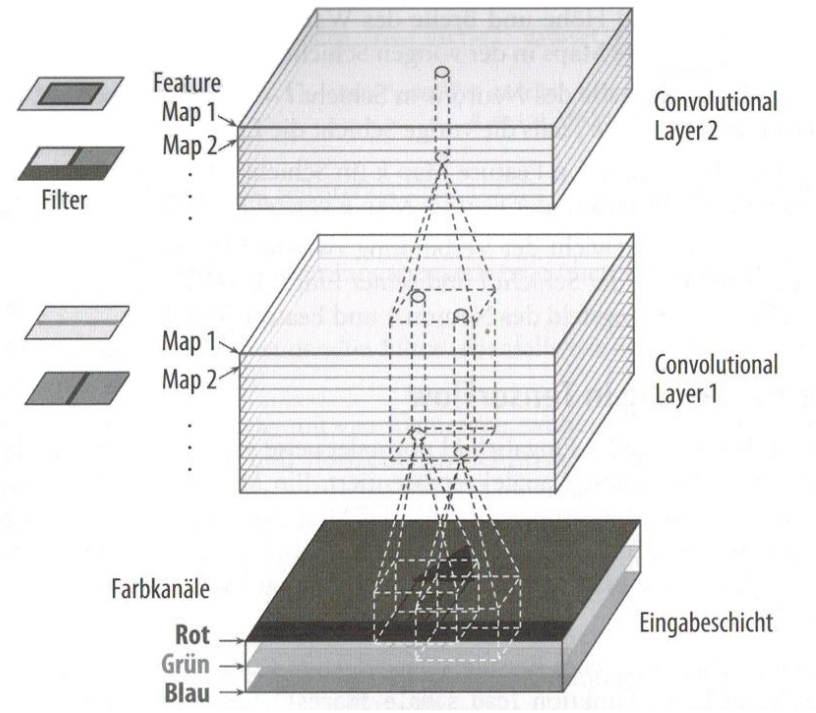


Abbildung 2.9.: Feature Maps [1]

unterschied liegt aber darin, das keine Filter existieren, die die Eingaben unterschiedlich gewichten und dabei Muster erkennen, sondern stattdessen Aggregatfunktionen wie *MAX()* oder *MEAN()* dazu benutzt werden, um Eingaben zu verkleinern. So wird beispielsweise bei einem MAX-Pooling Layer mit Schrittweite größer als 1 der jeweils größte Wert des lokalen Wahrnehmungsfeld weitergereicht und durch die gewählte Schrittweite das vorherige Ergebnis verkleinert (siehe Abbildung 2.10), was mit einem Informationsverlust verbunden ist. Dies ist allerdings ein wesentlicher Schritt, um weiter abstrahieren zu können [1].

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 [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

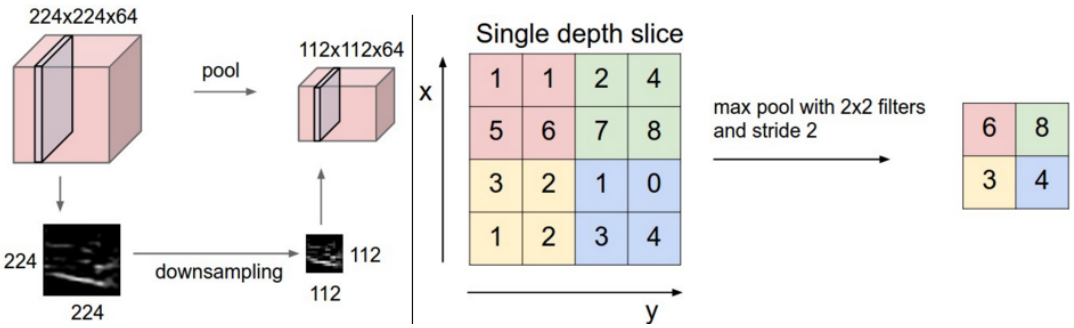


Abbildung 2.10.: Pooling Layer [13]

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 vollständig verbundenen Schichten ab und trifft durch eine Softmax-Funktion eine Klassifikationsaussage des Bildes [1].

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

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) unterschiedet sich von den vorhergegangenen Modellen 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 [14].

Die Architektur des *SSD* zielt zunächst darauf ab, ein Bild in mehrere unterschiedliche

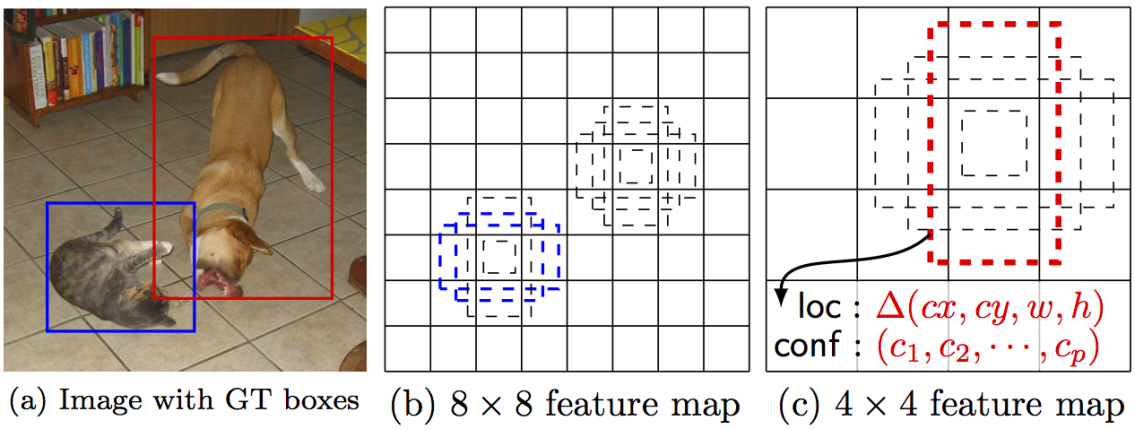


Abbildung 2.11.: SSD Grundprinzip [14]

Gitterstrukturen zu unterteilen, die sich nach ihrer Skalierung unterscheiden. Dadurch ist es möglich Objekte unterschiedlicher Größe zu erkennen. Jede Lokation im Gitter besitzt eine gleiche Anzahl an festen, vordefinierten Bounding Boxen, die unterschiedliche Seitenverhältnisse aufweisen (siehe Abbildung 2.11). So wird sichergestellt, dass sowohl horizontal als auch vertikal ausgeprägte Objekte in der selben Lokation gleichzeitig erkannt werden können (siehe Abbildung 2.12) [14].

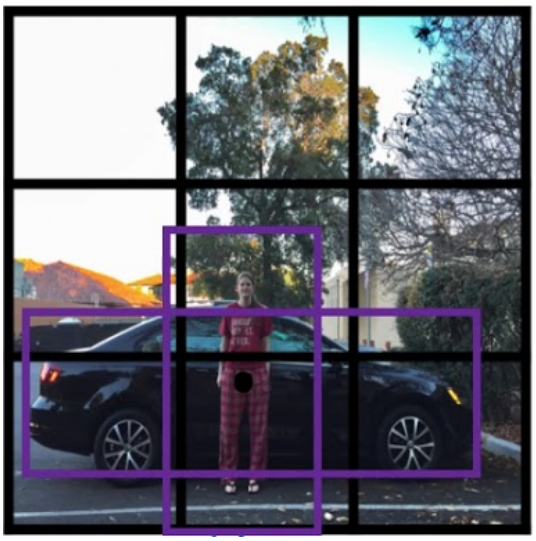


Abbildung 2.12.: Bounding Boxes [15]

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. Dies wird mit kleinen Faltungsfiltern erreicht, die in den hinteren Schichten des Netzwerks eingesetzt werden [14].

Die Kostenfunktion ist durch die gewichtete Summe des Lokalisationsverlustes und des Klassifikationsverlustes bestimmt. Während der Klassifikationverlust durch eine *Softmax* Funktion bestimmt werden kann, wird der Lokalisationsverlust über die *Smooth L1* Funktion (2.8) bestimmt [14].

$$
\begin{aligned}
L_{loc}(t^u, v) &= \sum_{i \in (x,y,w,h)}^{n} SM_{L1}(t_i^u - v_i) \\
SM_{L1}(t_i^u - v_i) &= \begin{cases} 0.5x^2 & \text{wenn } |x| < 1 \\ |x| - 0.5 & \text{sonst} \end{cases}
\end{aligned}
\tag{2.8}
$$

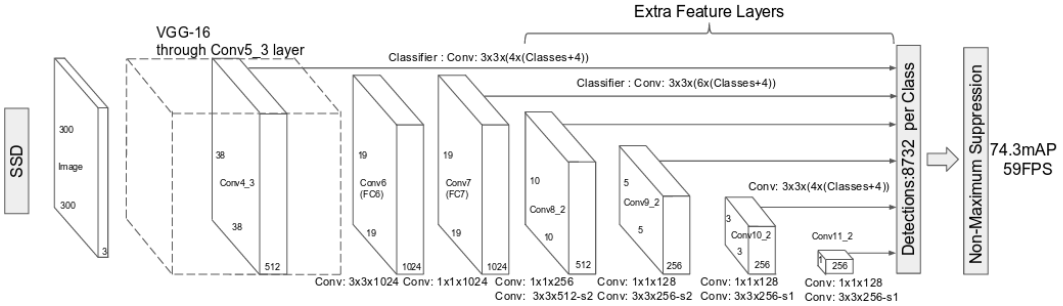


Abbildung 2.13.: SSD Architektur [14]

Der *SSD* benutzt ein *VGG-16* Basis Netzwerk, um Klassifikationen zu ermöglichen. *VGG-16* ist ein auf dem Datensatz von *ImageNet* basierendes neuronales Netz, das bis zu 1000 unterschiedliche Kategorien klassifizieren kann [16]. Unabhängig vom Basisnetzwerk werden Convolutional Layer als Hilfsstrukturen zur Objektdetektion eingesetzt. Diese verarbeiten die Bilder unterschiedlicher Gittergrößen, wobei die Gittergröße mit fortschreitenden Convolutional Layern abnimmt. Jeder 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 durch einen Eckpunkt $P(x, y)$ und eine Höhe und Breite bestimmt. Bei c Klassen hat der

Ausgangsvektor demnach die Größe $c + 4$. Für jede vordefinierte Bounding Box werden, falls vorhanden, die Koordinaten der originalen Bounding Box ermittelt dabei relativ zur vordefinierten Bounding Box abgespeichert. Die Zuordnung einer Bounding Box zu einer vordefinierten Bounding Box erfolgt über die sogenannte *Intersection over Union* (IoU) (siehe Abbildung 2.14). Überschreitet diese einen Wert von 0.5, so ist diese der originalen Bounding Box zugeordnet [14]. Demnach ist es möglich, dass eine originale Bounding Box mehreren vordefinierten Bounding Boxen zugeordnet werden kann [14] .

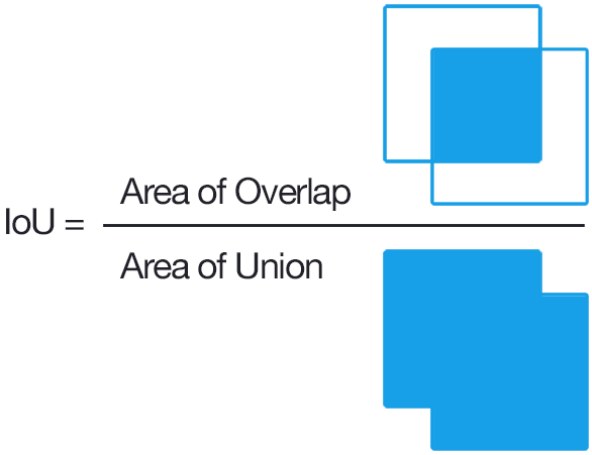


Abbildung 2.14.: Intersection over Union [17]

Bei $m \cdot n$ Lokationen und k verschiedenen vordefinierten Bounding Boxen pro Lokation ergeben sich also $m \cdot n \cdot k \cdot (c + 4)$ verschiedene Werte für eine Feature Map. Die Merkmalsextraktion zur Klassifikation wird durch Faltung mit $3x3$ Filtern erreicht [14].

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 sogenannte *Intersection over Union* (IoU) der detektierten Box zur wahren Box verwendet [14].

Während der Trainingsprozesses des *SSD300*[1] mit PASCAL VOC2007 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 wur-

[1] 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.

de die Lernrate für 10.000 Iterationen auf $\eta = 10^{-4}$ reduziert und schließlich auf $\eta = 10^{-5}$ [14].

Method	mAP	FPS	batch size	# Boxes	Input resolution
Faster R-CNN (VGG16)	73.2	7	1	~ 6000	$\sim 1000 \times 600$
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 2.15.: Vergleich SSD [14]

Nach obiger Grafik ist eindeutig festzustellen, dass *SSD300* ein gutes Verhältnis zwischen Präzision und Reaktionsvermögen bewahrt. Durch den Verzicht auf den Schritt der Generierung von Bounding Box Vorschlägen und des *Poolings* kann *SSD* deutlich schneller ablaufen als die Vergleichsdetektoren, während durch das Vordefinieren von Bounding Boxen ebenso eine hohe Präzision erzielt werden kann [14].

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 Daten-Augmentierung für den Lernprozess angewandt werden. Weitere mögliche Fehler können falsche Positivbefunde aufgrund von fehlerhafter Lokalisation sein, Verwechslung mit ähnlichen Kategorien oder dem Hintergrund sowie weitere [14].

Letztendlich lässt sich *SSD* als schneller Objektdetektor beurteilen, der nicht nur einfach zu trainieren und integrieren ist, sondern ebenso ein gutes Verhältnis zwischen Lokalisationspräzision und Echtzeitvermögen schafft.

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 vorherzusagen.

Hierzu wird ein $S \times S$ Gitter über das Bild gelegt. Für jedes Feld im Gitter werden B Bounding Boxen erzeugt. Jede Box besitzt neben den zum Gitterfeld relativen Positionswerten einen Wert, der die Vorhersage der jeweiligen Klasse und die Präzision der Box repräsentiert. Dieser Wert wird als *confidence score* bezeichnet und wird durch die Multiplikation der Wahrscheinlichkeit für eine Klasse mit der IoU, also die Präzision der berechneten Box im Verhältnis zu der Box aus den vortrainierten Testdaten festgelegt [18].

Aus der Menge an Bounding Boxen werden schließlich mit Hilfe eines festgelegten Schwellwertes die Boxen mit lokalisierten Objekten bestimmt (siehe Abbildung 2.16).

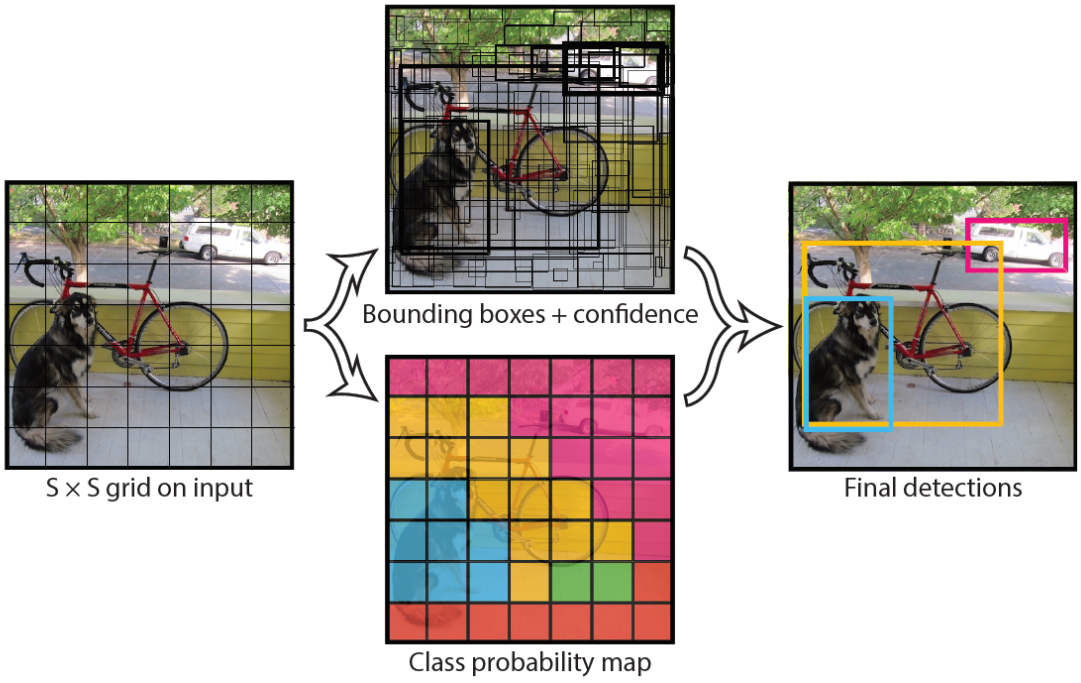


Abbildung 2.16.: Vereinfachte Darstellung der Objekterkennung mit dem YOLO Algorithmus [18]

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 definiert. Abbildung 2.17 zeigt den Aufbau des CNN von *YOLO* für die Detektion. Es besteht aus 24 Convolutional Layern gefolgt von 2 Fully-connected Layern. Für die Genauigkeit bei der Detektion wird die Auflösung des Bildes verdoppelt [18].

//TODO erkläre layer

In dem Netzwerk in Abbildung 2.17 wird ein Bild mit einer Auflösung von 224×224 Pixeln verwendet und die vorhergesagten Werte im $7 \times 7 \times 30$ Tensor ausgegeben.

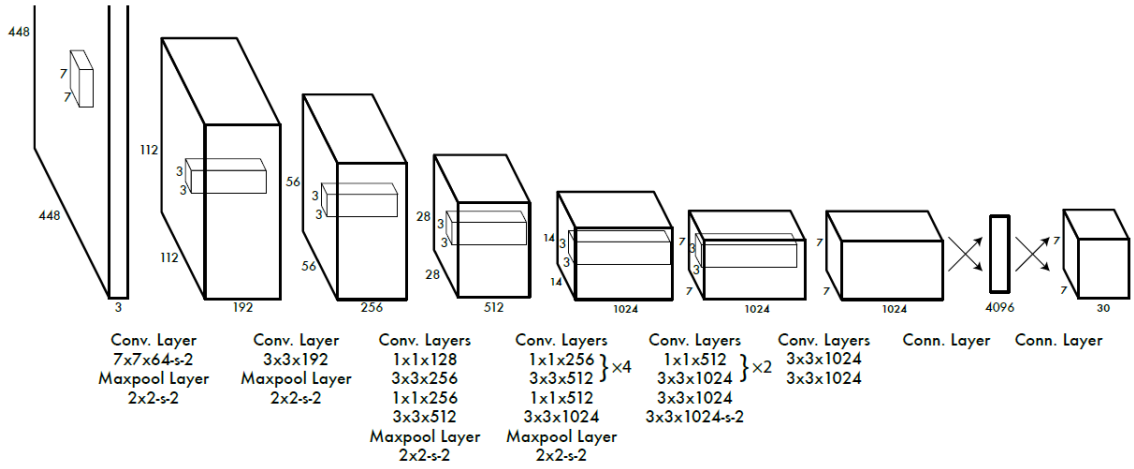


Abbildung 2.17.: *YOLO* Architektur [18]

Die mittlerweile dritte und aktuelle Version von *YOLO* weist enorme Verbesserungen auf, gerade im Bezug auf die Erkennung von sehr kleinen Objekten wie zum Beispiel einzelne Vögel in einem Schwarm [19].

TODO Veränderung im Netz: https://www.cyberailab.com/home/a-closer-look-at-yolov3

2.4. Datensatzlehre

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 *Annotations*, *ImageSets* und *JPEGImages*. Während in dem Ordner *JPEGImages* alle Bilder des Datensatzes vorhanden sind, befindet sich unter anderem die Information über die vorhandenen Objekte in dem Bild im Ordner *Annotations*. Für jedes Bild des Datensatzes werden die Informationen in einer gleichnamigen XML-Datei abgelegt.

```
1  <annotation>
2      <folder>JPEGImages</folder>
3      <filename>000001.jpg</filename>
4      <path>D:\Workspaces\Datasets\SmartWarehouse\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

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- und Testdatensatz durch zwei Textdateien realisiert, die die Dateinamen der Bilddateien als Auflistung enthalten [20].

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 [21].

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. Dies liegt daran, 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 dabei am besten abschneidende Modell mit dessen Hyperparametern ausgewählt. Der eigentliche Testdatendatz 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* Datensatzformat ebenso die Möglichkeit Objekte als „schwierig erkennbar“ zu markieren, dennoch sollten solche Objekte nicht die Mehrheit im gesamten Datensatz ausmachen. Auch die Aufnahme von Objekten in unterschiedlichen Umgebungen, Entfernungen 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 2.15 wurden beispielsweise durch Kombination der *Trainval* Datensätze von PascalVOC 2007 und 2012 erzielt und umfasst 16.551 Bilder im Trainingsverfahren [14] [22] [23].

Unter Hinzunahme des COCO *trainval135k* Datensatzes erreicht der *SSD* sogar das beste

Method	data	mAP	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
Fast [6]	07	66.9	74.5	78.3	69.2	53.2	36.6	77.3	78.2	82.0	40.7	72.7	67.9	79.6	79.2	73.0	69.0	30.1	65.4	70.2	75.8	65.8
Fast [6]	07+12	70.0	77.0	78.1	69.3	59.4	38.3	81.6	78.6	86.7	42.8	78.8	68.9	84.7	82.0	76.6	69.9	31.8	70.1	74.8	80.4	70.4
Faster [2]	07	69.9	70.0	80.6	70.1	57.3	49.9	78.2	80.4	82.0	52.2	75.3	67.2	80.3	79.8	75.0	76.3	39.1	68.3	67.3	81.1	67.6
Faster [2]	07+12	73.2	76.5	79.0	70.9	65.5	52.1	83.1	84.7	86.4	52.0	81.9	65.7	84.8	84.6	77.5	76.7	38.8	73.6	73.9	83.0	72.6
Faster [2]	07+12+COCO	78.8	84.3	82.0	77.7	68.9	65.7	88.1	88.4	88.9	63.6	86.3	70.8	85.9	87.6	80.1	82.3	53.6	80.4	75.8	86.6	78.9
SSD300	07	68.0	73.4	77.5	64.1	59.0	38.9	75.2	80.8	78.5	46.0	67.8	69.2	76.6	82.1	77.0	72.5	41.2	64.2	69.1	78.0	68.5
SSD300	07+12	74.3	75.5	80.2	72.3	66.3	47.6	83.0	84.2	86.1	54.7	78.3	73.9	84.5	85.3	82.6	76.2	48.6	73.9	76.0	83.4	74.0
SSD300	07+12+COCO	79.6	80.9	86.3	79.0	**76.2**	57.6	87.3	88.2	88.6	60.5	85.4	**76.7**	**87.5**	**89.2**	84.5	81.4	55.0	81.9	**81.5**	85.9	78.9
SSD512	07	71.6	75.1	81.4	69.8	60.8	46.3	82.6	84.7	84.1	48.5	75.0	67.4	82.3	83.9	79.4	76.6	44.9	69.9	69.1	78.1	71.8
SSD512	07+12	76.8	82.4	84.7	78.4	73.8	53.2	86.2	87.5	86.0	57.8	83.1	70.2	84.9	85.2	83.9	79.7	50.3	77.9	73.9	82.5	75.3
SSD512	07+12+COCO	**81.6**	**86.6**	**88.3**	**82.4**	76.0	**66.3**	**88.6**	**88.9**	**89.1**	**65.1**	**88.4**	73.6	86.5	88.9	**85.3**	**84.6**	**59.1**	**85.0**	80.4	**87.4**	**81.2**

Abbildung 2.18.: SSD Grundprinzip [14]

Ergebnis aus der ursprünglichen Veröffentlichung (siehe Abbildung 2.18) [14].

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 zwei bekannte Techniken, wie auch mit kleineren Datenbeständen ein sehenswertes Ergebnis erzielt werden kann.

Beim sogenannten *Transfer Learning* werden von einem bereits für ein ähnliches Problem trainiertes Modell die ersten Schichten des neuronalen Netzes für das neue Modell wiederverwendet. 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].

2.5. Cloud Infrastruktur

Das Trainieren eines *Deep Learning* Modells ist gerade bei großen CNN Architekturen äußerst 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 demnach 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 *NVIDIA* nimmt hierbei eine Vorreiterrolle in der Produktion von Server-GPUs ein. Die *Compute Unified Device Architecture* (CUDA) von *NVIDIA* ermöglicht hierbei als Programmiermodel und parallele 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* [24] [25].

Vergleich man gängige Tesla GPUs, die Kunden als PaaS-Angebot zur Verfügung gestellt werden, nach ihrer Rechenleistung, so ergibt sich Tabelle 2.5 [26].

	K80	P100	T4	V100	GTX 1080	TITAN RTX
CUDA Cores	2496	3584	2560	5120	2560	4608
Tensor Cores	/	/	320	640	/	576
TeraFLOPS (Single Precision)	4,113	9,526	8,141	14,13	8,873	16,31
Memory Bandwidth (in GB/sec)	240,6	732,2	320	897	320,3	672
Suggested Power Supply Unit	700	600	350	600	450	600

Tabelle 2.1.: Vergleich von GPUs nach Rechenleistung

Auch wurde die *GeForce GTX 1080* sowie die *Titan RTX*, zwei Desktop-Grafikkarten ausgelegt auf Gaming und Entertainment, in den Vergleich mit aufgenommen. Der Ver-

gleich dient später dazu, um eine begründete Kosten-Nutzen Abwägung zwischen einem Trainieren der Modelle auf der CLoud oder lokal vollziehen zu können.

Neben GPUs existieren seit 2015 die von Google entwickelten *Tensor Processing Units* (TPUs). Diese Art von Spezialhardware erreicht pro TPU-Kern eine Rechenleistung von bis zu 92 TOPS [27]. Schließt man 2048 solcher TPU-Kerne zu einem TPU-Pod zusammen, so ergibt sich eine Rechenleistung von über 100 PetaFLOPS [28]. Zudem ist die größere Rechenleistung gleichzeitig effizienter als herkömmliche GPUs (siehe Abbildung 2.19)

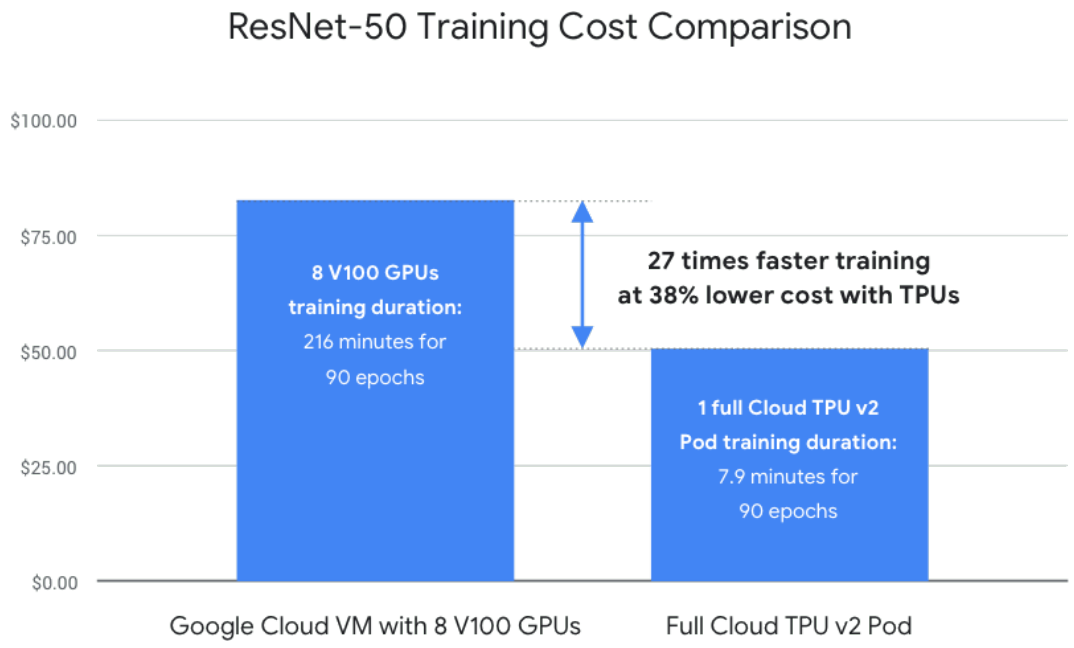


Abbildung 2.19.: Vergleich V100 - TPU Pod [29]

Eine weitere Steigerung versprechen Microsofts Field Programmable Gate Arrays (FPGAs), die allerdings nicht weiter im Rahmen dieser Arbeit betrachtet werden sollen [30].

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* [31].

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

```
machine: gpu2
env: pytorch-1.4
input:
  - destination: input
    source: <username>/datasets/smartwarehousessd/3
  - <username>/datasets/smartwarehousessd/3:ssd
description: Job to train the SSD
max_runtime: 3600
command: python train.py
```

Listing 2.2: Konfigurationsdatei zum Trainingsjob

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.

FloydHub betreibt ein zeitbasiertes Zahlungsmodell. So kosten 10 Stunden Laufzeit auf einer K80 12$, auf einer V100 bereits 42$. Zusätzlich werden monatliche Account-Gebühren berechnet. 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.

2.6. PyTorch

3. Konzeption

Um den *SSD* und den *YOLO* Objektdetektor miteinander vergleichbar zu machen und um deren Potential zum industriellen Einsatz zu bewerten, müssen konkrete Bewertungskriterien eingeführt werden. Anhand dieser Metriken werden die beiden Objektdetektoren auf einem Benchmark Datensatz initial verglichen, bevor der eigentliche Datensatz eingeführt wird. Dieser wird der Echtzeitumgebung entnommen.

3.1. Bewertungskriterien

Präzision

Um die Genauigkeit von Objektdetektoren zu messen, wird oft die Metrik *mean Average Precision* (mAP) gewählt. Diese setzt sich aus zwei grundlegenden Größen zusammen (siehe Abbildung 3.1).

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

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

TP = True positive
TN = True negative
FP = False positive
FN = False negative

Abbildung 3.1.: Precision und Recall [32]

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* [33].

Die Klassifikation, ob eine Region das gewünschte Objekt enthält und demnach ein positiver Fall vorliegt, wird anhand des definierten *IoU* Schwellwertes bestimmt. Dieser wird auch als *confidence score* bezeichnet. 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 [33].

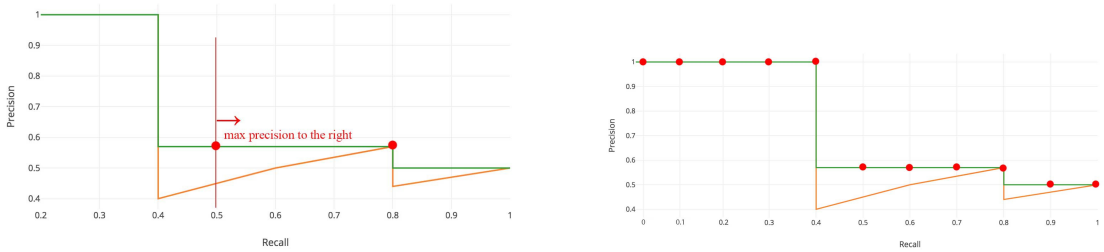


Abbildung 3.2.: Berechnung mAP [32]

Im Graphen ist meist ein klassisches „Zick-Zack“ Muster zu erkennen (siehe Abbildung 3.2). Dieses Muster wird geglättet, indem nach jedem Einbruch für jeden *Recall* Wert der maximale *Precision* Wert nach rechts übernommen wird. Bildet man anschließend das diskrete Integral, so ergibt sich der *Average Precision* Wert für eine zu klassifizierende Kategorie. Der Mittelwert der *Average Precisions* über alle kategorien hinweg ergibt letztendlich den *mAP* Wert [32].

Reaktionsvermögen

FPS

Trainingsverhalten

Interaktionsverhalten

beleuchtung, doppelte erkennung

3.2. Initialer Vergleich der Objektdetektoren

- Verfahren incl. Variantendiskussionen
- Funktionsweise an Benchmarkdatensatz erklären

- Parameter d. Methode diskutieren und u.U. bestimmen

3.3. Echtzeitumgebung

- Richtigen Datensatz beschreiben

4. Realisierung

4.1. Trainieren der Objektdetektoren

LabelImg Section, Auswahl der cloud Warum lokal trainieren -> Auf Tabelle Bezug nehmen, aus kosten gründen... -> Absprache mit DH und Markus datensatz fertig machen auf cloud fertig trainieren und migrieren aus zeit grüünden können wir den datensatz nicht größer machen markus Mail PC merge textit anpassungen robin kredit karte die drone! rcnn raus

arbeit ins knowledge portal, singapur auch linked In post

Problem: Merfah Detektionen werden alle als positiv angesehen

4.2. Drohnen Anbindung

4.3. Dashboard Entwicklung

- Herausforderungen bei der Drohne
- Trainieren auf die richtigen Datensätze
- Webapplikation

5. Ergebnisse

- Ergebnisse auswerten

6. Bewertung

- Ergebnisse bewerten und diskutieren
- Was liefert die Arbeit, was bisher noch nicht bekannt war?

7. Zusammenfassung und Ausblick

- Klare Darstellung, was die Arbeit geliefert hat
- Ca. 2-4 Anpunkte: Zukünftige Ziele

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] MathWorks. *Deep Learning: Drei Dinge, die Sie wissen sollten.* MathWorks, 2019. URL: https://de.mathworks.com/discovery/deep-learning.html (Einsichtnahme: 12. 10. 2019).

[3] doks. innovation. *inventAIRyX.* doks. innovation Homepage, 2019. URL: https://www.doks-innovation.com/inventairy-x/ (Einsichtnahme: 26. 10. 2019).

[4] Philippe Lucidarme. *Simplest perceptron update rules demonstration.* Homepage Blog, 2017. URL: https://www.lucidarme.me/simplest-perceptron-update-rules-demonstration/ (Einsichtnahme: 26. 10. 2019).

[5] Wikipedia. *Deep Learning.* Wikipedia, 2019. URL: https://de.wikipedia.org/wiki/Deep_Learning (Einsichtnahme: 27. 01. 2019).

[6] 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).

[7] 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).

[8] 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).

[9] Ronny Restrepo. *Calculating the derivative of Tanh: step by step.* Ronny Restrepo Homepage, 2017. URL: http://ronny.rest/media/blog/2017/2017_08_16_tanh/tanh_v_sigmoid.jpg (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] 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).

[12] 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).

[13] Leonadro Araujo Santos. *Pooling Layer: Introduction.* GitBook, 2018. URL: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/pooling_layer.html (Einsichtnahme: 26. 10. 2019).

[14] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Chang-Yang Fu, Alexander C. Berg. „SSD: Single Shot MultiBox Detector“. Diss. 2016-12-29. URL: https://arxiv.org/pdf/1512.02325.pdf (Einsichtnahme: 02. 11. 2019).

[15] Andrew Ng. *Anchor Boxes.* Coursera, 2019. URL: https://www.coursera.org/lecture/convolutional-neural-networks/anchor-boxes-yNwO0 (Einsichtnahme: 02. 11. 2019).

[16] MathWorks. *vgg16.* MathWorks Homepage, 2019. URL: https://de.mathworks.com/help/deeplearning/ref/vgg16.html;jsessionid=bf0fea41a0c7700184672711881f (Einsichtnahme: 02. 11. 2019).

[17] 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).

[18] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, Ali Farhadi. „You Only Look Once: Unified, Real-Time Object Detection“. Diss. 2019-11-10. URL: https://arxiv.org/pdf/1506.02640.pdf (Einsichtnahme: 10. 11. 2019).

[19] Joseph Redmon, A. F. „YOLOv3: An Incremental Improvement“. In: (2018). URL: https://pjreddie.com/media/files/papers/YOLOv3.pdf.

[20] Renu Khandelwal. *COCO and Pascal VOC data format for Object detection*. Towards Data Science, 2019. URL: https://towardsdatascience.com/coco-data-format-for-object-detection-a4c5eaf518c5 (Einsichtnahme: 02.02.2020).

[21] 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).

[22] 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/htmldoc/voc.html#SECTION00030000000000000000 (Einsichtnahme: 08.02.2020).

[23] 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/htmldoc/devkit_doc.html#SECTION00030000000000000000 (Einsichtnahme: 08.02.2020).

[24] NVIDIA. *TRAIN MODELS FASTER*. NVIDIA Homepage, 2020. URL: https://developer.nvidia.com/cuda-zone (Einsichtnahme: 09.02.2020).

[25] PyTorch. *GET STARTED*. PyTorch Homepage, 2020. URL: https://pytorch.org/get-started/locally/ (Einsichtnahme: 09.02.2020).

[26] TechPowerUp. *GPU Specs Database*. TechPowerUp Homepage, 2020. URL: https://www.techpowerup.com/gpu-specs/ (Einsichtnahme: 09.02.2020).

[27] Harald Bögeholz. *Künstliche Intelligenz: Architektur und Performance von Googles KI-Chip TPU*. Heise Online, 2017. URL: https://www.heise.de/newsticker/meldung/Kuenstliche-Intelligenz-Architektur-und-Performance-von-Googles-KI-Chip-TPU-3676312.html (Einsichtnahme: 09.02.2020).

[28] Google Cloud. *Systemarchitektur*. Google Cloud Dokumentation, 2020. URL: https://cloud.google.com/tpu/docs/system-architecture (Einsichtnahme: 09.02.2020).

[29] Google Cloud. *Cloud TPU*. Google Cloud Homepage, 2020. URL: https://cloud.google.com/tpu/?hl=de (Einsichtnahme: 09.02.2020).

[30] 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).

[31] FloydHub. *Home*. FloydHub Documentation, 2020. URL: https://docs.floydhub.com/ (Einsichtnahme: 15. 02. 2020).

[32] 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: 15. 02. 2020).

[33] Tarang Shah. *Measuring Object Detection models-mAP-What is Mean Average Precision?* Tarang Shahs Blog, 2018. URL: https://tarangshah.com/blog/2018-01-27/what-is-map-understanding-the-statistic-of-choice-for-comparing-object-detection-models/ (Einsichtnahme: 15. 02. 2020).

A. Anhang

Abbildungen