Echtzeit-Kollisionsvorhersage für autonome Multicopter basierend auf Deep Learning gestützter Tiefenwahrnehmung

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Markus Gross

[redacted]

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüferin: Prof. Dr.-Ing. Sabine Schumann

Zweitprüfer: Dr. rer. nat. Tobias de Taillez

Hamburg, 25. 02. 2020

Zusammenfassung

Ziel dieser Arbeit ist die Entwicklung einer Kollisionsvorhersage für autonome Multicopter. Hierfür wird ein Deep Learning Algorithmus einer aktuellen Publikation reimplementiert und adaptiert, um dessen Berechnung als Grundlage zur Interpretation einer potentiellen Kollision zu nutzen. Dabei werden differente Adaptionen verglichen und deren Ergebnisse evaluiert. Die Kernaufgabe besteht darin, die Berechnungszeit des Algorithmus dahingehend zu reduzieren, dass eine Anwendung in Echtzeit sichergestellt ist und parallel dazu eine Berechnungsqualität gewährleistet wird, mit der Kollisionen prinzipiell erkannt werden können.

Schlüsselwörter

Maschinelles Lernen, Deep Learning, Künstliches Neuronales Netz, Convolutional Neural Network, Kollisionsvorhersage, Autonomie, Echtzeitsystem, Multicopter, Optical Flow, Transfer Learning, Depth Map, Encoder - Decoder, Keras, TensorFlow

Abstract

The aim of this work is to develop a collision prediction for autonomous multicopters. For this purpose, a deep learning algorithm of a current publication is reimplemented and adapted to use its computation as a basis for the interpretation of a potential collision. Different adaptations are compared and their results are evaluated. The core task is to reduce the computation time of the algorithm to ensure a real-time application and at the same time to guarantee a computation quality that allows collisions to be detected.

Keywords

Machine Learning, Deep Learning, Artificial Neural Network, Convolutional Neural Network, Collision Prediction, Autonomy, Real-Time System, Multicopter, Optical Flow, Transfer Learning, Depth Map, Encoder - Decoder, Keras, TensorFlow

Inhaltsverzeichnis

1 Einleitung **5**

1.1 Motivation 5

1.2 Struktur der Arbeit 6

2 Grundlagen **7**

2.1 Multicopter 7

2.2 RGB-Bilder 8

2.3 Depth Maps 8

2.4 Künstliche Neuronale Netze 9

2.4.1 Allgemeiner Aufbau 9

2.4.2 Gewicht und Aktivierungsfunktion 10

2.4.3 Loss Function und Initializer 11

2.4.4 Backpropagation 12

2.4.5 Optimizer und Learning Rate 13

2.4.6 Batch Size und -Normalization 15

2.4.7 Epochen, Iterationen, Parameter und Hyperparameter 16

2.4.8 Trainings-, Validierungs- und Testdatensatz 16

2.5 Convolutional Neural Networks 17

2.6 Transfer Learning 20

2.7 Encoder - Decoder Architektur 21

2.8 Optical Flow 22

3 Problemstellung **24**

4 Konzeption **25**

4.1 Depth Map Erzeugung 25

4.2 Ablauf der Kollisionsvorhersage 25

5 Implementation **29**
5.1 Entwicklungs- und Testumgebung 29
5.1.1 Python 29
5.1.2 SPYDER 29
5.1.3 MatLab 30
5.1.4 TensorFlow und Keras 30
5.2 Hardware 31
5.2.1 Verwendeter Multicopter 31
5.2.2 Implementationsrechner 31
5.2.3 Trainingsrechner 32
5.3 Datenaufbereitung 32
5.4 Depth Map 34
5.4.1 Datenvorbereitung 34
5.4.2 Dense Depth Architektur 36
5.4.3 Experiment Design 36
5.4.4 Trainingsablauf 38
5.4.5 Ergebnisse 41
5.5 Kollisionsvorhersage 46

6 Evaluation **53**

7 Fazit und Ausblick **56**

Abbildungsverzeichnis **58**

Literaturverzeichnis **60**

1 Einleitung

1.1 Motivation

Autonome Fortbewegung ist eines der großen, technologischen Themen unserer Zeit. Sie ist sowohl im öffentlichen Verkehr wiederzufinden (Autos, Shuttles, Flugzeuge), als auch im Güterverkehr (LKWs, Züge, Schiffe) und der Industrie (Multicopter, Transportfahrzeuge). Die Vorteile dieser Entwicklung sind vielzahlig. Neben erhöhter Transporteffizienz [KPMG20] führt diese Enwicklung auch zu reduzierten Verkehrsunfällen und CO2-Emmissionen [Ohio20]. Letzteres spielt im Sinne des Klimawandels eine zunehmend große Rolle und es besteht ein großes Interesse darin, die Forschung dieser Entwicklung weiter voranzutreiben.

Die Basis autonomer Systeme bilden zumeist hochkomplexe Algorithmen, die ein Konglomerat sensorischer Informationen verarbeiten, um die ihnen übertragene Aufgabe mit einem höchstmöglichen Maß an Sicherheit auszuführen. Bei diesen hochdimensionalen Problemen arbeiten Neuronale Netze äußerst performant. Für aktuelle Lösungen ist diese große Menge an Sensordaten notwendig, um ein angemessenes Ergebnis erzielen zu können. Jedoch entstehen zunehmend Anwendungsfälle, die eine Nutzung bestimmter Sensoren und Technologien nicht erlauben oder gänzlich überflüssig machen. Somit stehen in diesen Anwendungsfällen nicht ausreichend Daten zur Verfügung.

Um in solchen Fällen auf Ausweichmöglichkeiten zurückgreifen zu können, werden stetig intelligentere Lösungen entwickelt, welche derartige Probleme beheben sollen. In autonomen Fahrzeugen werden unterschiedliche Sensoren, wie z.B. LiDar (light detection and ranging) für die Tiefenwahrnehmung genutzt, welche notwendig ist, um die Entfernung aller Objekte, relativ zum Fahrzeug, zu bestimmen. Die Tiefenwahrnehmung ist ausschlaggebend für eine Kollisionsvorhersage und steht damit in

direktem Zusammenhang mit der sicheren Nutzung eines solchen Fahrzeugs.

In dieser Arbeit soll eine Kollisionsvorhersage im Rahmen der *WARGdrones GmbH* erarbeitet werden. Das Unternehmen arbeitet in einem professionllen Kontext mit autonomen Multicoptern, die aufgrund von konstruktions- und gewichtsbedingtem Platzmangel nicht auf derartige Sensoren wie LiDar zurückgreifen können. Die Vorhersage soll aus lediglich einer Kamera (monokular) generiert werden, welche in Echtzeit anwendbar sein soll. Hierfür werden aktuelle Forschungsergebnisse aus einem Teilbereich des Maschinellen Lernens, dem Deep Learning, implementiert, adaptiert und weiter erforscht. Zusätzlich zum möglichst geringen Gewicht besteht die Problematik, dass Multicopter per se keine langsamen Gefährte sind. Sie erfordern durch ihre Geschwindigkeit ein gewisses Mindestmaß an Steuerungsgeschwindigkeit, um Kollisionen zu vermeiden. Die Echtzeitfähigkeit der hier erforschten und adaptierten Algorithmen steht daher gleichsam im Fokus.

1.2 Struktur der Arbeit

Nachdem das aktuelle Kapitel eine grundlegende Motivation für die Arbeit dargelegt hat, werden in Kapitel 2 die Grundlagen erörtert, die für diese Arbeit notwendig sind. Vom prinzipiellen Aufbau eines Multicopters, Kamera-generierten RGB-Bildern und Depth Maps, die sich daraus entwickeln lassen, hin zu Künstlichen Neuronalen Netzen und deren Funktionsweise. Zudem werden Technolgien und Architekturen zur Bildverarbeitung beschrieben, die im späteren Verlauf dieser Arbeit verwendet werden. Im darauffolgenden Kapitel werden die Probleme geschildert, mit der sich diese Arbeit befasst, um anschlißend die Konzepte in Kapitel 4 zu erläutern, welche diese Probleme lösen sollen. In der Implementation (Kapitel 5) wird die Umsetzung der Lösungsansätze beschrieben und deren Ergebnisse präsentiert, um eben diese in Kapitel 6, der Evaluation, fachgerecht zu bewerten. Zum Schluss folgt in Kapitel 7 das Fazit und ein Ausblick.

2 Grundlagen

Im folgenden Kapitel werden alle Inhalte erläutert, die für ein grundlegendes Verständnis dieser Arbeit notwendig sind. In den darauffolgenden Kapiteln werden diese Grundlagen genutzt, um Probleme, Lösungen und Vorgehensweisen zu erörtern.

2.1 Multicopter

Der Begriff Multicopter ist ein Überbegriff für Luftfahrzeuge mit mehr als einem Propeller, wobei der Begriff ein Neologismus aus dem lateinischen „multi“ und dem griechischen „pteron“ ist, was somit für „Mehrfachflügler“ steht.

Ein Multicopter besitzt 4 Freiheitsgrade, welche sich drosseln (*throttle*), gieren (*yaw*), neigen (*pitch*) und rollen (*roll*) nennen, wobei der Throttle das vertikale Auf- und Absteigen des Multicopters beschreibt, siehe Abbildung 2.1.

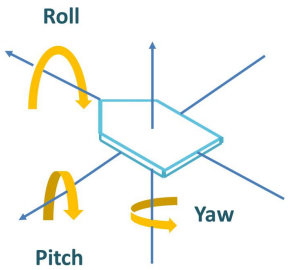

Abbildung 2.1: Freiheitsgrade eines Multicopters [Ameb20]

Bei konstantem Throttle führt ein Anstellen des Multicopters um seine Querachse (Pitch nach vorne/zurück) zu einer Umwandlung des Throttle-Schubs in Vortrieb. Der verlorene Throttle muss entsprechend wieder hinzugefügt werden, um eine konstante Flughöhe zu gewährleisten. Multicopter besitzen häufig eine Kamera, dessen Bildaufnahmen im folgenden Abschnitt erklärt werden.

2.2 RGB-Bilder

Der klassische Farbraum für Bilder ist der RGB-Farbraum, wobei das Akronym RGB für die drei Grundfarben Rot, Grün und Blau steht. Aus diesen Farben kann im richtigen Mischungsverhältnis jede beliebige Farbe erzeugt werden. Diesen Vorteil machen sich digitale RGB-Bilder zunutze. Jedes Pixel eines digitalen Bildes hat drei Channel, einen für die jeweilige Grundfarbe. Die Bezeichnung „6x6x3“im Beispiel von Abbildung 2.2 bedeutet, dass dies ein Bild mit folgender *Auflösung* ist: eine Breite von 6 Pixel, einer Höhe von 6 Pixel und 3 Channel. Durch eine additive Farbmischung [Simo13], dem aufaddieren aller Farbwerte der übereinanderliegenden Pixel, kann die gewünschte Farbe für jedes Pixel erzeugt werden.
Einzelne RGB-Bilder eines Videos, die eine Kamera konsekutiv erstellt, werden auch als *Frames* bezeichnet.

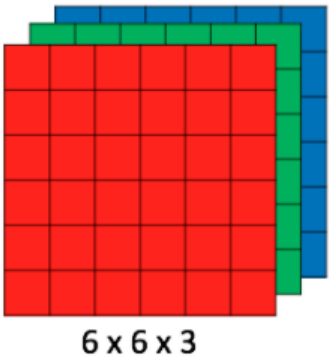

Abbildung 2.2: Digitale RGB-Bilder [Prab18]

2.3 Depth Maps

Depth Maps sind Bilder, die Informationen darüber enthalten, wie weit jedes Objekt vom Aussichtspunkt entfernt ist. Ihre Pixel-Werte entsprechen dabei der Entfernung in Meter. Zu Darstellungszwecken kann diese Entfernungsangabe in ein Graustufenbild umgerechnet werden, was bedeutet, dass die resultierenden Depth Maps exakt einen Channel besitzen. Eine Möglichkeit, Depth Maps zu erzeugen, ist mit Infrarot-Sensoren [Pagl14]. In Abbildung 2.3 ist beispielhaft eine Depth Map aufgeführt. Je heller ein Pixel ist, desto näher ist dieses Pixel, bzw. der Teil des Objekts, welcher durch das Pixel repräsentiert wird.

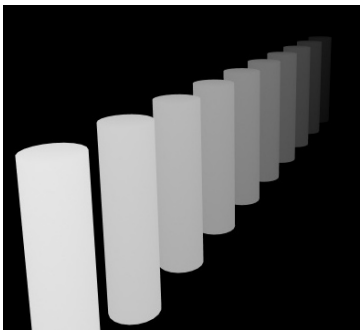

Abbildung 2.3: Depth Map Beispiel [Blen20]

2.4 Künstliche Neuronale Netze

Die Basis für *Künstliche Neuronale Netze* (KNNs) wurde bereits 1943 geschaffen [McCu43]. Allerdings sorgten unzureichende Rechenkapazitäten und zu hohe Erwartungen in den 1970er und '80er Jahren für zwei sogenannte KI - Winter (Künstliche Intelligenz - Winter), in welchen die Finanzierung und Forschung von KI - Projekten stark schwand. Erst die zweckentfremdete Anwendung von Grafikkarten im Jahr 2008 durch NVIDIA [Nvid20] führte zum großen Durchbruch. Mit der Veröffentlichung von CUDA [CUDA20] wurden Matrixoperationen erstmals parallelisierbar, welche für eine effiziente Berechnung in KNNs notwendig sind.

2.4.1 Allgemeiner Aufbau

Ein Neuronales Netz besteht aus mehreren Einheiten, die der Informationsverarbeitung dienen. Diese Einheiten werden *Neuronen* genannt, als Anlehnung an die Informationsverarbeitungseinheiten im menschlichen Gehirn. Jeder Knoten in Abbildung 2.4 entspricht so einem Neuron. Eine vertikale Ansammlung mehrerer Neuronen wird als Layer bezeichnet, wobei mehrere in Reihe geschaltete Layer ein Künstliches Neuronales Netz ergeben. Der erste und letzte Layer bildet dabei respektive den *Input*- und *Output* Layer. Dazwischen können sich mehrere Layer befinden, die gemeinsam als *Hidden Layer* bezeichnet werden. Zudem kann die Anzahl der Neuronen in jedem Layer variieren. Besitzt ein KNN mehr als einen Hidden Layer, so wird es als *Deep Neural Network* bezeichnet. Während der Entwicklung wird eine Instanz des KNN auch als *Modell* (*Model*) bezeichnet.

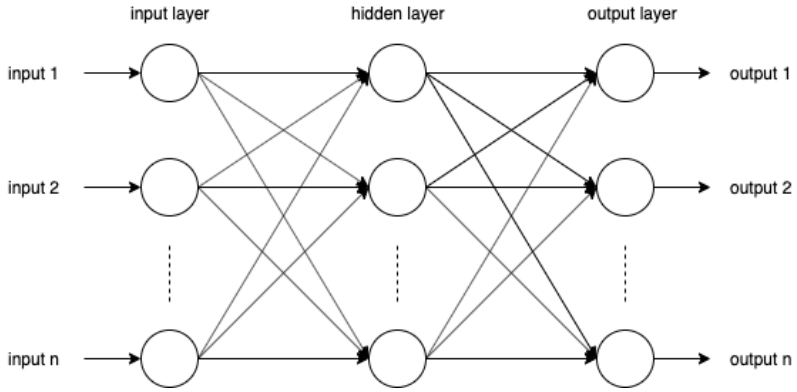

Abbildung 2.4: Künstliches Neuronales Netz

2.4.2 Gewicht und Aktivierungsfunktion

Die Verbindung zwischen zwei Neuronen ist das sogenannte *Gewicht*, in Form eines Skalars. Diese Skalare werden bei der Erstellungs des KNN zufällig, meist in einem Intervall zwischen 0 und 1, generiert. Wenn der Input Layer nun die Eingabe in Form eines Inputvektors bekommt, werden die darin enthalten Skalare mit den verbundenen Gewichten multipliziert und das Ergebnis wird an das nächste Neuron weitergegeben. In Abbildung 2.4 ist zu sehen, dass jedes Neuron mit jedem Neuron des nächsten, bzw. vorherigen Layers verbunden ist. Dies bedeutet, dass jedes Neuron im Hidden- und Output Layer mehrere Skalare als Input bekommt, welche aufaddiert werden. Jedes Neuron (außer die des Input Layers) besitzen eine sogenannte *Aktivierungsfunktion*. Die aufaddierten Skalare werden der Aktivierungsfunktion übergeben, welche entsprechend einer austauschbaren mathematischen Funktion eine nichtlineare Verarbeitung durchführt und somit darüber entscheidet, mit welcher Intensität diese Information an das Neuron im nächsten Layer weitergegeben wird. Es folgt eine schematische Darstellung für die Informationsverarbeitung eines Neurons im Hidden Layer (Abb. 2.5).

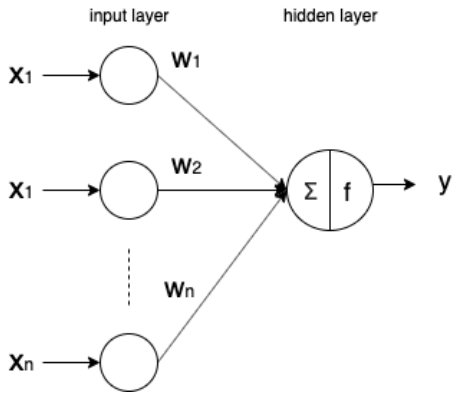

Abbildung 2.5: Output eines Neurons

Somit lässt sich der Ouput eines Neurons wie folgt definieren:

$$y_{\text{neuron}} = f_{\text{act}}(\sum_{i=1}^{n} x_i w_i) \tag{2.1}$$

2.4.3 Loss Function und Initializer

Der Ouput eines Neurons wird an alle Neuronen des nächsten Layers weitergegeben, bis der Output-Layer erreicht ist, wonach der generelle Output des KNN folgt, an dem das Ergebnis des KNN abgelesen wird. Dieser wird auch als *Vorhersage* (*prediction*) bezeichnet. Wenn die Trainingsdaten das gesamte Netzwerk durchlaufen, wird dies als *Vorwärtsausbreitung* (*Forwardpropagation*) bezeichnet, da die Informationen "nach vorne" durch das KNN weitergegeben werden. Die *Inference Time* beschreibt dabei die Zeit, die das Model benötigt, um aus den Input-Daten eine Prediction zu generieren. Beim Training eines KNN werden Daten in Form von Input und gewünschtem Output benötigt, letzteres wird als *Ground Truth Label*, kurz Label, bezeichnet. Im Bereich des maschinellen Lernens wird ein Training mit vorhandenem Label als *Supervised Learning* [Mül17] bezeichnet. Gemeinsam werden Input und Label als *Sample* bezeichnet. Dabei entspricht das Label derjenigen Information, die in der späteren Nutzung des KNN gesucht wird. Am Anfang des Trainings werden die Gewichte durch den *Initializer* zufällig generiert. Er bestimmt, in welchem Intervall und auf welche Art und Weise die Gewichte initialisiert werden. Dass die Gewichte zufällig initialisiert sind führt dazu, dass der berechnete Output nicht zu dem dazugehörigen Label passt. Mithilfe einer *Loss Function* kann ein *Fehler* (*Loss* oder *Error*) bestimmt werden, um zu vergleichen, wie gut oder schlecht das vorhergesagte

Ergebnis in Relation zum Label ist. Im besten Fall ist der Loss = 0, dann würde die Prediction des KNN gänzlich mit dem Label übereinstimmen. Eine häufig verwendete Loss Function im Bereich des Deep Learnings ist die *Mittlere Quadratische Abweichung* (*mean squared error*). Diese ist in Gleichung 2.2 dargestellt, wobei x dem Output des Models entspricht und L dem Label.

$$f_{\text{loss}}(x) = (x - L)^2 \tag{2.2}$$

Die in dieser Arbeit verwendete Loss Function bildet geometrische Eigenschaften ab und ist somit speziell an das Problem der Tiefenwahrnehmung angepasst (siehe Unterabschnitt 5.4.4).

2.4.4 Backpropagation

Mithilfe des Losses können die Gewichte während des Trainings aktualisiert werden. Die Grundlage hierfür hat 1974 Paul Werbos mit der *Backpropagation* [Werb19] geschaffen, welche später von Rumelhart et al. [Rume86] zur *Backpropagation of Error* weiterentwickelt wurde. Diese wird umgangssprachlich als Backpropagation bezeichnet und wird im Deep Learning dafür genutzt, die Gewichte eines KNN zu aktualisieren. Dabei werden die Gewichte derart angepasst, dass während der nächsten Prediction das Label in Relation minimal korrekter errechnet wird. Die Backpropagation ist der zentrale Mechanismus, mit dem versucht wird, den Loss weitestgehend zu reduzieren, um damit die Genauigkeit der Prediction zu maximieren. Das KNN *lernt* durch die Backpropagation. Schematisch kann der gesamte Berechnungszyklus der Forwardpropagation, der Loss-Berechnung und der Backpropagation wie folgt dargestellt werden (Abb. 2.6):

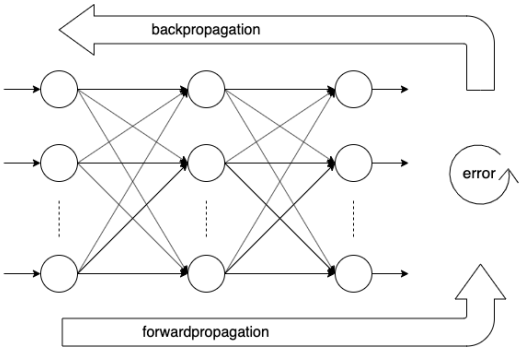

Abbildung 2.6: Berechnungszyklus eines KNN

2.4.5 Optimizer und Learning Rate

Der Prozess der Backpropagation kann als globale Optimierung betrachtet werden, bei der die Gewichte so aktualisiert werden sollen, dass das Ergebnis der Loss Function minimiert wird (Abbildung 2.7).

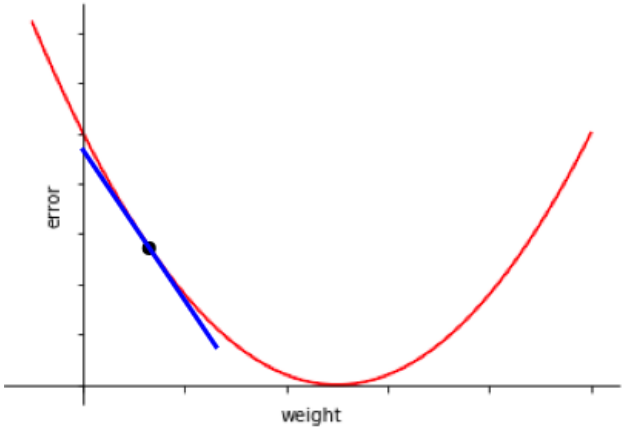

Abbildung 2.7: Aktualisierung der Gewichte

Es gibt unterschiedliche Möglichkeiten, die Gewichtsanpassung während der Backpropagation durchzuführen. Ist diese Anpassung allerdings zu groß, kann es vorkommen, dass das lokale Minimum übersprungen wird (Abbildung 2.8).

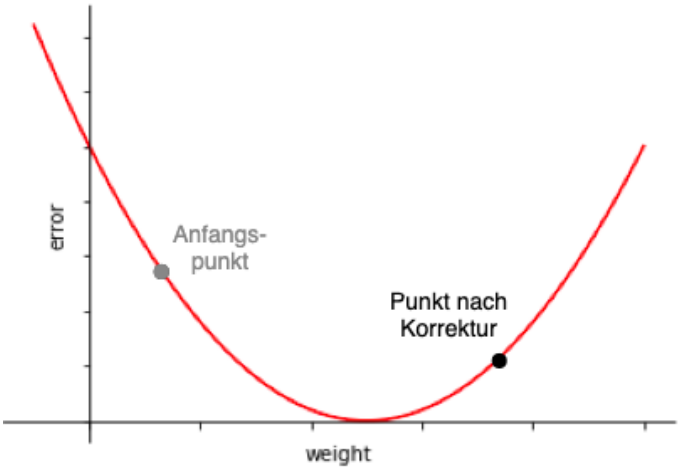

Abbildung 2.8: Überspringen des lokalen Minimums

Gewünscht ist jedoch eine schrittweise Annäherung an das lokale Minimum, wie in Abbildung 2.9 zu sehen ist.

Abbildung 2.9: Schrittweise Annäherung an lokales Minimum

Hierfür werden sogenannte *Optimizer* eingesetzt, der bekannteste unter ihnen ist der *Gradient Descent* [Rude17], welcher mithilfe des Gradienten arbeitet (siehe Abbildung 2.7). Er korrigiert das aktuelle Gewicht minimal auf linearem Weg zum Zielgewicht. Dies ist möglich, da durch die Backpropagation der Grad der Beteiligung jedes Neurons am Gesamt-Loss bekannt ist. Die Stärke dieser Korrektur wird mithilfe der sogenannten *Learning Rate* bestimmt. Die Learning Rate wird in Form eines Skalars mit dem Betrag des Gradientenvektors multipliziert. Dadurch ist die Größe der Schrittweite proportional zur Entfernung des Minimums (siehe Abbildung 2.9), wodurch ein Überspringen verhindert wird.

Diese Annäherung geschieht allerdings häufig mit unzureichender Geschwindigkeit, da unter Umständen die Schrittweite nicht groß genug ist, um ein effizientes Konvergieren zu ermöglichen. Zudem ergibt sich durch die Anwendung des Gradient Descent der Nachteil, dass nicht mit Sicherheit gesagt werden kann, ob das gefundene Minimum lokal oder global ist, da dies davon abhängt, wie die Gewichte initialisiert wurden. Daher gibt es fortgeschrittenere Optimizer, die den bisherigen Verlauf des Trainings einbeziehen und die Schrittweite und auch die vektorielle Richtung der Gewichtsänderung adaptiv anpassen. Solch ein Optimizer wird in dieser Arbeit verwendet, siehe Unterabschnitt 5.4.4

2.4.6 Batch Size und -Normalization

Wie häufig die Gewichte aktualisiert werden, lässt sich über die *Batch Size* bestimmen. In den bisherigen Besipielen wurde von einer Batch Size der Größe 1 ausgegangen, bei der die Gewichte nach jedem Input aktualisiert werden. Allerdings verlängert diese Vorgehensweise die Trainingszeit bei großen Datensätzen erheblich, weil dazu ein neues Datenpaket von der Prozessor-Ebene (CPU-Ebene) zur Grafikkarte (GPU) transportiert werden muss. Diese Kommunikation ist um ein vielfaches langsamer als die Berechnung auf der GPU. Es bietet sich daher an, mehrere Berechnungen zu bündeln. Die Menge an Daten, die gebündelt werden kann, hängt ausschließlich von der Größe des GPU-Zwischenspeichers (GPU-RAM) ab. Deshalb wird der Loss für mehrere Inputs (einen Batch) berechnet. Das bedeutet, dass die Loss Function weiterhin für jede Prediction ausgeführt wird, jedoch wird für die Backpropagation das arithmetische Mittel über alle Losses eines Batches gebildet. Der Mean Squared Error aus Unterabschnitt 2.4.3 würde in diesem Fall für eine Batch Size der Größe n wie folgt definiert werden:

$$f_{\text{loss}}(x) = \sum_{i=1}^{n} (x_i - L_i)^2 \frac{1}{n} \tag{2.3}$$

Nach [Kesk17] eignen sich kleinere Batch Sizes besser für flache Minima, wohingegen sich große Batch Sizes für steile Minima eignen. Im Allgemeinen ist die Wahl der richtigen Batch Size abhängig vom vorliegenden Problem.

Im Bereich des maschinellen Lernens ist es häufig der Fall, dass die Menge der Trainingsdaten, die nötig ist, um brauchbare Ergebnise zu erzielen, zu groß ist, um sie dem Computer auf einmal zu übergeben. Dies ist ein weiter Grund für die Implementation einer Batch Size.

Eine Normalisierung der Batches (*batch normalization*) ist notwendig, da der Betrag des Ouputs eines Neurons die nachfolgenden Neuronen beeinflusst. Ist dieser Output beispielsweise sehr groß, ist der Output der nachfolgenden Neuronen mit hoher Wahrscheinlichkeit auch sehr groß. Analog gilt dies auch für sehr kleine Outputs von Neuronen. Umso tiefer das KNN ist, desto stärker wirkt sich dieser Effekt auf nach-

folgende Layer aus und hat somit auch Einfluss auf die Anpassung der Gewichte bei der Backpropagation. Der Gradient kann dadurch explodieren (*exploding gradient*) oder schwinden (*vanishing gradient*) [Bjor18]. Bei der Batch Normalization werden die Outputs der Neuronen meist auf Werte zwischen -1 und 1 skaliert. Dadurch wird gewährleistet, dass jedes Neuron einen Ouput im gleichen Intervall an die nachfolgenden Neuronen weitergibt. Dieser Mechanismus schränkt die Effekte eines vanishing oder exploding gradient stark ein.

2.4.7 Epochen, Iterationen, Parameter und Hyperparameter

Eine Epoche beschreibt die Forward- und Backpropagation des gesamten, vorliegenden Datensatzes. Die Definition dieses Ausdrucks ist wichtig, da KNNs während des Trainings häufig einer Vielzahl von Epochen ausgesetzt sind. Die Anzahl der Iterationen beschreibt dabei die Anzahl der Batches, die notwendig sind, um exakt eine Epoche zu durchlaufen.

Der Unterschied zwischen *Parametern* und *Hyperparametern* eines KNN ist, das Hyperparameter Werte sind, die das Netz lediglich von „außerhalb“ beeinflussen, sie haben nichts mit dem Netz per se zu tun. Beispiele hierfür sind die Learning Rate (Unterabschnitt 2.4.5) oder die Batch Size (Unterabschnitt 2.4.6). Parameter hingegen sind Bausteine und Eigenschaften des Netzes [Rash17]. Beispiele dafür sind Anzahl der Hidden Layer (Unterabschnitt 2.4.1) oder die Art der Aktivierungsfunktion (Unterabschnitt 2.4.2). Es bietet sich an, Parameter und Hyperparameter in mehreren Zyklen anzupassen, um eine optimale Lösung für eine Problemstellung zu finden.

2.4.8 Trainings-, Validierungs- und Testdatensatz

Der Datensatz, der zum Erstellen eines KNN zur Verfügung steht, wird in je einen Trainings-, Validierungs- und Testdatensatz unterteilt.

- Trainingsadatensatz
  Dieser Datensatz dient dazu, das Model zu trainieren, bzw. dessen Gewichte anzupassen. Hiermit findet das eigentliche Lernen statt.

- Validierungsdatensatz
  Mit dem Validierungsdatensatz werden die Hyperparameter eines KNN angepasst. Nach jeder Epoche durchläuft dieser Datensatz das Model, um zu prüfen, ob sich der berechnete Loss dem Loss des Trainingsdatensatzes ähnelt. Ist die Differenz dieser beiden Werte zu groß, findet wahrscheinlich eine *Überanpassung* (*overfitting*) oder *Unteranpassung* (*underfitting*) statt [Jabb14].

- Testdatensatz
  Mit diesem Datensatz wird die Fähigkeit der Generalisierung, bzw. die letztendliche Genauigkeit des Models geprüft. Der Testdatensatz durchläuft das KNN während des gesamten Trainingsprozesses nicht, weshalb dessen Aussage über die Genauigkeit des Models entscheidend ist.

Der Unterschied zwischen Validierungs- und Testdatensatz ist, dass der Validierungsdatensatz im Bildungsprozess des Models zur Auswahl von Hyperparametern verwendet wird. Er kann daher als Teil des Trainingsdatensatzes betrachtet werden. Der Testsatz dient dazu, das Model endgültig zu bewerten.

Vorliegende Datensätze werden üblicherweise im Verhältnis 8:1:1 in Trainings-, Validierungs- und Testdatensatz aufgeteilt.

2.5 Convolutional Neural Networks

Ein *Convolutional Neural Network* (CNN) gehört im Bereich des Deep Learnings zur Bildverarbeitung. Alle Eigenschaften und Prinzipien der in Abschnitt 2.4 erwähnten KNNs gelten auch bei CNNs. Jedoch besteht der Inputvektor in diesem Fall aus einem digitalen RGB-Bild (siehe Abschnitt 2.2). Generell kann der Prozess eines CNN in zwei Teile untergliedert werden, der Feature Extraction und der darauffolgenden Klassifizierung.

Die *Feature Extraction* dient dazu, das Input Bild so vorzubereiten, sodass es bei der anschließenden Klassifizierung einfacher verarbeitet werden kann [LeCu98]. Ginge es zum Beispiel darum, einen Hund auf einem Bild zu erkennen, würden hier die Eigenschaften von Beinen, Rute, Hundekopf und -körper extrahiert werden.

Bei der sogenannten *Convolution*, der *Faltung* im mathematischen Sinne, werden *Filter* (*kernel*) genutzt, um *Feature Maps* zu erstellen. Der Kernel ist prinzipiell aufgebaut wie ein digitales RGB-Bild, jedoch ist er kleiner als das Input Bild, das er bearbeitet. Lediglich dessen Anzahl der Channels entspricht der Anzahl der Channels im Input Bild.
Um die Feature Maps zu erstellen, fährt der Kernel schrittweise über das Input Bild, abwechselnd horizontal und vertikal wie in Abbildung 2.10 zu sehen ist.

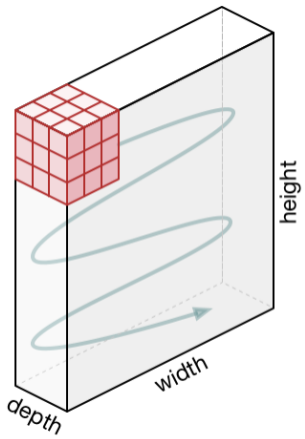

Abbildung 2.10: Kernel Bewegung [Towa18]

Um wie viele Pixel der Kernel sich dabei bewegt, bestimmt der sogenannte *Stride Value*. Bei einem Stride mit dem Wert 1 bewegt sich der Kernel pro Verarbeitungsschritt innerhalb einer Convolution um genau ein Pixel nach rechts, bzw. nach unten. Die Pixel-Werte des Kernels entsprechen dabei den Gewichten in einem klassischen KNN (Abschnitt 2.4), sie werden ebenfalls mit einem Initializer initialisiert. Bei jedem Schritt wird separat für jeden Channel das Skalarprodukt (siehe Gleichung 2.4) aus Kernel und dem Bereich des Bildes, den der Kernel zu diesem Zeitpunkt bearbeitet, berechnet. Anschließend wird die Summe der Skalarprodukte gebildet, dieser Wert entspricht exakt einem Pixel-Wert der Feature Map. Dieser Vorgang ist in Abbildung 2.11 beispielhaft dargestellt, wobei der „Output“ der Feature Map entspricht. Der Wert *Bias = 1* in Abbildung 2.11 kann im Rahmen dieser Erläuterung vernachlässigt werden.

$$\vec{a} \cdot \vec{b} = a_1 \cdot b_1 + a_2 \cdot b_2 + ... + a_n \cdot b_n \tag{2.4}$$

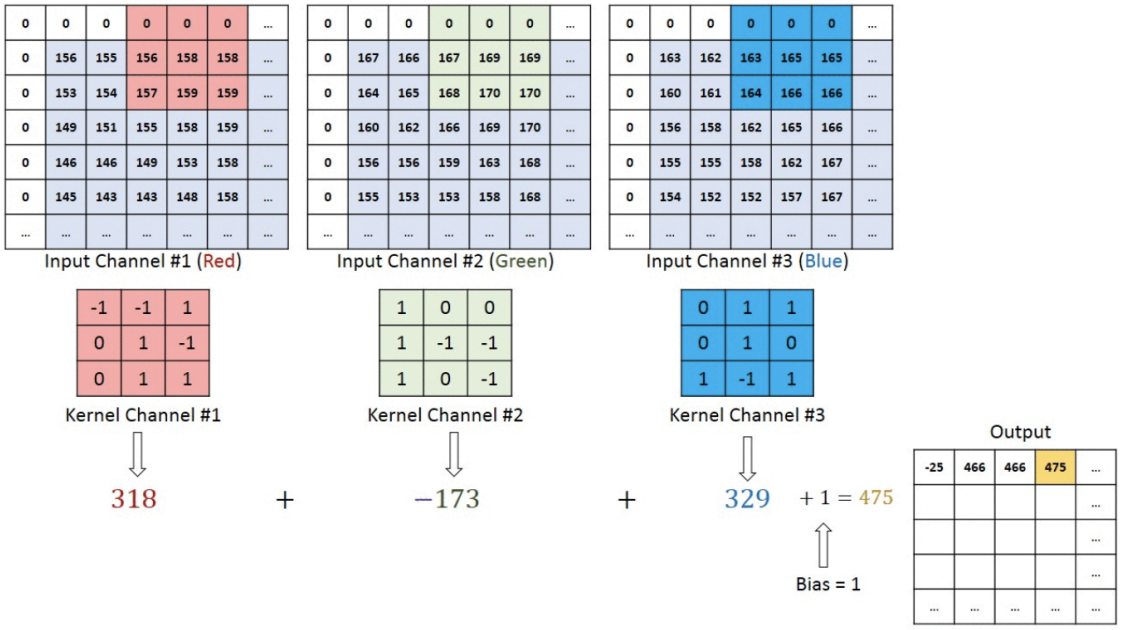

Abbildung 2.11: Convolution eines MxNx3 Bildes mit einem 3x3x3 Kernel [Towa18]

Bei einer Convolution erzeugt ein Kernel exakt eine Feature Map. Ihre Dimension (Höhe und Breite) ist abhängig vom sogenannten *Padding*, bei welchem dem Input Bild zusätzliche Dimensionen mit Null-Werten hinzugefügt werden (siehe Abb. 2.11). In den meisten Fällen wird das Padding jedoch so gewählt, dass die Feature Map die gleiche Höhe und Breite wie das Input Bild hat. Dies ist jedoch abhängig vom vorliegenden Problem. Generell entspricht die Convolution einem Layer eines CNN. In einem Convolution Layer kommen mehrere Kernels zum Einsatz, die unterschiedlich initialisiert sind, um unterschiedliche Features zu extrahieren. Nach der Feature Extraction wird die Feature Map einer Aktivierungsfunktion übergeben, um jedes Pixel auf den Grad seiner Aktivierung zu reduzieren.

Im Anschluss an einen Convolution Layer folgt ein sogenannter *Pooling Layer*. Dieser reduziert die Dimensionen der Feature Map, um redundante Informationen zusammenzufassen. Beispielsweise überspannt die Rute eines Hundes mehrere hundert Pixel, die Information des Vorhandenseins kann hingegen in einer geringeren Menge von Informationen codiert werden. Die Feature Map wiederum dient der Vorbereitung zur anschließenden Klassifizierung. Zusätzlich wird dadurch die Feature Extraction verstärkt. Der Mechanismus eines Pooling Layers entspricht dem eines Convolutional

Layers. Der Pooling Layer hat ebenfalls einen Kernel, allerdings hat dieser keine Werte. Er betrachtet lediglich seinen Input und wählt den *Maximalwert* des betrachteten Ausschnitts aus (*Max Pooling*) oder bildet den Durchschnitt aller betrachteten Werte (*Avarage Pooling*). Ein Beispiel hierfür wird in Abbildung 2.12 gezeigt.

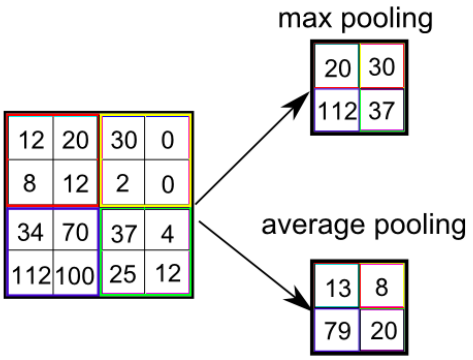

Abbildung 2.12: Pooling eines 4x4 Bildes [Quor19]

In einem CNN folgen mehrere Kombinationen Convolution- plus Pooling Layer hintereinander. In den ersten Layern werden relativ einfache Features gelernt, wie zum Beispiel Ecken und Kanten, wohingegen in den tieferen Layern komplexere Features erkannt werden, die abhängig von der jeweiligen Klassifizierung sind. Das ursprüngliche Input Bild wird in seinen Dimensionen dahingehend verändert, dass Höhe und Breite sehr gering werden, die Anzahl der Channels allerdings sehr hoch wird. Ist dieser Zustand erreicht, kann die Klassifizierung beginnen.

Nach [Rasc17] wird bei der *Klassifizierung* das Ergebnis der Feature Extraction zu einem Vektor umgewandelt, dieser Prozess wird als *flattening* bezeichnet. Ab diesem Zeitpunkt entspricht die Architektur des CNN einem KNN, wie es in Abschnitt 2.4 vorgestellt wurde. Hier werden die extrahierten Informationen genutzt, um unterschiedliche Probleme zu lösen.

2.6 Transfer Learning

Das *Transferieren* von bereits gelerntem Inhalt (*Transfer Learning*) dient der schnellen und effizienten Lösung von Klassifizierungs-Problemen, die bereits in ähnlicher Form gelöst wurden. In Abschnitt 2.5 wurde beispielhaft die Klassifizierung, bzw.

Objekterkennung, eines Hundes genannt. Wäre ein CNN benötigt, das Katzen erkennen soll, wäre es nicht sinnvoll ein gänzlich neues Netz zu entwickeln, da Hunde und Katzen viele Eigenschaften teilen. Beim Transfer Learning wird ein bereits trainiertes CNN verwendet, dessen Teil der Klassifizierung gelöscht wird, damit weitere Convolution- und Pooling Layer (siehe Abschnitt 2.5) angehängt werden können. Diese sollen die Features des neuen Problems lernen und dabei im Vorfeld die bereits gelernten Features beibehalten. Dafür werden beim Training die Layer des bereits trainierten Netzes in den meisten Fällen eingefroren, damit sich deren Gewichte nicht mehr verändern können, bzw. die gelernten Features erhalten bleiben. Anschließend wird eine neue, dem neuen Problem entsprechende Klassifizierung angehängt, wodurch die neu hinzugefügten Layer in der Lage sind, sich an das vorliegende Problem anzupassen.

In dieser Arbeit werden die Informationen, die im Objekterkenner über die Objekte selbst gespeichert sind, genutzt, um die neue Objekteigenschaft der Entfernung anzutrainieren. Somit wird Wissen aus dem Gebiet der Objekterkennung in das Gebiet der Tiefenwahrnehmung transferiert.

Durch die reduzierte, trainierbare Anzahl der Gewichte im CNN sind weniger Berechnungsschritte bei der Forward- und Backwardpropagation notwendig, was Zeit und Ressourcen spart.

2.7 Encoder - Decoder Architektur

Für die Kollisionsvorhersage ist eine Tiefenerkennung essentiell. Diese basiert auf Objekterkennung, welche wiederum über die Dimensionsreduktion durch Pooling Layer in CNNs umgesetzt wird. Diese Dimensionsreduktion komprimiert die Informationen über ihre Objekte, was als *encoding* bezeichnet wird. Das Ergebnis wird als *Latent Space Representation* bezeichnet, siehe Abbildung 2.13. Bei der Tiefenerkennung werden daraus wieder Pixel-Informationen extrapoliert. In der Latent Space Representation sind nur noch die Eigenschaften und die grobe Position im Bild codiert, für die Entfernungsschätzung wird aber eine räumlich exakte Information benötigt, was durch Convolution- und sogenannte *Upsampling* - Layer geschieht und als *decoding* bezeichnet wird. Das Gesamtergebnis ist damit ein rekonstruiertes Bild, wobei die

Anzahl der Channel über den letzten Covolution Layer definiert wird.

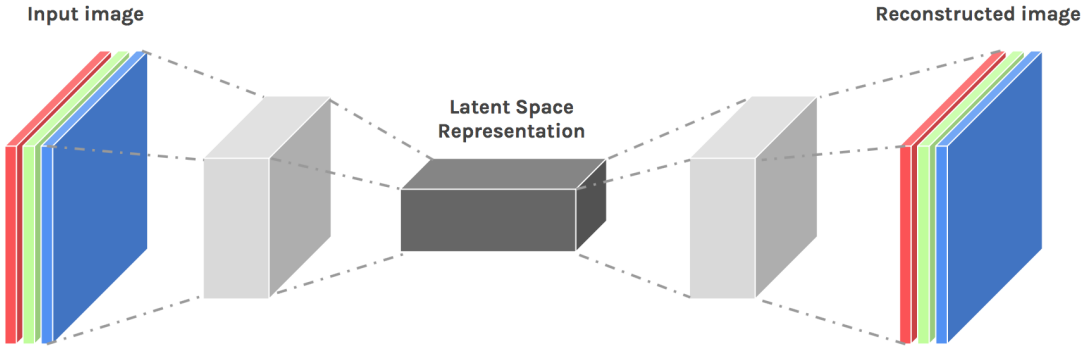

Abbildung 2.13: Enoder - Decoder Architektur für Convolutional Neural Networks [Medi18]

Eine Encoder - Decoder Architektur wird genutzt, wenn das zu bearbeitende Bild eine Modifikation erhalten soll. Dies ist mit klassischen CNNs nicht möglich, da diese lediglich Informationen aus dem Bild extrahieren, es jedoch nicht wieder rekonstruieren.

2.8 Optical Flow

Der *Optical Flow* beschreibt die Bewegung der Pixel zwischen zwei oder mehreren Frames einer Sequenz, die durch die relative Bewegung zwischen Objekt und Kamera entsteht. Das mathematische Ergebnis des Optical Flows ist ein Vektorfeld, das die Bewegung der einzelnen Pixel der aufeinanderfolgenden Frames beschreibt [Farn03].

Abbildung 2.14: Optical Flow Beispiel

Im Beispiel der Abbildung 2.14 bewegt sich das rote Pixel des 3x3 Bildes von seinem Ursprung (1,0) zu der Position (2,2). Das Pixel bewegt sich also um eine Stelle

nach rechts und zwei Stellen nach unten. Damit entspricht die Bewegung des Pixels (1,0) dem Richtungsvektor (1,2). Das entsprechende Vektorfeld ist in Abbildung 2.15 dargestellt (ausschließlich für die Bewegung des roten Pixels).

	0	1	2
0	...	(1,2)	...
1	...	...	...
2	...	...	...

Abbildung 2.15: Optical Flow Beispiel - Vektorfeld

3 Problemstellung

Ziel dieser Arbeit ist es, die Inference Time des CNN, das die Depth Map erstellt und somit die Grundlage für die Kollisionsvorhersage darstellt, so weit wie möglich zu beschränken, wobei die Depth Map eine ausreichende Qualität aufweisen muss, um Kollisionen prinzipiell erkennen zu können.

Der Grund für die geringe Inference Time ist, dass der Multicopter, dessen Kamera die Frames erzeugt, sich mit hohen Geschwindigkeiten fortbewegt und somit eine Berechnungszeit von ein paar hundertstel Sekunden notwendig ist. Um eine Kollisionsvorhersage in Echtzeit aus monokularen RGB-Bildern zu generieren, muss somit ein Algorithmus entwickelt werden, der dies mit ebenso hoher Geschwindigkeit umsetzt. Convolutional Neural Networks sind dabei die effizienteste Möglichkeit, so eine komplexe Aufgabe in realistischer Zeit umzusetzen. Ein weiterer Grund ist die Tatsache, dass der Multicopter neben der Kollisionsvorhersage weitere Berechnungen wie Objekterkennung und Flugsteuerung durchführen muss und somit nur eine geringe Zeitspanne für die Kollisionsvorhersage zur Verfügung steht. Ausschlaggebend ist hierbei, dass Veränderungen am CNN, welche eine reduzierte Inference Time beabsichtigen, gleichzeitig eine Qualitätsminderung der Depth Map zur Folge haben können, weshalb der Fokus bei der Entwicklung des CNN auf der Balance dieser beiden gegensätzlichen Auswirkungen liegt.

Die Entwicklung der anschließenden Kollisionsvorhersage mithilfe der generierten Depth Maps erfolgt in unterschiedlichen Komplexitätsgraden.

4 Konzeption

Die Problemstellungen aus dem vorherigen Kapitel werden mit den Konzepten des folgenden Kapitels gelöst, wobei die Konzepte auf den Grundlagen aus Kapitel 2 aufbauen.

4.1 Depth Map Erzeugung

Für jedes Bild der Multicopter-Kamera wird eine Depth Map generiert, mithilfe derer eine Kollisionsvorhersage erzeugt werden soll. Für die Umsetzung des CNN wird die Architektur des *Dense Depth* nach [Alha19] rekonstruiert, welches zum Zeitpunkt der Erstellung dieser Arbeit das State Of The Art - Neural Network für monokulare Tiefenwahrnehmung ist. Das *Dense Depth* entspricht einem CNN mit Encoder - Decoder Architektur, dessen Code Open Source ist [Dens20]. Für das Training wird der Datensatz *NYU Depth V2* (Abschnitt 5.3) verwendet.

Unter den Spezifikationen des Implementationsrechners (Unterabschnitt 5.2.2) beträgt die Inference Time des originalen Dense Depth 50ms. Um diese Zeit zu reduzieren, werden verschiedene *Konfigurationen* getestet, welche unterschiedliche Parameter und Hyperparameter beinhalten. Parallel dazu spielt die Qualität der dadurch erzeugten Depth Map eine entscheidende Rolle (siehe Kapitel 3), weshalb die Evaluation neben der Inference Time auch die Qualität der Depth Map berücksichtigt.

4.2 Ablauf der Kollisionsvorhersage

Sobald die Depth Map vorliegt, ist die Entfernung jedes Pixels bekannt, bzw. die Entfernung des Objekts, welches das jeweilige Pixel repräsentiert. Die darauffolgende Kollisionsvorhersage erfolgt zunächst anhand *eines* Frames, anschließend mithilfe eines weiteren, konsekutiven Frames.

Erkennung anhand eines Frames

Wird eine Kollision an nur *einem* Frame vorhergesagt, ist ausschließlich die Entfernung aller Objekte bekannt, repräsentiert durch deren Pixel. Dadurch lässt sich keine Information über die Bewegung der Objekte gewinnen, was zur Folge hat, dass keine Kollision *vorhergesagt* werden kann, sie kann lediglich *erkannt* werden. Dies geschieht, indem mithilfe der Abmessungen des Multicopters und dem Sichtfeld seiner Kamera (*field of view*, FOV) berechnet wird, ob sich ein Objekt im Weg des Multicopters befindet, siehe Abbildung 4.1. Ist dies der Fall, findet eine Kollision statt. Befindet sich kein Objekt im Weg des Multicopters, findet keine Kollision statt. Unter diesen Annahmen sind die Aktivitäten der Achsen des Multicopters irrelevant, da keine Bewegung abgebildet werden kann.

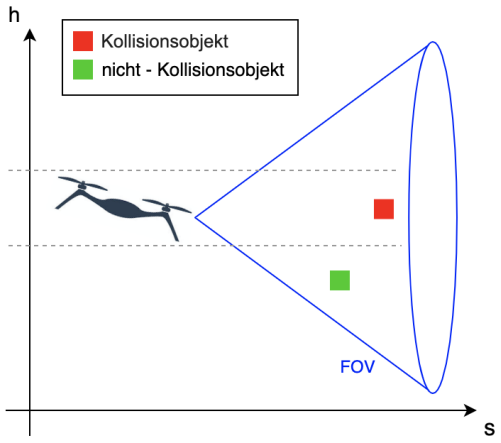

Abbildung 4.1: Kollisionserkennung anhand eines Frames

Da lediglich die Entfernung jedes Pixels bekannt ist, lässt sich die Umgebung, die durch diese Art der Kollisionserkennung abgebildet werden kann, mathematisch wie folgt definieren:

$$f(\vec{x}) = \vec{b} \tag{4.1}$$

Die dadurch bezeichneten Objekte sind folglich statisch in ihrer Höhe b.

Vorhersage anhand zweier, konsekutiver Frames

Durch die Berechnung eines Optical Flows (Abschnitt 2.8) mithilfe zweier konsekutiver Frames, kann ein zweidimensionaler Richtungsvektor für jedes Pixel erzeugt werden. Wird zusätzlich zum Optical Flow auch die Tiefeninformation durch die Depth Map berücksichtigt, lässt sich ein dreidimensionaler Richtungsvektor erstellen, mit dem lineare Bewegungen abgebildet werden können. Dieser Vektor beschreibt die Richtung und Geschwindigkeit der sichtbaren Objekte, repräsentiert durch ihre Pixel.

Unter diesen Umständen gibt es zwei mögliche Kollisions-Szenarien:

1. Der Multicopter fliegt durch schlechte Steuerung aktiv in ein festes Objekt.
2. Der Multicopter fliegt konstant geradeaus, aber ein bewegliches Objekt steuert aktiv in den Flugpfad.

Somit können lineare Bewegungen mit konstanter Geschwindigkeit des Multicopters oder der sichtbaren Objekte abgebildet werden (Abbildung 4.2).Wird die Bewegung in die Zukunft projiziert, lässt sich feststellen, ob und wann eine Kollision zwischen dem Multicopter und seiner Umgebung stattfindet. Aus der vorherigen Kollisionserkennung wird damit eine Kollisionsvorhersage.

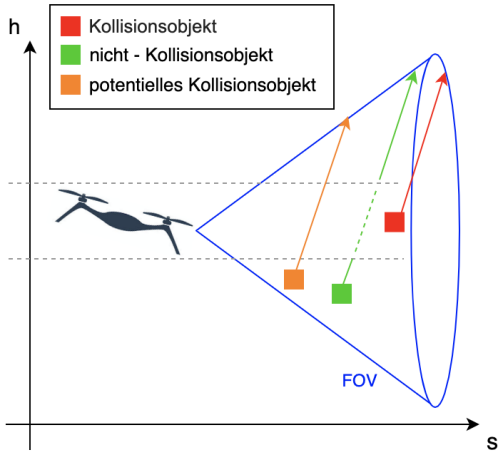

Abbildung 4.2: Kollisionsvorhersage anhand zweier konsekutiver Frames

Da unter diesen Annahmen lineare Bewegungen bekannt sind, lässt sich die Umgebung, die durch so eine Art der Kollisionsvorhersage abgebildet werden kann, mathematisch wie folgt definieren:

$$f(\vec{x}) = m\vec{x} + \vec{b} \tag{4.2}$$

Diese mathematischen Funktionen sind Vereinfachungen der Realität, bei der nicht nur statische und lineare Bewegungen eine Rolle spielen, sondern auch beschleunigte Bewegungen:

$$f(\vec{x}) = a\vec{x}^{\,2} + m\vec{x} + \vec{b}$$

Oder Bewegungen mit variierender Beschleunigung:

$$f(\vec{x}) = c\vec{x}^{\,3} + a\vec{x}^{\,2} + m\vec{x} + \vec{b}$$

Natürlich ließe sich diese Reihe noch beliebig fortführen. Zur Erfassung solcher Bewegungen wäre dementsprechend die Beobachtung weiterer Frames nötig, worauf mit Blick auf den Umfang dieser Arbeit jedoch verzichtet wird.

5 Implementation

In folgendem Kapitel soll die gesamte Umsetzung der gesamten Kollisionsvorhersage beschrieben werden. Diese beinhaltet die Beschreibung der Entwicklungsumgebung und der Hardware, der Erläuterung der Daten und deren Vor- und Aufbereitung, der Ausarbeitung des CNN, dessen Training und der Interpretation des Ergebnisses zur letztendlichen Kollisionsvorhersage.

5.1 Entwicklungs- und Testumgebung

5.1.1 Python

Für die Entwicklung des Algorithmus wird die Skriptsprache *Python* genutzt. Der allgemeine Vorteil von Python besteht in dessen geringer Komplexität, der starken Anlehnung an die menschliche Sprache und der daraus resultierenden einachen Lesbarkeit [Pyth20]. Bei der Verwendung komplexer mathematischer Konzepte ist es von großem Vorteil, den Code einfach schreiben zu können, aber auch für andere Entwickler einfach lesbar zu gestalten. Python kommt mit einer Vielzahl von *Paketen*, die eine Reihe vorgefertiger Python-Funktionen beinhalten. Allen voran ist hier das Paket *NumPy* (*Numerical Python*) zu erwähnen, welches numerische Rechenoperationen optimiert und deshalb sehr zeit- und ressourcensparend arbeitet. Dies ist ein wichtiger Grund dafür, warum Python eine sehr häufige Anwendung im Bereich des Maschinellen Lernens und damit auch im Deep Learning findet.

5.1.2 SPYDER

SPYDER (*Scientific Python Development Environment*) ist eine integrierte Entwicklungsumgebung für die Programmiersprache Python.

„SPYDER ist eine leistungsfähige wissenschaftliche [Entwicklungs-] Umgebung, die in Python geschrieben wurde, für Python, und von und für Wissenschaftler, Ingenieu-

re und Datenanalytiker entworfen wurde. Sie bietet eine einzigartige Kombination aus den erweiterten Bearbeitungs-, Analyse-, Debugging- und Profiling-Funktionen eines umfassenden Entwicklungswerkzeugs mit den Datenexplorations-, interaktiven Ausführungs-, Tiefenprüfungs- und schönen Visualisierungsmöglichkeiten eines wissenschaftlichen Pakets.“ [Spyd18]

5.1.3 MatLab

MatLab ist eine Software um Daten zu analysieren, Algorithmen zu entwickeln und mathematische Modelle zu erstellen [Math20]. Das *Dense Depth* CNN (Abschnitt 4.1) hat Daten für das Training genutzt, welche in MatLab aufbereitet wurden, siehe Abschnitt 5.3.

Oftmals liefern Open Source - Projekte, welche komplexe Berechnungen mit MatLab durchführen, eine Toolbox mit, um bestimmte Berechnungen und Ergebnisse innerhalb von MatLab reproduzieren zu können. Eine Toolbox beinhaltet den dazugehörigen Code.

5.1.4 TensorFlow und Keras

Laut [Tens20] ist *TensorFlow* eine in der Programmiersprache Python entwickelte Open Source Library um Modelle des Maschinellen Lernens zu entwickeln und zu trainieren. Sie wird u.a. von *Keras* verwendet, einer Open Source - Deep Learning Library in der Programmiersprache Python. Keras wurde dafür entwickeklt, mit möglichst geringer Verzögerung entwickeln zu können und schnell zu experimentieren [Kera20]. Es vereinfacht die Nutzung von TernsorFlow und liefert bereits vorgefertige Deep Learning Modelle. TensorFlow und Keras sind sehr weit verbreitet und viele allgemeine Probleme des Maschinellen Lernens wurden bereits mit diesen beiden Bibliotheken gelöst. Das *Dense Depth* CNN (Abschnitt 4.1) wurde mithilfe von Keras entwickelt, weshalb die Implementation des CNN dieser Arbeit ebenfalls in Keras stattfindet.

5.2 Hardware

5.2.1 Verwendeter Multicopter

Der Multicopter, der für diese Arbeit verwendet wird, ist der *BetaFPV85X 4K Whoop Quadcopter (4S)* (Abbildung 5.1).

Abbildung 5.1: Verwendeter Multicopter [Beta20]

Spezifikationen [Beta20]:

- Abmessungen: 60x60 mm
- Gewicht: 88.6 Gramm (ohne Akku)
- CPU: STM32F405
- Kamera: CADDX Tarsier (30-60 fps); FOV: 165°, Auflösung: 720x540 px

5.2.2 Implementationsrechner

Im folgenden werden die Spezifikationen des Implementationsrechners gelistet, mit dem neben der Implementation des CNN auch die Datenvorverarbeitung mithilfe von MatLab umgesetzt wurde.

- Arbeitsspeicher (RAM): 16 GB
- Grafikkarte (GPU): 1x Nvidia Titan Xp
- Betriebssystem: Ubuntu 16.04
- Typ: 64 Bit
- Prozessor: AMD Ryzen 5 2600X (1 Socket, 6 Cores/Socket, 2 Threads/Core)

5.2.3 Trainingsrechner

Nachfolgend sind die Spezifikationen des Trainingsrechners gelistet, der das Training des CNN durchführt. Dieser ist Teil eines Rechenclusters an der *Hochschule für Angewandte Wissenschaften Hamburg*.

- Arbeitsspeicher (RAM): 168 GB
- Grafikkarten (GPUs): 4x NVIDIA Quadro P6000
- Betriebssystem: CentOS Linux 7
- Typ: 64 Bit
- Prozessor: Intel Core Processor (9 Sockets, 2 Cores/Socket, 1 Thread/Core)

5.3 Datenaufbereitung

Um das CNN zu trainieren, wird der Datensatz *NYU Depth V2* [Nyud12] der Universität New York verwendet, der auch im originalen Paper [Alha19] genutzt wird. Der Datensatz besteht aus 574 Videosequenzen bzw. Szenen, die in den Räumlichkeiten der Universität New York aufgezeichnet wurden. Diese Videosequenzen beinhalten 500.670 Bilder, welche zum einen als RGB-Bilder vorliegen und zum anderen als Infrarot-generierte Depth Maps (Beispiel in Abbildung 5.2), wie sie unter Abschnitt 2.3 beschrieben wurden. Die Depth Maps werden in diesem Zustand als *raw* bezeichnet, da sie noch nicht weiter aufbereitet wurden. Unter Verwendung von MatLab und der mitgelieferten Toolbox werden die Daten aufbereitet, um eine Verwendung innerhalb des CNN zu ermöglichen.

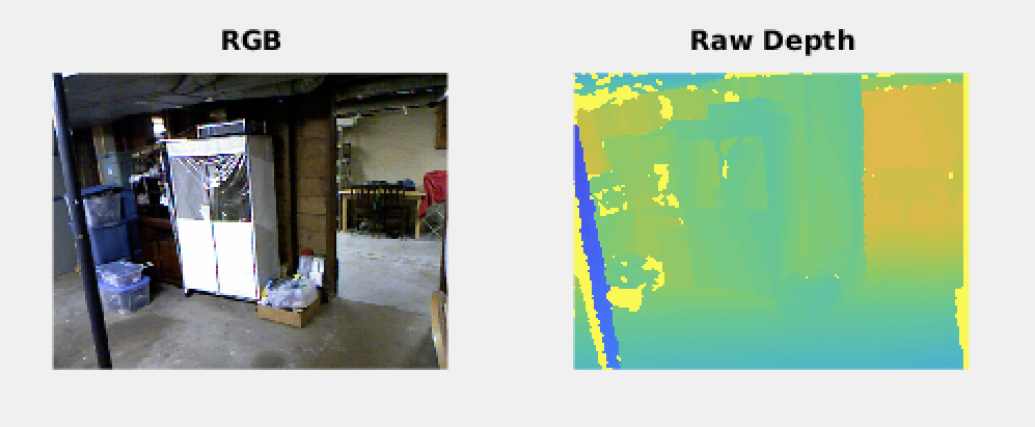

Abbildung 5.2: Beispiel eines RGB-Bildes und der dazugehörigen Raw Depth Map

Bei der Erstellung des RGB-Bildes und der dazugehörigen Depth Map, haben sich die RGB-Kamera und die Infrarot-Kamera physikalisch nebeneinander befunden. Dies hat zur Folge, dass an den Rändern der Bilder Teile der Umgebung zu sehen sind, die auf dem jeweils anderen Bild nicht vorhanden sind. Um diesen Versatz auszugleichen, werden die beiden Bilder aufeinander *projiziert* (*projected*), damit die Inhalte sich gleichen. Lediglich die Schnittmenge beider Bilder wird weiter verwendet. Zudem weist die Depth Map ein hohes Rauschen auf und es können Pixel fehlen, was der Erstellung durch den Infrarotsensor zu verschulden ist. Um die Qualität der Depth Map zu verbessern, wird das *Rauschen entfernt* (*denoising*). Beide der soeben genannten Schritte sind Funktionen der mitgelieferten MatLab Toolbox und in Abbildung 5.3 zu sehen.

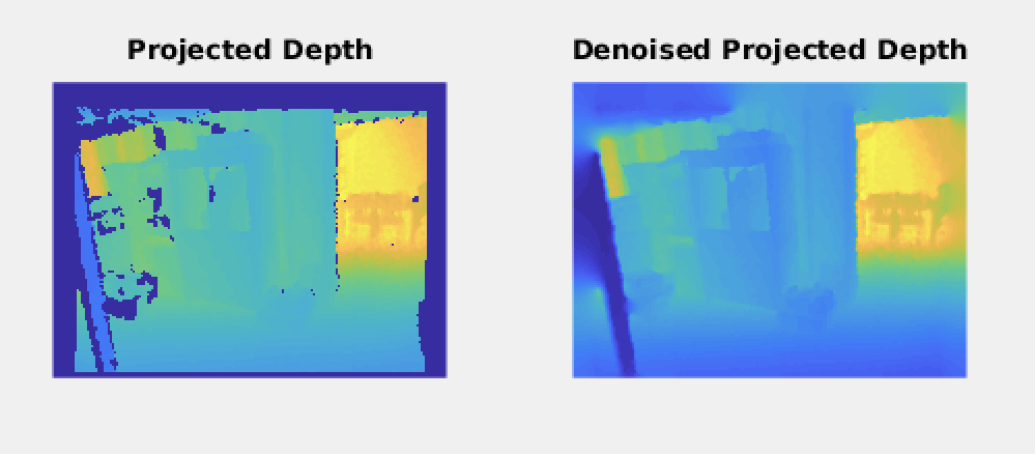

Abbildung 5.3: Beispiel Projection und Denoising

Beim Vergleich zwischen „Raw Repth“ und „Denoised Projected Depth“ ist eine erhebliche Qualitätsverbesserung zu erkennen. Die Datenaufbereitung der 500.670 Bilder mit MatLab hatte unter den Spezifikationen des Implementationsrechner (Unterabschnitt 5.2.2), bei einer Auslastung von 95%, eine Laufzeit von ca. 9 Tagen.

Beim Training des CNN enstprechen die RGB-Bilder dem Input und die Denoised Projected Depth Maps dem Label. Während der Datenaufbereitung wird die vorhandene Datenstruktur gespiegelt, worin die aufbereiteten Daten gesichert werden. Die Kombinationen aus Inputs und Labels werden in jeweils eine *.mat-Datei* (*mat-file*) gespeichert. Dieses Dateiformat beinhaltet Key-Value-Paare, was es ermöglicht, beide Bilder in *einer* Datei zu speichern. Dem Key „imgRgb“ wird als Value das RGB-Bild zugeordnet und dem Key „denoisedImgDepthProj“ wird das Label zugeordnet. Somit muss in der späteren Nutzung der Daten lediglich eine Datei pro Sample geladen werden, um mit den jeweiligen Keys auf die Bilder zugreifen zu können.

Bei der Erzeugung der 500.670 mat-files kann es aufgrund der Quantität durchaus zu fehlerhaften Datein kommen, welche anschließend Probleme bei der Entwicklung erzeugen können. Deshalb wurde ein Skript geschrieben (*matfile_verifier.py*), das jedes mat-file einmal öffnet und auf Lesbarkeit überprüft. Die fehlerhaften Dateien werden automatisch gelöscht (Fehlerrate $< 0.0001\%$).

5.4 Depth Map

5.4.1 Datenvorbereitung

Damit das CNN beim Training korrekt auf den Datensatz zugreifen kann, wurde ein Python Skript geschrieben (*csv_generator_general.py*), das die Pfade aller mat-files in eine CSV Datei (*data.csv*) schreibt. Mit dieser Datei als Datengrundlage findet im weiteren Verlauf eine Unterteilung in Trainings-, Validierungs- und Testdatensatz statt. Wie in Unterabschnitt 2.4.8 beschrieben, ist es allerdings wichtig, dass diese drei Datensätze inhaltlich gänzlich unabhängig voneinander sind. Eine einfache Unterteilung des gesamten Datensatzes in einem 8:1:1 Verhältnis mithilfe der Anzahl der erzeugten Pfade ist nicht ohne weiteres möglich, da die Daten als Frames von Videosequenzen vorliegen. Dies hat zur Folge, dass unterschiedliche Frames innerhalb *einer* Szene sich sehr ähnlich sehen und somit zwei beinahe identische Bilder in mehr

als einem Datensatz vorkommen könnten. Damit wäre eine Generalisierung des CNN nicht mehr gewährleistet und die berechnete Genauigkeit hinfällig. Ein Besipiel der ähnlichen Frames ist in Abbildung 5.4 zu sehen. Dort ist der erste und hundertste Frame der Szene „library_0005“ zu sehen.

Abbildung 5.4: Ähnliche Bilder innerhalb einer Szene

Um dieses Problem zu lösen, wird der data.csv - Datei nicht nur der Pfad jedes Frames hinzugefügt, sondern zusätzlich auch die dazugehörige Nummer der Szene, wie in Abbildung 5.5 zu sehen ist.

```
/media/wargdrones/PATRIOT_2TB/NYU_matfiles/basement_0001b/98.mat  1
/media/wargdrones/PATRIOT_2TB/NYU_matfiles/basement_0001b/99.mat  1
/media/wargdrones/PATRIOT_2TB/NYU_matfiles/basement_0001c/1.mat   2
/media/wargdrones/PATRIOT_2TB/NYU_matfiles/basement_0001c/10.mat  2
```

Abbildung 5.5: Beispiel für Pfad und Szenen-Nummer der Datei data.csv

Mithilfe dieser data.csv wird der gesamte Datensatz unterteilt. Hierfür wurde ein weiteres Python Skript geschrieben (*csv_generator_train_valid_test.py*), das drei CSV Dateien (*training_data.csv*, *validation_data.csv*, *test_data.csv*) erstellt. Das Skript wählt die ersten 80% aller Szenen (Szene 0 bis 459) für das Training, die folgenden 10% für die Validierung (Szene 460 bis 515) und die restlichen 10% für den Test (Szene 516 bis 573). Somit ist ein maximales Abstraktionsvermögen des CNN gewährleistet.

5.4.2 Dense Depth Architektur

Das Dense Depth [Alha19], welches in dieser Arbeit reimplementiert wird, hat eine Encoder - Decoder Architektur. Für den Encoder wird Transfer Learning genutzt, explizit das *DenseNet-169* [Huan18], das auf dem ImageNet Datensatz [Imag16] vortrainiert ist und sonst keine weiteren Modifikationen erhalten hat. Eine Besonderheit beim Decoder ist, dass er sogenannte *Skip Connections* zum Encoder enthält (Abbildung 5.6). Diese Connections sorgen dafür, dass das Ergebnis eines Upsampling Layers des Decoders, mit dem Convolution Layer des Encoders verbunden wird, dessen Dimension identisch ist. Dieser Verbund entspricht *einem* Layer des Decoders. Der Effekt der Skip Connection ist, dass der Mechanismus, der die Informationen im Encoder verdichtet, gegenläufig im Decoder angewandt wird, um die Informationen wiederherzustellen. Der Grund dafür ist, dass prinzipiell versucht wird, das Input-Bild mit dem CNN wiederherzustellen, allerdings nicht mit drei RGB-Channel, sonder mit einem Tiefen-Channel. Deshalb sorgt die letzte Convolution des Decoders dafür, dass der erzeugte Output lediglich einen Channel hat und somit einer Depth Map entspricht.

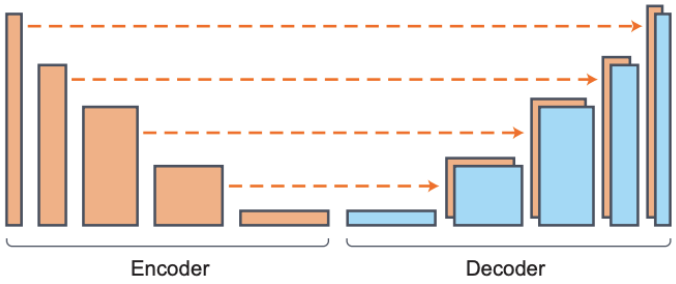

Abbildung 5.6: Dense Depth Architektur mit Skip Connections [Alha19]

5.4.3 Experiment Design

Um die Auswirkungen unterschiedlicher Konfigurationen der Parameter und Hyperparameter auf das CNN eindeutig identifizieren zu können, werden ausschließlich *einzelne* Änderungen an der *Dense Depth* - Konfiguration vorgenommen. Um eine reibungslose Analyse zu gewährleisten, wird nach dem Training nicht nur das Model gespeichert, sondern zusätzlich auch die Trainingshistorie, die Evaluation des Testdatensatzes und die Prediction von drei bestimmten Bildern aus dem Testdatensatz (siehe Unterabschnitt 5.4.4).

Folgende Konfigurationen werden untersucht:

1. **Default**
   Die reguläre Architektur des *Dense Depth* [Alha19].

2. **KITO - Keras Inference Time Optimizer**
   Dieser Algorithmus reduziert die Infernce Time, indem die vorhandenen Batch Normalization Layer aus dem Encoder entfernt werden. Diese sind nicht notwendig, da der Encoder im Sinne des Transfer Learning (Abschnitt 2.6) kein Training mehr erfährt und die Batch Normalization - Layer für die reine Prediction nicht notwendig sind [KITO20]. Dadurch kann das Ergebnis des Encoders schneller berechnet werden, was eine verringerte Inference Time zur Folge hat.

3. **Batch Normalization**
   Im Dense Depth Algorithmus wird keine Batch Normalization angewandt, was nach [Bjor18] jedoch einen negativen Einfluss auf die Performance und die Fähigkeit der Generalisierung eines CNN haben kann. Deshalb werden nach den Upsampling-Layern im Encoder (siehe Unterabschnitt 5.4.4) zusätzlich Batch Normalization - Layer implementiert.

4. **Batch Size Adaption**
   Wie in Unterabschnitt 2.4.6 beschrieben, bestimmt die Batch Size, wie oft die Gewichte während des Trainings aktualisiert werden. Bei einer konstanten Anzahl an Bildern pro Epoche werden die Gewichte bei einer geringen Batch Size häufiger aktualisiert, wohingegen das arithmetische Mittel eines Batches bei großer Batch Size aufgrund von geringerem Rauschen aussagekräftiger ist. Die Auswirkungen dieser beiden Tatsachen sollen mit einer Verdopplung und einer Halbierung der Default - Batch Size auf 16 und 4 ermittelt werden.

5. **Trainingsdaten Reduktion**
   Umso größer die zur Verfügung stehende Datenmenge ist, desto besser ist die Generaliserungsfähigkeit eines CNN und somit auch die Qualität der Berechnungen. Diese Konfiguration soll untersuchen, wie sich das Ergebnis des CNN bei reduzierter Datenmenge verhält, da die Menge der zur Verfügung stehenden Daten im Machine Learning meist stark begrenzt ist. Damit können die Untersuchungen dieser Arbeit als Grundlage für weitere Forschungen genutzt werden. Um die Auswirkungen mit den anderen Konfigurationen vergleichen zu

können, bleiben Validierungs- und Testdaten erhalten, lediglich die Menge der Trainingsdaten wird auf 70% und 40% reduziert.

5.4.4 Trainingsablauf

Die folgende Beschreibung des Trainingsablaufs anhand des Python Skripts *training.py* gilt repräsentativ für alle Konfigurationen. Für die unterschiedlichen Konfigurationen werden lediglich geringfügige Abwandlungen am Algorithmus vorgenommen, um die gewünschten Änderungen zu erreichen.

Aufgrund der Konventionen, die an der Hochschule für Angewandte Wissenschaften Hamburg für die Nutzung des Trainingsrechners (Unterabschnitt 5.2.3) gelten, muss die Umgebung des Trainingsprogramms isoliert werden. Hierfür wird eine Containervirtualisierung mit *Docker* [Dock20] genutzt. Um von der isolierten Anwendung dennoch auf die GPUs des Rechners zugreifen zu können, wurde das *Nvidia-Docker Base Image* [GitH20] verwendet.

Beim Training eines CNN mit kleinen Datensätzen werden normalerweise alle vorhandenen Daten in den RAM eines Rechners geladen, um von dort den Transfer auf die GPU zu gewährleisten. Sobald jedoch die Datenmenge den zur Verfügung stehenden RAM übersteigt, muss eine Alternative gefunden werden. Hierfür werden sogenannte *Data Generator* genutzt, die lediglich kleine Batches laden, wodurch dem Rechner zu jeder Zeit ausreichend RAM zur Verfügung steht. Im späteren Training kann durch die Variable *max_queue_size* bestimmt werden, wie viele dieser Batches der Generator präventiv vorbereiten soll. Ein Generator wird mit der Funktion *data_generator()* instanziiert. Sie wird genutzt, um im Skript zunächst drei Generator zu erstellen, für den Trainings-, Validierungs- und Testdatensatz (respektive die Variablen *train_datagen*, *val_datagen*, *test_datagen*). Die Variable *batch_size* definiert dabei die Größe des Batches. In Übereinstimmung zum Dense Depth ist die batch_size auf 8 gesetzt.

Anschließend wird das *Model* mit der Funktion *create_model()* erstellt, welche ein CNN mit der Architektur des Dense Depth (Unterabschnitt 5.4.2) instanziiert. Hierfür wird zunächst der Encoder erzeugt (im Skript *base_model* genannt), der dem vortrainierten *DenseNet-169* entspricht, das bereits nativ in Keras implementiert

ist. Als Übergabeparameter *input_shape* bekommt das DenseNet die Auflösung der Multicopter-Kamera, da diese bei der Anwendung die Bilder für das CNN erzeugt. Zudem wird durch die Layer des Encoders iteriert, um sie auf *trainable=False* zu setzen. Dadurch werden die gelernten Features im nachfolgenden Training nicht verlernt, weil die Gewichte nicht mehr aktualisiert werden können (siehe Abschnitt 2.6).

Des Weiteren wird in create_model() der Decoder erstellt, welcher als input_shape die Output Layer - Dimensionen des Encoders bekommt, um eine spätere Konkatenation zu gewährleisten. Der Aufbau des Decoders wird durch die gekapselte Funktion *upproject()* realisiert, welche einen eigenen Layer darstellt, der das Upsampling durchführt und die Skip Connection erstellt. Der Decoder besteht primär aus solchen upproject()-Layern. Die Gewichte werden gemäß dem Dense Depth in dem Intervall $[-limit, limit]$ initialisiert, wobei *limit* dem Wert $\sqrt{\frac{6}{in+out}}$ entspricht. Dabei beschreibt *in* die Anzahl der Inputs und *out* die Anzahl der Outputs eines Layers [Glor06].
Letztlich wird der Decoder mit dem Encoder konkateniert, was somit dem Rückgabewert der Funktion create_model() entspricht.

Im Anschluss wird das Model mithilfe der Keras-eigenen Funktion *multi_gpu_model()* Multi-GPU - fähig gemacht, indem der Funktion das Model und die Anzahl der zur Verfügung stehenden GPUs übergeben wird. Dadurch wird das *parallel_model* erzeugt, das im direkten Anschluss kompiliert wird. Bei der Kompilation werden zwei wichtige Variablen übergeben. Bei der ersten Variable handelt es sich um einen Optimizer (Unterabschnitt 2.4.5), der analog zum Dense Depth als *ADAM* [King15] implementiert ist. Diesem wird zusätzlich die Learning Rate mit einem Wert von 0,0001 übergeben. Die zweite Variable ist die Loss Function (Unterabschnitt 2.4.3). Hierfür wurde beim Dense Depth eine eigene Funktion entwickelt, welche den Namen *Depth Loss Function* trägt und an das vorliegende Problem der Tiefenwahrnehmung angepasst ist [Alha19]. Diese Funktion wird hier ebenfalls verwendet.

Für das Training werden drei unterschiedliche Callbacks definiert, die nativ in Keras implementiert sind:

- *ModelCheckpoint()*
  Hier wird ein Dateipfad angegeben, unter dem die Gewichte des Models nach

jeder Epoche gespeichert werden, unter der Voraussetzung, dass sich der Loss in der Validierung verbessert hat. Sollte sich die Qualität des CNN im weiteren Trainingsverlauf aufgrund von z.b. Overfitting wieder verschlechtern, kann der letzte Checkpoint im Anschluss an das Training geladen werden. Somit steht nach jedem Training stets die beste Variante des aktuellen Models zur Verfügung.

- *EarlyStopping()*
  Sollte ein Overfitting am Trainingsdatensatz geschehen, ist die beste Variante des aktuellen Models bereits durch den ModelCheckpoint gespeichert und somit ein weiteres Training hinfällig. Diese Tatsache kann durch das EarlyStopping detektiert werden und das Training somit frühzeitig abgebrochen werden. Dafür wird dem Callback eine Variable *patience* übergeben, welche angibt, nach wie vielen Epochen das Training nach dem letzten, minimalen Loss der Validierung abgebrochen werden soll. Somit kann ein unnötig langes Training verhindert werden. Die patience wurde so gewählt, dass das CNN einmal den gesamten Trainingsdatensatz ohne Verbesserung des Validierungs-Losses durchlaufen muss, bevor das Training abgebrochen wird.

- *ReduceLROnPlateau()*
  Falls der Optimizer den Loss nicht weiter verringern kann, besteht die Möglichkeit, dass die Learning Rate zu groß ist und sich somit dem lokalen Minimum nicht weiter genähert werden kann. Damit befindet sich der Optimizer auf einem Plateau. Um diesem Problem vorzubeugen, wird die Learning Rate verringert, sobald der Optimizer den Loss nicht weiter verbessern kann. Hierfür wird ebenfalls eine *patience* übergeben, die bestimmt, nach wie vielen Epochen die Learning Rate angepasst werden soll, sofern der Optimizer keine Verbesserung erzielen kann. Für die Anpassung wird ein *factor* übergeben, mit dem die aktuelle Learning Rate multipliziert wird. Die beiden Variablen werden wie folgt gesetzt, patience=5 und factor=0.6.

Um das Training zu starten, wird die Keras-eigene Funktion *fit_generator()* genutzt. Diese wird über das kompilierte parallel_model ausgeführt und bekommt als Übergabeparameter die beiden Data Generator, die drei Callbacks und die max_queue_size. Des Weiteren müssen die beiden Parameter *steps_per_epoch* und *validation_steps*

übergeben werden, die der Funktion fit_generator() mitteilen, wie viele Batches pro Epoche des Trainings und der Validierung verarbeitet werden sollen. Die Anzahl der Steps ist abhängig davon, wie viele Bilder das CNN pro Epoche sehen soll, was über die Variable *images_per_epoch* bestimmt wird. Wird die Variable images_per_epoch durch die batch_size dividiert, ist das Ergebnis steps_per_epoch und validation_steps.

Das Ergebnis der Trainingsfunktion wird in der Variablen *history* gespeichert. Durch sie kann im Anschluss an das Training auf den Loss des Trainings (*loss*) und der Validierung (*val_loss*) zugegriffen werden.

Nach dem Training werden die Gewichte des letzten Checkpoints geladen, um das Model als Ganzes abzuspeichern und den Testdatensatz zu evaluieren. Bei letzterem wird dafür der Data Generator *test_datagen* erzeugt, der durch den gesamten Testdatensatz iteriert, um den dazugehörigen Loss (*test_loss*) zu bestimmen. Zudem werden die drei Testbilder des Testdatensatzes, die in Unterabschnitt 5.4.3 erwähnt wurden, predicted und unter den Variablen *test_image1*, *test_image2* und *test_image3* gespeichert. Parallel dazu wird die Inference Time des CNN berechnet (*inference_time*).

Um eine spätere Anlyse der Ergebnisse zu gewährleisten, wird für jede Konfiguration ein mat-file mit dem Namen *log.mat* gespeichert, in dem die Werte folgender Variablen gespeichert werden:

- inference_time
- loss, val_loss und test_loss
- test_image1, test_image2 und test_image3

5.4.5 Ergebnisse

In diesem Unterkapitel werden die Ergebnisse der Konfigurationen aus dem vorherigen Unterabschnitt präsentiert, ohne sie dabei zu bewerten. Eine Evaluation dieser Ergebnisse findet in Kapitel 6 statt.

Wie in Unterabschnitt 5.4.4 beschrieben, werden pro Konfiguration drei Predictions der jeweils erarbeiteten Konfiguration gespeichert. Die Prediction und das korrespondierende RGB-Bild des *test_image3* wird für einen späteren Vergleich in der folgenden Abbildung 5.7 dargestellt, da sich dieses Bild aufgrund von klaren Strukturen und eindeutiger Tiefenstaffelung am besten für eine spätere Evaluation eignet. Dabei ist die Depth Map mithilfe des *originalen Dense Depth* berechnet worden, um als optischer Indikator für die nachfolgenden Ergebnisse zu dienen.

Abbildung 5.7: RGB-Bild und Original Dense Depth Prediction

Im Folgenden wird für jede Konfiguration die Entwicklung des Trainings- und Validierungs-Losses dargestellt. Zudem wird die Prediction für das RGB-Bild des *test_image3* von Abbildung 5.7 abgebildet.

1. **Default**

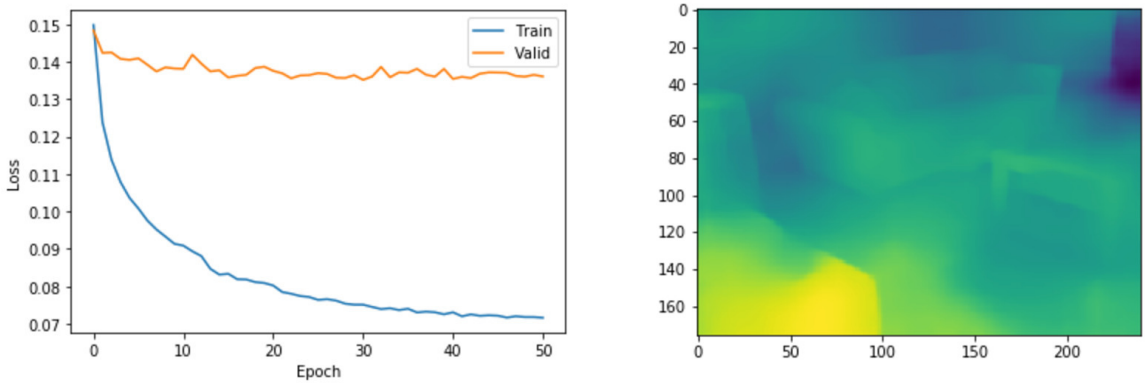

Abbildung 5.8: Ergebnis Default

Inference Time:	30.9ms	Validation Loss:	0.135
Training Loss:	0.072	Test Loss:	0.151

2. **KITO - Keras Inference Time Optimizer**

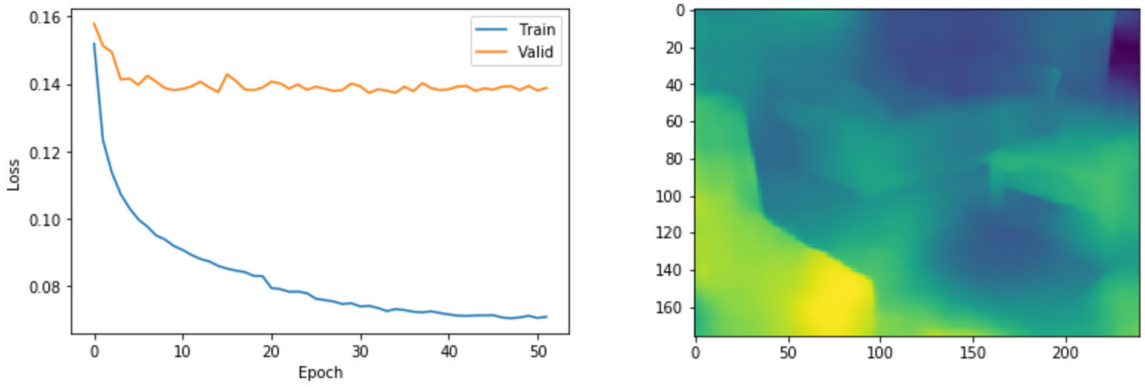

Abbildung 5.9: Ergebnis KITO

Inference Time:	27.7ms	Validation Loss:	0.139
Training Loss:	0.078	Test Loss:	0.159

3. **Batch Normalization**

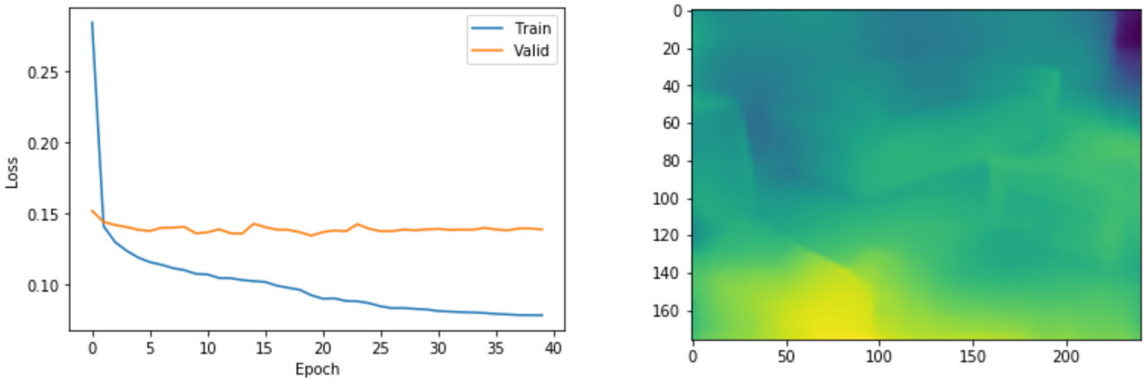

Abbildung 5.10: Ergebnis Batch Normalization

Inference Time:	30.0ms	Validation Loss:	0.135
Training Loss:	0.079	Test Loss:	0.154

4. **Batch Size Adaption**

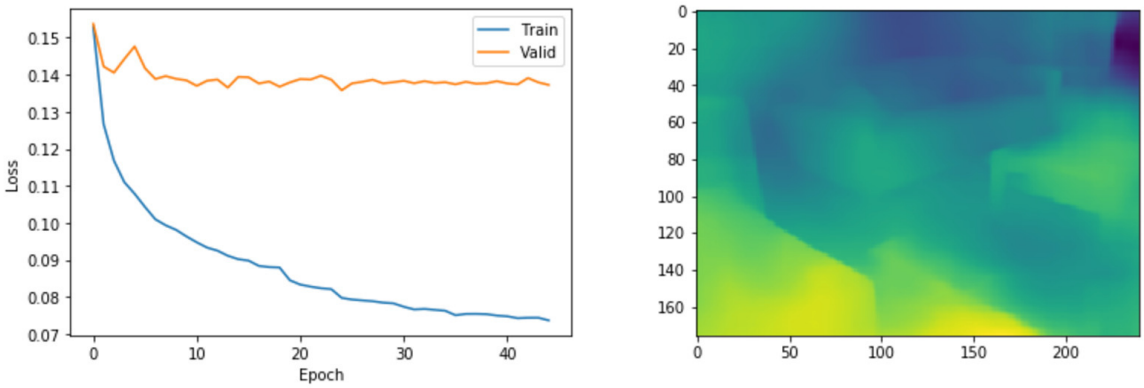

Abbildung 5.11: Ergebnis Batch Size **4**

Inference Time:	31.4ms	Validation Loss:	0.136
Training Loss:	0.074	Test Loss:	0.152

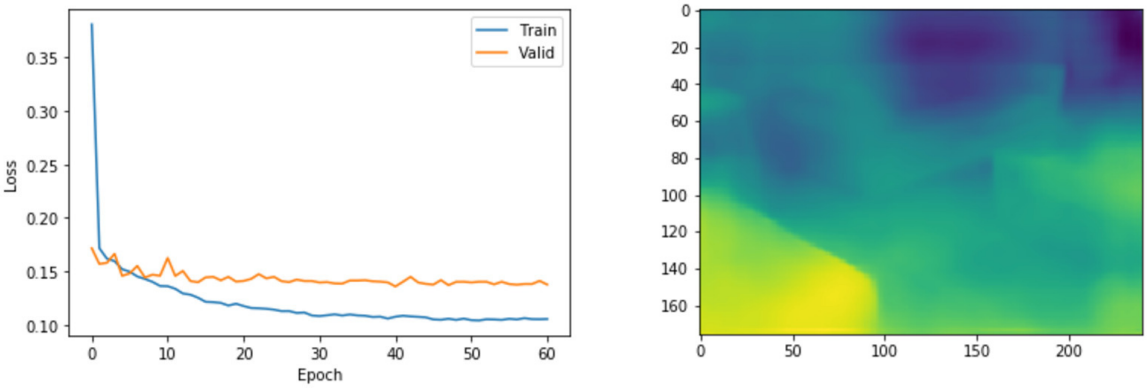

Abbildung 5.12: Ergebnis Batch Size **16**

Inference Time:	32.5ms	Validation Loss:	0.136
Training Loss:	0.104	Test Loss:	0.155

5. **Trainingsdaten Reduktion**

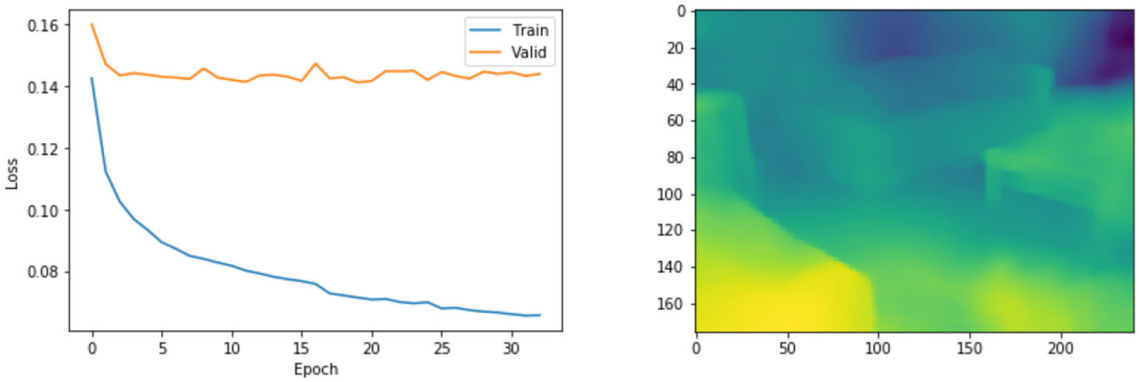

Abbildung 5.13: Ergebnis Trainingsdaten Reduktion auf **70%**

Inference Time:	29.1ms	Validation Loss:	0.141
Training Loss:	0.066	Test Loss:	0.162

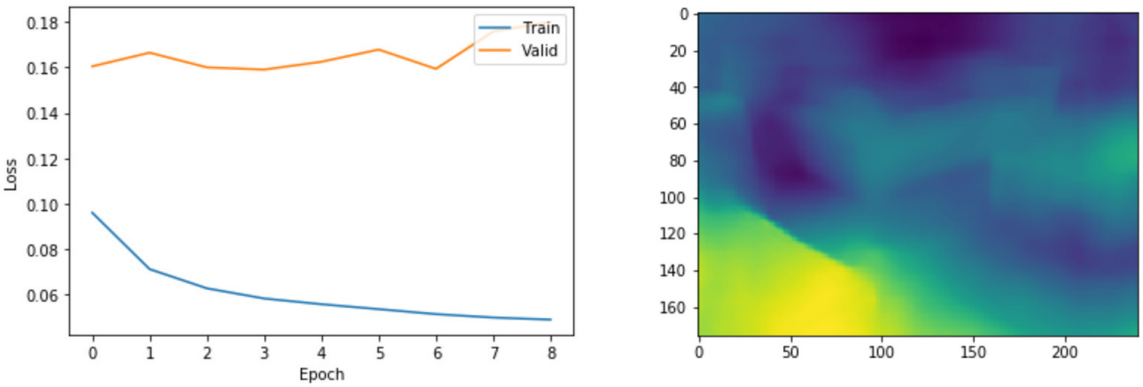

Abbildung 5.14: Ergebnis Trainingsdaten Reduktion auf **40%**

Inference Time:	30.6ms	Validation Loss:	0.159
Training Loss:	0.049	Test Loss:	0.18

5.5 Kollisionsvorhersage

In folgendem Abschnitt wird der komplizierte Sachverhalt der Kollisionsvorhersage in mehreren Schritte mit aufsteigender Komplexität zerteilt. Jeder Schritt nähert sich genauer der Realität an.

Erkennung anhand eines Frames

Sobald die Depth Map vorhanden ist, kann mit den Abmessungen des Multicopters, dem FOV der Kamera und deren Auflösung (Unterabschnitt 5.2.1) die Kollisionserkennung mithilfe eines Frames berechnet werden. Wie unter Abschnitt 4.2 beschrieben, muss bestimmt werden, ob sich ein Kollisionsobjekt vor dem Multicopter befindet. Abbildung 5.15 zeigt ein allgemeines Szenario in der Draufsicht.

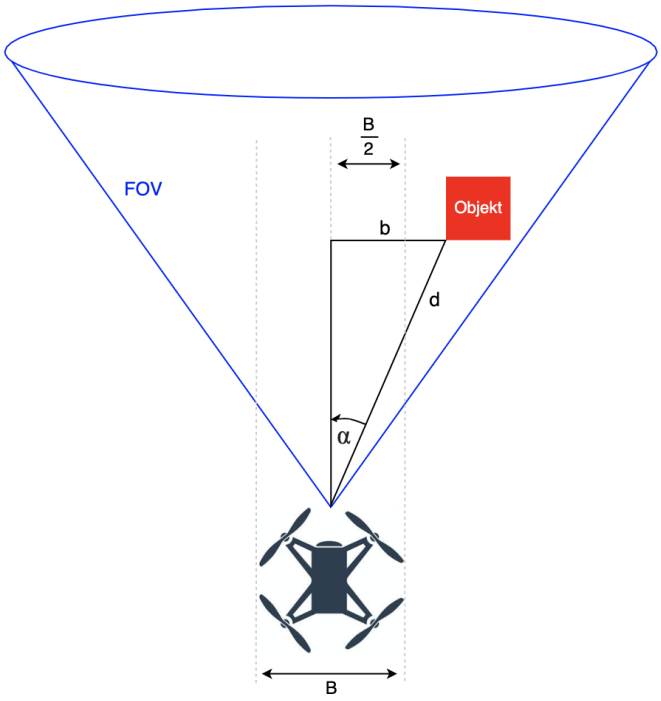

Abbildung 5.15: Szenario einer Kollisionserkennung

Zur besseren Darstellbarkeit wird hier eine 2D Ansicht auf horizontaler Ebene gezeigt, jedoch gelten die nachfolgenden Berechnungen in der Realität analog für die

vertikale Raumrichtung.

Die horizontale Entfernung b zwischen der Bildmitte der Multicopter-Kamera und einem Objekts soll größer sein als die halbe Breite B des Multicopters, damit keine Kollision stattfindet.

$$b \overset{!}{>} \frac{B}{2} \tag{5.1}$$

Wobei b über den Sinus berechnet werden kann, da die Distand d durch die Depth Map bekannt ist:

$$sin(\alpha) = \frac{b}{d}$$

$$\Leftrightarrow \quad b = d \cdot sin(\alpha) \tag{5.2}$$

Wird b aus Gleichung 5.1 durch Gleichung 5.2 substituiert ergibt sich:

$$\Rightarrow \quad d \cdot sin(\alpha) \overset{!}{>} \frac{B}{2}$$

$$\Leftrightarrow \quad d \overset{!}{>} \frac{B}{2 \cdot sin(\alpha)} \tag{5.3}$$

Das bedeutet, die Distanz d muss für jedes Pixel mindestens so groß sein wie der rechte Term der Gleichung 5.3, welcher allerdings vom Winkel α abhängt. Dieser kann über das FOV und die Auflösung der Kamera bestimmt werden, siehe Abbildung 5.16.

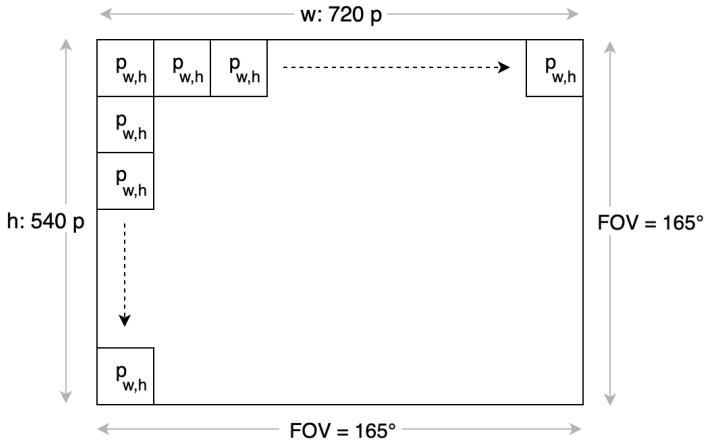

Abbildung 5.16: Zusammenhang zwischen FOV und Auflösung der Kamera

Um α für die Breite w und die Höhe h zu berechnen, wird für w und h jeweils eine Konstante k bestimmt, die beschreibt, um wieviel Grad sich α pro Pixel aus der Mitte des Bildes bewegt.

$$k_w = \frac{FOV}{w}$$

$$k_h = \frac{FOV}{h}$$

Daraus ergibt sich die Änderung des horizontalen Winkels α_w für jedes horizontale Pixel p_w, und die Änderung des vertikalen Winkels α_h für jedes vertikale Pixel p_h:

$$\alpha_{\mathrm{w,h}} = k_{\mathrm{w,h}} \cdot p_{\mathrm{w,h}} \tag{5.4}$$

Durch die Substitution von α in Gleichung 5.3 durch Gleichung 5.4, ergibt sich:

$$\Rightarrow \quad d \overset{!}{>} \frac{B}{2 \cdot sin(k \cdot p)}$$

Somit kann jeweils für die Breite und Höhe eines Frames der Mindestabstand eines Pixels als Funktion dargestellt:

$$d_{\min}(p_{\mathrm{w,h}}) = \frac{B}{2 \cdot sin(k_{\mathrm{w,h}} \cdot p_{\mathrm{w,h}})} \tag{5.5}$$

Diese Art der Kollisionserkennung wird in dem Skript *collision_detection.py* berechnet. Der dazugehörige Algorithmus berechnet eine Maske, die für jedes Pixel die minimale Distanz trägt. Diese Maske wird mit der erzeugten Depth Map abgeglichen. Sobald ein oder mehrere Pixel ihren zugehörigen Mindestabstand unterschreiten, findet eine Kollision zwischen dem Multicopter und seiner Umgebung statt. Die Maske der Kollisionserkennung ist auf Abbildung 5.17 zu sehen. Dort ist ein Tunnel zu erkennen, in dem sich kein Objekt befinden darf, da sonst eine Kollision stattfindet.

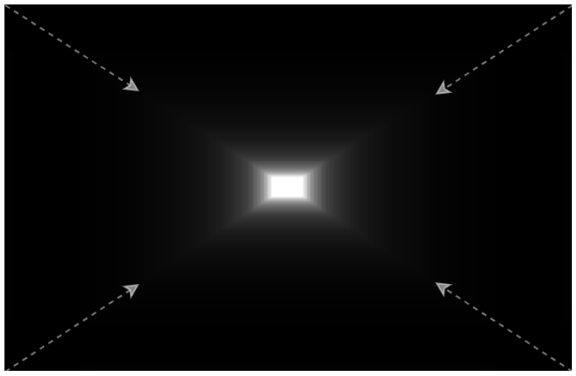

Abbildung 5.17: Kollisionstunnel aus Sicht der Multicopter-Kamera

Generell gilt es zu erwähnen, dass typische Multicopter-Objektive eine linsenbedingte Verzerrung enthalten, welche in dieser Arbeit nicht berücksichtigt wird. Um dies auszugleichen, muss für jede Kamera eine individuelle Konstante bestimmt werden, mit der jeder Frame vor der Erstellung der Depth Map entzerrt wird.

Vorhersage anhand zweier, konsekutiver Frames

Sobald ein zweiter Frame für die Berechnung berücksichtigt wird und dessen Depth Map vorliegt, kann die Richtung und Geschwindigkeit eines Objekts bestimmt werden. Zur Vereinfachung wird in der folgenden Beschreibung ein Objekt durch ein einziges Pixel dargestellt. Diese Beschreibung gilt repräsentativ für alle Pixel.

Auf Abbildung 5.18 ist die Bewegung eines Pixels vom Zeitpunkt t_1 zum Zeitpunkt t_2 dargestellt.

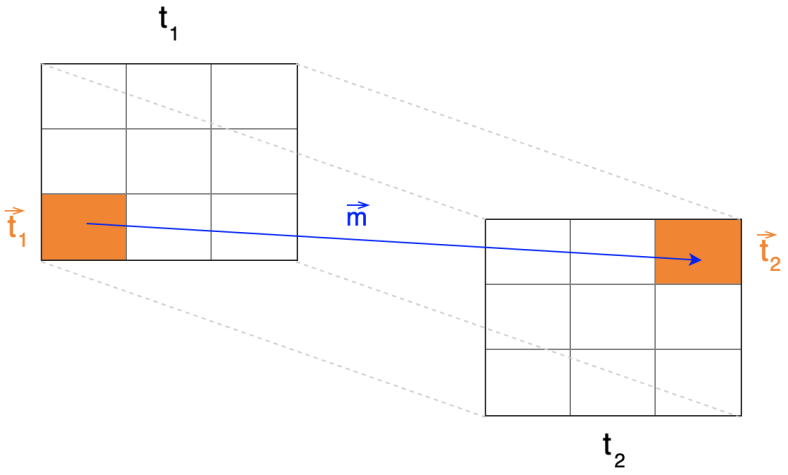

Abbildung 5.18: Richtungsvektor eines Pixels

Die räumliche Position des Pixels zum Zeitpunkt t_1 wird als Ortsvektor $\vec{t_1}$ bezeichnet, analog der Ortsvektor $\vec{t_2}$ für den Zeitpunkt t_2. Dabei beinhalten diese Vektoren die Koordinaten in allen 3 Raumrichtungen. Somit kann der Richtungsvektor $\vec{m}$ des Pixels wie folgt dargestellt werden:

$$\vec{m} = \vec{t_2} - \vec{t_1} \tag{5.6}$$

Die Koordinaten von $\vec{t_1}$ entsprechen der Stelle x für die horizontale Raumrichtung und der Stelle y für die vertikale Raumrichtung auf dem RGB-Bild. Die Tiefeninformation des Pixels ist an den selbigen Stellen der dazugehörigen Depth Map, welche als D_{t_1} bezeichnet wird.
Dadurch lässt sich $\vec{t_1}$ folgendermaßen beschreiben:

$$\Rightarrow \quad \vec{t_1} = \begin{pmatrix} x \\ y \\ D_{t_1}(x,y) \end{pmatrix} \tag{5.7}$$

Um $\vec{t_2}$ zu bestimmen, muss der Optical Flow (Abschnitt 2.8) des Pixels berechnet werden, welcher die Änderung in horizontaler und vertikaler Raumrichtung angibt. Der Algorithmus des Optical Flow (Abschnitt 4.2) liefert mithilfe der beiden RGB-Bilder das Vektorfeld F, welches die relative Änderung der Pixel in horizontaler Raumrichtung (F_x) und vertikaler Raumrichtung (F_y) angibt, jeweils ausgehend von den ursprünglichen Stellen x und y.

Somit ist die horizontale und vertikale Koordinate des Pixels zum Zeitpunkt t_2 bekannt. Mit diesen Koordinaten und der dazugehörigen Depth Map D_{t_2}, kann die Tiefe des Pixels zum Zeitpunkt t_2 beschrieben werden. Dadurch lässt sich $\vec{t_2}$ folgendermaßen darstellen:

$$\Rightarrow \quad \vec{t_2} = \begin{pmatrix} x + F_x[\,x, y\,] \\ y + F_y[\,x, y\,] \\ D_{t_2}[\,x + F_x[x, y],\ y + F_y[x, y]\,] \end{pmatrix} \tag{5.8}$$

Werden $\vec{t_1}$ und $\vec{t_2}$ in Gleichung 5.6 durch Gleichung 5.7 und Gleichung 5.8 substituiert, ergibt sich für den Richtungsvektor $\vec{m}$:

$$\Rightarrow \quad \vec{m} = \begin{pmatrix} x + F_x[\,x, y\,] \\ y + F_y[\,x, y\,] \\ D_{t_2}[\,x + F_x[x, y],\ y + F_y[x, y]\,] \end{pmatrix} - \begin{pmatrix} x \\ y \\ D_{t_1}[x, y] \end{pmatrix}$$

$$\Leftrightarrow \quad \vec{m} = \begin{pmatrix} F_x[\,x, y\,] \\ F_y[\,x, y\,] \\ D_{t_2}[\,x + F_x[x, y],\ y + F_y[x, y]\,] \ -\ D_{t_1}[x, y] \end{pmatrix} \tag{5.9}$$

Dabei beschreibt der Betrag dieses Vektors $|\vec{m}|$ die zurückgelegte Strecke des Pixels. Wenn zusätzlich die Zeitdifferenz Δt zwischen den beiden Frames bestimmt wird, kann somit die Geschwindigkeit v des Pixels berechnet werden:

$$v = \frac{|\vec{m}|}{\Delta t} \tag{5.10}$$

Mithilfe der FPS-Angabe (Frames Per Second) der Multicopter-Kamera (Unterabschnitt 5.2.1) kann Δt bestimmt werden, indem der Kehrwert $\frac{1}{FPS}$ gebildet wird. Dieser beschreibt die gesuchte Zeitdifferenz, womit sich für Gleichung 5.10 ergibt:

$$\Rightarrow \quad v = \frac{|\vec{m}|}{\frac{1}{FPS}}$$

$$\Leftrightarrow \quad v = |\vec{m}| \cdot FPS \tag{5.11}$$

Somit ist die Richtung (Gleichung 5.9) und die Geschwindigkeit (Gleichung 5.11) aller sichtbaren Objekte bekannt.Wird diese Bewegung in die Zukunft projiziert, kann mithilfe des Kollisionstunnels der Multicopter-Kamera (Abbildung 5.17) vorhergesagt werden, ob und wann eine Kollision stattfindet.

Die Überlegungen zur Kollisionsvorhersage mit zwei konsekutiven Frames bleiben aufgrund des Umfanges dieser Arbeit rein theoretischer Natur.

6 Evaluation

In diesem Kapitel werden sowohl die Ergebnisse des Trainings (Unterabschnitt 5.4.5), als auch die Implementation der Kollisionsvorhersage (Abschnitt 5.5) interpretiert.

Interpretation der Trainingsergebnisse

1. **Default**
   In Abbildung 5.8 ist zu sehen, dass der Trainingsloss monoton abnimmt, während der Loss der Validation nach ca. 17 Epochen stagniert. Optimalerweise wäre ein Beginn des Overfittings zu sehen, da somit eine optimale Anpassung zwischen Trainings- und Validierungsdatensatz stattfinden würde. Dies ist vermutlich aufgrund der Parameteranzahl des CNN nicht eindeutig zu identifizieren. Wäre diese größer, sollte der Effekt deutlicher sein. Dennoch ist die dazugehörige Depth Map von ausreichender Qualität, obwohl sie nicht die Schärfe und Qualität der Depth Map des Dense Depth (Abbildung 5.7) erreicht. Ein möglicher Grund für die qualitativ hochwertigere Prediction im originalen Dense Depth könnten andere Hyperparameter-Einstellungen sein, welche nicht in der Publikation festgehalten wurden. Zudem ist beim originalen Algorithmus Data Augmentation angewandt worden, was zu einer Vervielfachung der vorhanden Daten führt. Diese Augmentation hat in Form von horizontaler Spiegelung und Channel-Permutation der RGB-Bilder stattgefunden. Des Weiteren ist die Qualität der Labels durch [Levi04] verbessert worden.

   Dessen ungeachtet sind die Kanten der hier erarbeiteten Version klar zu erkennen und auch für das menschliche Auge ist eine Tiefenwahrnehmung eindeutig möglich. Auffällig ist hier jedoch die Inference Time von 30.9ms, die aufgrund identischer Konfiguration zum Dense Depth CNN theoretisch zur gleichen Inference Time von 50 ms führen sollte. Die Ursache dieser Tatsache ist unklar, weswegen Kontakt zu den Autoren der Publikation [Alha19] aufgenommen wur-

de. Der Grund für die kürzere Inference Time ist jedoch auch den Autoren der Studie nicht einleuchtend und benötigt daher weitere Untersuchungen.

2. **KITO - Keras Inference Time Optimizer**
   Der Effekt des stagnierenden Validation Losses ist auch in Abbildung 5.9 zu erkennen, allerdings sind die drei Loss-Werte minimal schlechter als in der Default-Konfiguration. Letzteres spiegelt sich auch in der Depth Map wieder. Die beiden dunklen Flecken auf der Couch und neben dem Tisch auf Abbildung 5.9 suggerieren eine größere Tiefe als tatsächlich vorhanden. Für eine Kollisionsvorhersage ist dieser Umstand fatal und somit nicht von ausreichender Qualität. Positiv ist hingegen die verringerte Inference Time von 27.7ms, womit das Ziel dieser Konfiguration erreicht wurde. Unklar bleibt, warum die Entfernung der Batch Normalization - Layer des Encoder-Netzwerks überhaupt zu einer verschlechterten Performance führt, obwohl diese aus theoretischer Sicht von [KITO20] ausgeschlossen wurde.

3. **Batch Normalization**
   Die Inference Time ist mit 30.0ms sehr ähnlich zur Default-Konfiguration von 30.9ms. Daraus lässt sich schlussfolgern, dass die Batch Normalization nur marginalen Einfluss auf die Berechnungszeit des CNN nimmt. Auffällig hingegen ist die erzeugte Depth Map, welche durch ein erhöhtes Rauschen wesentlich unschärfer ist als die Depth Maps der bisherigen Konfigurationen. Dadurch sind Kanten und Konturen schwerer zu erkennen, was eine Unterscheidung von Objekten ebenfalls erschwert. Somit ist diese Konfiguration nicht von ausreichender Qualität für eine Kollisionsvorhersage. Über eine Verwendung von Batch Normalization für spätere Anwendungen der Tiefenvorhersage muss also noch gründlich geforscht werden, da sie qualitativ wie auch quantitativ zu schlechteren Ergebnissen geführt haben, trotz ihrer allgemeinen Beliebtheit und nachgewiesenen positiven Effekten.

4. **Batch Size Adaption**
   Die Inference Times der beiden Konfigurationen mit Batch Sizes der Größe 4 und 16 unterscheiden sich kaum, was aufgrund der unveränderten Architekur des CNN zu erwarten war. Jedoch ist eine verminderte Qualität beider Depth Maps im Vergleich zur Default-Konfigurationen zu erkennen. Besonders die Verdopplung der Batch Size auf 16 führt zu ausgeprägten Fehleinschätzungen der

Tiefe, welche, ähnlich zur Batch Normalization - Kofiguration, fatal für eine Kollisionsvorhersage sind. Deshalb ist zumindest eine weitere Erhöhung der Batch Size nicht für eine reale Anwendung angeraten.

5. **Trainingsdaten Reduktion**
   Auch in dieser Konfiguration unterscheiden sich die Inference Times nicht merklich von der Default-Konfiguration, was aufgrund der unveränderten Architektur ebenfalls zu erwarten war. Beim Vergleich der Depth Maph, die mit 100% der Trainingsdaten trainiert wurde, mit der Depth Map der Reduktion auf 70% der Trainingsdaten, ist ein marginaler Unterschied festzustellen, weshalb eine Kollisionsvorhersage prinzipiell möglich ist. Eine Reduktion auf 40% der Trainingsdaten verschlechtert allerdings die Prediction des CNN maßgeblich. Hohe Unschärfe durch starkes Rauschen würde die Abgrenzung unterschiedlicher Objekte erheblich erschweren, weshalb diese Konfiguration für eine Kollisionsvorhersage als ungeeignet einzuschätzen ist.

Interpretation der Kollisionsvorhersage

Die Umsetzung der Kollisionserkennung an einem Frame ist als gelungen zu betrachten. Durch die Abbildung eines Kollisionstunnels (Abbildung 5.17) können Kollisionen anhand der Depth Map *eines* Frames detektiert werden. Durch die Implementation der theoretischen Ausarbeitung zur Kollisionserkennung linearer Bewegungen von Objekten, würden sich Kollisionen nicht nur erkennen, sondern auch vorhersagen lassen. Dadurch wäre Zeit geschaffen, die es dem Multicopter erlaubt, Entscheidungen zu treffen, die eine Kollision im Optimalfall verhindern würden. Aufgrund der Annahme konstanter Geschwindigkeiten, sind die Freiheitsgrade des Multicopters jedoch bisher eingeschränkt. Diese Tatsache beeinflusst auch den Reaktionsspielraum des Multicopters negativ. Dennoch ist das Ziel im Kontext dieser Arbeit erreicht.

7 Fazit und Ausblick

In Kapitel 2 wurden die Grundlagen detailliert ausgearbeitet, um ein Verständnis dieser Arbeit und der anschließenden Erörterung der Problemstellung in Kapitel 3 zu gewährleisten. Bei Letzterem wurde das Problem klar in Depth Map Prediction und Kollisionsvorhersage unterteilt, was als Fundament für die Konzeption in Kapitel 4 diente. Diese Struktur konnte ebenfalls bei der Implementation übernommen werden, um für inhaltliche Klarheit zu sorgen und der Argumentation logisch folgen zu können. Der Fokus der Implementation lag dabei auf der Entwicklung/Adaption des CNN. In der anschließenden Evaluation wurden die Ergebnisse interpretiert und diskutiert.

Das Ziel dieser Arbeit war die Entwicklung einer qualitativen Kollisionsvermeidung mithilfe eines Deep Learning Alogrithmus, bei gleichzeitiger Gewährleistung der Echtzeitfähigkeit. Die dargestellten Untersuchungen haben einen ersten Einstieg gezeigt, mithilfe dessen eine rudimentäre Kollisionsvorhersage entwickelt wurde. Durch die Untersuchungen konnte ein besserer Einblick in die Eigenschaften der Tiefenschätzung aus monokularen Kamerabildern gewonnen werden. Diese Einblicke und Techniken werden in Zukunft direkten Einfluss auf die technische Entwicklung in der *WARGdrones GmbH* haben. Damit ist das Gesamtergebnis dieser Arbeit als Grundlage für eine Kollisionsvorhersage in realistischerem Umfang zu betrachten.
Dabei könnte zum einen die Qualität der Depth Map weiter verbessert werden. Dies könnte durch Data Augmentation geschehen, wie in Kapitel 6 beschrieben, bei welcher der vorhandene Datensatz durch Bildverarbeitung vervielfacht wird, was zu einer besseren Generalisierung des CNN führen würde. Auch weitere Konfigurationen für das Training, vor allem bei den Hyperparametern, könnten die Qualität der Depth Map weiter erhöhen. Eine mögliche Lösung für die Stagnation des Validation Losses während des Trainings wäre, die minimalste Learning Rate im ReduceLROnPlateau-Callback noch tiefer zu setzten oder alternativ einen anderen Optimizer zu testen,

weil sich dadurch dem lokalen Minimum besser angenähert werden könnte.
Zum anderen könnte der Komplexitätsgrad der Kollisionsvorhersage weiter erhöht werden. Durch Hinzunahme eines dritten Frames könnten komplexere Bewegungen abgebildet werden (siehe Abschnitt 4.2). Diese würden die Änderung der Geschwindigkeit, also eine konstante Beschleunigung, berücksichtigen. Ein weiterer Frame würde eine variierende Beschleunigung berücksichtigen. Eine Möglichkeit, diese Abbildungen mathematisch zu beschreiben und daraus die Trajektorie der Objekte zu nähern, wäre durch Polynominterpolation [Kuma08].

Gemessen an der Komplexität des Problems und der aufwändigen Lösung, die in der vorliegenden Arbeit dargestellt ist, lässt sich erahnen, über wie viel Rechenleistung das menschliche Gehirn verfügen muss, wenn es die Fähigkeit der Kollisionsvorhersage, welche genutzt wird um laufen zu lernen, spätestens im Alter von 2-3 Jahren gemeistert hat. Für Kinder in diesem Alter ist autonome Fortbewegung gewiss auch eines der großen Themen ihrer Zeit.

Abbildungsverzeichnis

2.1 Freiheitsgrade eines Multicopters 7
2.2 Digitale RGB-Bilder 8
2.3 Depth Map Beispiel 9
2.4 Künstliches Neuronales Netz 10
2.5 Output eines Neurons 11
2.6 Berechnungszyklus eines KNN 12
2.7 Aktualisierung der Gewichte 13
2.8 Überspringen des lokalen Minimums 13
2.9 Schrittweise Annäherung an lokales Minimum 14
2.10 Kernel Bewegung 18
2.11 Convolution eines MxNx3 Bildes mit einem 3x3x3 Kernel 19
2.12 Pooling eines 4x4 Bildes 20
2.13 Enoder - Decoder Architektur für Convolutional Neural Networks . . 22
2.14 Optical Flow Beispiel 22
2.15 Optical Flow Beispiel - Vektorfeld 23

4.1 Kollisionserkennung anhand eines Frames 26
4.2 Kollisionsvorhersage anhand zweier konsekutiver Frames 27

5.1 Verwendeter Multicopter 31
5.2 Beispiel eines RGB-Bildes und der dazugehörigen Raw Depth Map . . 33
5.3 Beispiel Projection und Denoising 33
5.4 Ähnliche Bilder innerhalb einer Szene 35
5.5 Beispiel für Pfad und Szenen-Nummer der Datei data.csv 35
5.6 Dense Depth Architektur mit Skip Connections 36
5.7 RGB-Bild und Original Dense Depth Prediction 42
5.8 Ergebnis Default 42
5.9 Ergebnis KITO 43

5.10 Ergebnis Batch Normalization 43
5.11 Ergebnis Batch Size 4 44
5.12 Ergebnis Batch Size 16 44
5.13 Ergebnis Trainingsdaten Reduktion auf 70% 45
5.14 Ergebnis Trainingsdaten Reduktion auf 40% 45
5.15 Szenario einer Kollisionserkennung 46
5.16 Zusammenhang zwischen FOV und Auflösung der Kamera 48
5.17 Kollisionstunnel aus Sicht der Multicopter-Kamera 49
5.18 Richtungsvektor eines Pixels 50

Literaturverzeichnis

[Alha19] Alhashim, Ibraheem; Wonka, Peter: „*High Quality Monocular Depth Estimation via Transfer Learning*“, https://arxiv.org/pdf/1812.11941v2.pdf, 2019, letzter Zugriff: 21.02.2020

[Beta20] Beta FPV: „*Beta85X 4K Whoop Quadcopter (4S)*“, https://betafpv.com/collections/brushless-hd-drones/products/beta85x-4k-whoop-quadcopter-4s, letzter Zugriff: 21.02.2020

[Bjor18] Bjorck, Johan; Gomes, Carla; Selman, Bart; Weinberger, Kilian Q.: „*Understanding Batch Normalization*“, https://arxiv.org/pdf/1806.02375.pdf, 2018, letzter Zugriff: 15.02.2020

[CUDA20] CUDA: „*High Performance Computing*“, https://developer.nvidia.com/cuda-zone, letzter Zugriff: 13.02.2020

[Dens20] GitHub: „*DenseDepth*“, https://github.com/ialhashim/DenseDepth, letzter Zugriff: 10.01.2020

[Dock20] Docker: „*Debug your app, not your environment*“, https://www.docker.com/, letzter Zugriff: 10.02.2020

[Farn03] Gunnar Farnebäck „*Two-Frame Motion Estimation Based on Polynomial Expansion*“, http://www.diva-portal.org/smash/get/diva2:273847/FULLTEXT01.pdfTwo-Frame, 2003, letzter Zugriff: 21.02.2020

[GitH20] GitHub: „*Nvidia - Docker*“, https://github.com/NVIDIA/nvidia-docker, letzter Zugriff: 10.02.2020

[Glor06] Glorot, Xavier; Bengio, Yoshua: „*Understanding the difficulty of training deep feedforward neural networks*“, http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf, 2006, letzter Zugriff: 11.02.2020

[Huan18] Huang, Gao; Liu, Zhuang; van der Maaten, Laurens; Weinberger, Kilian Q.: „*Densely Connected Convolutional Networks)*“, https://arxiv.org/pdf/1608.06993.pdf, 2018, letzter Zugriff: 21.01.2020

[Imag16] ImageNet: „*Image Database*“, http://www.image-net.org/, letzter Zugriff: 21.01.2020

[Jabb14] Jabbar, Haider Khalaf; Khan, Rafiqul Zaman: „*Methods To Avaoid Over-Fitting And Under-Fitting In Supervised Machine Learning (Comperative Study)*“, https://www.researchgate.net/profile/Haider_Allamy/publication/295198699_METHODS_TO_AVOID_OVER-FITTING_AND_UNDER-FITTING_IN_SUPERVISED_MACHINE_LEARNING_COMPARATIVE_STUDY/links/56c8253f08aee3cee53a3707.pdf, 2014, letzter Zugriff: 20.01.2020

[Kera20] Keras: „*Documentation for Keras, the Python Deep Learning library*“, https://keras.io/, letzter Zugriff: 10.01.2020

[Kesk17] Keskar, Nitish Shirish; Mudigere, Dheevatsa; Nocedal, Jorge; Smelyanskiy, Mikhail; Tang, Ping Tak Peter: „*On Large-Batch Training For Deep Learning: Generalization Gap And Sharp Minima*“, https://arxiv.org/pdf/1609.04836.pdf, 2017, letzter Zugriff: 21.02.2020

[King15] Kingma, Diederik P.; Ba, Jimmy Lei: „*ADAM: A Method for Stochastic Optimization*“, https://arxiv.org/pdf/1412.6980.pdf, 2015, letzter Zugriff: 11.02.2019

[KITO20] GitHub: „*Keras Inference Time Optimizer*“, https://github.com/ZFTurbo/Keras-inference-time-optimizer, letzter Zugriff: 10.02.2020

[KPMG20] KPMG: „*Impact Of Autonomous Vehicles In Public Transport*“, https://assets.kpmg/content/dam/kpmg/ie/pdf/2017/07/ie-impact-av-vehicles-public-transport-2017.pdf, letzter Zugriff: 10.01.2020

[Kuma08] Kumar, M. Ramesh: „*New Formulas and Methods for Interpolation, Numerical Differentiation and Numerical Integration*“, https://arxiv.org/pdf/0809.0465.pdf, 2008, letzter Zugriff: 21.02.2020

[LeCu98] LeCunn, Yann; Bottou, Leon; Bengio, Yoshua; Haffner, Patrick; Tang: „*GradientBased Learning Applied to Document Recognition*“, http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf, 1998, letzter Zugriff: 21.02.2020

[Levi04] Levin, Anat; Kischinksi, Dani; Weiss, Yair: „*Colorization using Optimization*“, https://webee.technion.ac.il/people/anat.levin/papers/colorization-siggraph04.pdf, 2004, letzter Zugriff: 19.02.2020

[Math20] MathWorks: „*MatLab*“, https://www.mathworks.com/products/matlab.html, letzter Zugriff: 21.02.2020

[McCu43] McCulloch, Warren S.; Pitts, Walter: „*A logical calculus of the ideas immanent in nervous activity*“, https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf, 1943, letzter Zugriff: 11.02.2020

[Mül17] Müller, Andreas C.; Guido, Sahra: *Einführung in Machine Learning mit Python*, 1. Aufl., O'REILLY 2017

[Nvid20] Nvidia: „*Marktführer für Visual Computing*“, https://www.nvidia.com/de-de/, letzter Zugriff: 10.02.2020

[Nyud12] NYU Depth V2: „*Indoor Segmentation and Support Inference from RGBD Images*“, https://cs.nyu.edu/~silberman/datasets/nyu_depth_v2.html, 2012, letzter Zugriff: 21.02.2020

[Ohio20] Ohio State University: „*The Future of Driving*“, https://onlinemasters.ohio.edu/blog/the-future-of-driving/, letzter Zugriff: 10.01.2020

[Open20] OpenCV: „*Optical Flow*“, https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html, letzter Zugriff: 21.02.2020

[Pagl14] Pagliari, D.; Menna F.; Roncella R.; Remondino, F.; Pinto, L.: „*Kinect Fusion Improvement using Depth Camera Calibration*“, https://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XL-5/479/2014/isprsarchives-XL-5-479-2014.pdf, 2014, letzter Zugriff: 20.01.2020

[Pyth20] Python: „*The official home of the Python Programming Language*“, https://www.python.org/, letzter Zugriff: 10.01.2020

[Rasc17] Raschka, Sebastian: *Machine Learning mit Python*, 1. Aufl., mitp 2017

[Rash17] Rashid, Tariq: *Neuronale Netze selbst programmieren*, 1. Aufl., O'REILLY 2017

[Rude17] Ruder, Sebastian: „*An overview of gradient descent optimization algorithms*“, https://arxiv.org/pdf/1609.04747.pdf, 2017, letzter Zugriff: 21.02.2020

[Rume86] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J.: „*Learning representations by back-propagating errors*“, http://www.cs.toronto.edu/~hinton/absps/naturebp.pdf, 1986, letzter Zugriff: 10.02.2020

[Simo13] Simonot, Lionel; Hébert, Mathieu: „*Between additive and subtractive color mixings: intermediate mixing models*“, https://hal.archives-ouvertes.fr/hal-00962256/document, 2013, letzter Zugriff: 21.02.2020

[Spyd18] Spyder: „*The Scientific Python Development Environment*“, https://www.spyder-ide.org/, letzter Zugriff: 10.01.2020

[Tens20] TensorFlow: „*An end-to-end open source machine learning platform*“, https://www.tensorflow.org/, letzter Zugriff: 10.01.2020

[Werb19] Werbos, Paul J.: „*Beyond Regression: new tools for prediction and analysis in the behavioral sciences*“, https://https://www.researchgate.net/publication/35657389_Beyond_regression_new_tools_for_prediction_and_analysis_in_the_behavioral_sciences, 1974, letzter Zugriff: 21.02.2020

Abbildungen

[Ameb20] Ameba Internet of Things: „*Introduction to Ameba Quadcopter*“, https://www.amebaiot.com/en/ameba-arduino-quadcopter, letzter Zugriff: 21.02.2020

[Blen20] Blender Artists: „*I want the greyscale Z-depth map*“, https://blenderartists.org/t/i-want-the-greyscale-z-depth-map/540637, letzter Zugriff: 21.02.2020

[Medi18] Medium: „*Autoencoders made simple*“, https://towardsdatascience.com/autoencoders-made-simple-6f59e2ab37ef, letzter Zugriff: 21.02.2020

[Prab18] Understanding of Convolutional Neural Network (CNN): „*Deep Learning*“, https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148, 2018, letzter Zugriff: 21.02.2020

[Quor19] Quora: „*Max Pooling*“, https://www.quora.com/What-is-max-pooling-in-convolutional-neural-networks, letzter Zugriff: 21.02.2020

[Towa18] Towards Data Science: „*A Comprehensive Guide to Convolutional Neural Networks*“, https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, letzter Zugriff: 21.02.2020

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Markus Gross