Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Torben Geldermann

Implementierung eines faltenden neuronalen Netzwerks auf einer NVIDIA-CUDA-Plattform für Detektionsaufgaben

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Torben Geldermann

Implementierung eines faltenden neuronalen Netzwerks auf einer NVIDIA-CUDA-Plattform für Detektionsaufgaben

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. -Ing. Hans Peter Kölzer
Zweitgutachter : Prof. Dr. -Ing. Andreas Meisel

Abgegeben am 21. Januar 2019

Torben Geldermann

Thema der Bachelorthesis

Implementierung eines faltenden neuronalen Netzwerks auf einer NVIDIA-CUDA-Plattform für Detektionsaufgaben

Stichworte

KNN CUDA NVIDA CNN YOLO DARKNET

Kurzzusammenfassung

Diese Arbeit umfasst die Erstellung und Implementierung eines faltenden neuronalen Netzwerks auf einer NVIDIA-CUDA-Plattform für Detektionsaufgaben von Schiffen

Torben Geldermann

Title of the paper

KNN CUDA NVIDA CNN YOLO DARKNET

Keywords

KNN CUDA NVIDA CNN YOLO DARKNET

Abstract

This thesis includes the implementation of a convolution neural network on an NVIDIA CUDA platform for ship detection tasks.

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während meines Studiums und besonders bei der Erstellung dieser Arbeit unterstützt haben. Zuerst gebührt mein Dank meinem Erstprüfer und Aufgabensteller, Prof. Dr.Ing. Hans Peter Kölzer, für die Vergabe dieses äußerst interessanten Themas. Ich bedanke mich für die Unterstützung, das entgegengebrachte Vertrauen sowie die Möglichkeit diese interessante Arbeit praktisch umzusetzen. Zudem danke ich Herrn Prof. Dr.-Ing. Andreas Meisel für die freundliche Übernahme des Zweitprüfers bei dieser Arbeit. Auch möchte ich mich bei meiner Familie bedanken, insbesondere meinen Eltern und meiner Schwester, die mir mein Studium ermöglicht und mich in all meinen Entscheidungen unterstützt haben. Abschließend möchte ich mich bei meiner Freundin bedanken, die mich während des Studiums, insbesondere in der Abschlussphase motiviert hat und mir zur Seite stand.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CNN/ConVNet Convolutional Neural Network

ANN Artificial Neural Network

TK1 Jetson TK1

CPU Central Processing Unit

GPU Grafic Processing Unit

CUDA Compute Unified Device Architecture

KNN Künstliches Neuronales Netz

MLP Multilayer Perceptron

z.B. zum Beispiel

SSD Single Shot Detectors

ReLU Rectified linear unit, dt. rektifizierte lineare Einheit

API Application Programming Interface

FPS frames per second

OpenCV Open Source Computer Vision

YOLO You Only Look Once

KI Künstliche Intelligenz

ca. cirka

1. Einleitung

Diese Bachelorarbeit beschreibt die Erstellung und Implementierung eines faltenden neuronalen Netzwerks auf einer NVIDA-CUDA-Plattform zum Detektieren von Schiffen in Echtzeit.

1.1. Einführung in die Thematik

Ob in der Medizientechnik, als auch in der Automobilindustrie werden neuronale Netze in vielen Bereichen eingesetzt. Besonders geeignet sind diese beim Lösen von Problemen, bei dem ein systematisches Lösen oft schwierig ist und viele teils unpräzise Daten zu einem konkreten Ergebnis führen müssen. Einige Anwendungsbeispiele sind z.B. die Bilderkennung, Spracherkennung oder Schrifterkennung [13, vgl.]. Auch kreative Aufgaben, wie z.B. das Kreieren von Bildern oder das Komponieren von Musik, können immer mehr von künstlichen neuronalen Netzen übernommen werden. Speziell das Deep Learning besitzt das Potential in der Zukunft ganze Branchen positiv als auch negativ zu revolutionieren.

In dieser Thesis soll die Echtzeitfähigkeit von unterschiedlichen faltenden neuronalen Netzen auf einer NVIDIA-CUDA-Plattform für das Erkennen und Detektieren von Schiffen realisiert und anschließend analysiert werden. Dafür werden Convolutional Neural Networks, zu Deutsch „Faltende neuronale Netzwerke“ (CNN) eingesetzt. Die CNNs sind künstliche neuronale Netze, dessen Architektur unter anderem speziell für das Verarbeiten von Bildern entwickelt wurde [2, vlg.]. Im Laufe dieser Bachelorarbeit werden diese unterschiedlichen Netzwerke zu erst trainiert und anschließend auf der NVIDIA-CUDA-Plattform implementiert.

1.2. Aufbau der Arbeit

Diese Arbeit beschreibt zuerst die Analyse in Verbindung mit den Anforderungen im Kapitel 2. In diesem Kapitel wird analysiert, welche Hardware so wie Software den Anforderungen der Aufgabe entspricht und unter welchen Aspekten diese für die Arbeit gewählt wurden. Anschließend werden im Kapitel 3 die wesentlichen Grundlagen welche für diese Arbeit benötigt werden aufgearbeitet. Das Kapitel 4 zeigt die Realisierung der Arbeit. Unter anderem wird dort die Software und der Aufbau der unterschiedlichen Netzwerke genauer erläutert. Zusätzlich wird im Unterkapitel 4.3 beschrieben, wie die Trainings-/Testdaten beschafft, aufgearbeitet und erstellt werden. Im vorletzten Kapitel 5 werden die unterschiedlichen Netzwerke auf ihre Performance und Echtzeitfähigkeit getestet bzw. verifiziert. Das letzten Kapitel 6 gibt eine Zusammenfassung der Arbeit und einen Ausblick für die Zukunft.

2. Analyse und Anforderungen

In diesem Kapitel werden zunächst die Anforderungen und Ziele der Arbeit analysiert. Anschließend wird mit dem daraus resultierenden Ergebnis eine Auswahl der Hard- und Software getroffen. Eine Analyse der Anforderungen ist bei der Projektplanung fundamental, da durch Unklarheiten oder spätere Veränderungen ein hoher Kostenaufwand im Projektverlauf entstehen kann. Diesen gilt es zu vermeiden.

2.1. Anforderungen und Ziele der Arbeit

Die Anforderungen und Ziele der Arbeit sind wie folgt klar definiert: Es soll ein faltendes neuronales Netzwerk zur Detektion in Echtzeit auf einer möglichst kleinen und kostengünstigen Hardware implementiert werden.

2.2. Hardware

Auf Grund der Anforderungen und Recherchen stellt sich heraus, dass ein extrem hoher parallelisierbarer Rechenaufwand für diese Aufgabe benötigt wird. Da bekanntlich eine Grafic Processing Unit (GPU) bei hochgradig parallelisierbaren Programmabläufen signifikant schneller arbeitet als die CPU wird ein System benötigt, welches eine eigene GPU besitzt [16, vgl.]. Die meisten Frameworks für neuronale Netze besitzen eine GPU Unterstützung. Diese beschränkt sich auf Grund der Effektivität zum größten Teil auf Compute Unified Device Architecture (CUDA) unterstützte Grafikkarten von NVIDIA. Diese Erkenntnis schränkt die Suche nach einem kleinen günstigen System stark ein. Durch weitere Recherchen konnte die Wahl des Systems auf drei eingegrenzt werden. Alle drei folgenden Boards sind Entwicklerboards von NVIDIA und besitzen eine GPU, welche CUDA unterstützt.

Typ	Jetson TK1 Developer Kit	Jetson TX1 Developer Kit	Jetson TX2 Developer Kit
GPU	NVIDIA Kepler "GK20A"GPU with 192 CUDA cores	NVIDIA Maxwell GPU with 256-cores	256-core Pascal GPU
CPU	NVIDIA 4-Plus-1 quad-core ARM Cortex-A15	Quad-core ARM Cortex-A57	Dual-core NVIDIA Denver2 + quad-core ARM Cortex-A57
DRAM	2GB DDR3L 933MHz	4GB LPDDR4	8GB LPDDR4 128-bit interface
Storage	16 GB fast eMMC	16GB eMMC	32GB eMMC
Size	about 127x127mm	about 222x222mm	about 170x170mm
Infos	-	Dual IPSs (Image Singal Processors)	Dual IPSs (Image Singal Processors)
Power	<11W typical	10-15W typical	7,5W typical
Price*	about 160 Euro	About 440 Euro	about 500 Euro

Tabelle 2.1.: Vergleich Entwicklerboards Jetson TK1,TX1 und TX2 von NVIDIA [5, 6, 7, *siliconhighway.com 06.12.2018]

In der Tabelle 2.1 ist die engere Auswahl der recherchierten Entwicklerboards zu sehen. Zuerst das Jetson TK1: Dieses Board ist das erste Entwicklerboard von NVIDIA mit einer GPU, welche CUDA unterstützt. Das Board besitzt 192 CUDA cores und ist somit einzuordnen mit der Leistung bei einer NVIDIA GeForce GT 720 Grafikkarte. Die beiden Boards Jetson TX1 und TX2 besitzen 256 CUDA cores und sind vergleichsweise knapp über der GeForce GTX 285. Der Stromverbrauch aller Boards ist vergleichbar mit einer Grafikkarte sehr gering und liegt bei allen drei zwischen 7,5W und 15W. Der RAM hat sich mit jedem Board verdoppelt. Das Jetson TK1 besitzt 2GB DDR3, das TX1 4GB DDR4 und das TX2 8GB DDR4 RAM. Zwei weitere wichtige Punkte sind die Abmaße und der Preis. Dort liegt das Jetson TK1 deutlich vorne mit 127x127mm bei einem Preis von ungefähr 160Euro. Dieses ist ein gutes Preis-Leistungs-Verhältnis. Aufgrund der Anforderung, dass es möglichst klein und günstig sein sollte, fällt die Entscheidung auf das Jetson TK1. Anlässlich der erhöhten Datenmengen beim Training der Netzwerke wird dort ein weitaus stärkeres System benötigt. Hierfür wird ein PC-Tower verwendet, der als CPU einen I7-3770 mit 3.40GHz, 16GB Ram und eine NVIDIA GeForce GTX 970 mit 1664 CUDA cores besitzt. Dabei wird die Trainingszeit der einzelnen Netzwerke stark reduziert.

2.3. Software

Nachdem die Auswahl im Kapitel Hardware 2.2 auf das Jetson TK1 fällt, befasst sich dieser Teil mit der Auswahl der passenden Software für das TK1. Recherchen im Internet ergeben, dass oft die gleichen Frameworks mit GPU Unterstützung genannt werden. Diese sind Tensoflow und Caffe. Tensorflow erschien 2015 und wurde von Google entwickelt. Tensorflow wurde zunächst als internes Tool von Google geplant, bis es schließlich als Open-Source-Projekt zugänglich gemacht wurde. Der Name Tensorflow beschreibt das Verfahren des Frameworks. „Dies arbeitet mit sogenannten Data-Flow-Graphen, in denen Bündel von Daten ("Tensors") durch eine Reihe von Algorithmen verarbeitet werden, die durch einen Graph beschrieben sind."[28]. Bei weiteren Untersuchungen stellt sich heraus, dass Tensorflow ein 64-Bit fähiges System benötigt. Das vorhandene Jetson TK1 besitzt lediglich ein 32-Bit System. Somit fällt die erste Wahl auf das Caffe Framework. Caffe erschien im Jahr 2014 und wurde von Yangqing Jia während seiner Zeit am Vision and Learning Center der University of California, Berkeley entwickelt. Caffe ist komplett in C++ geschrieben und ist somit für hohe Geschwindigkeiten ausgelegt. Im Internet sind einige Beiträge zum Thema Installation von Caffe auf dem Jetson TK1 zu finden. Die Installation ist dennoch schwieriger als gedacht. Viele Libarys müssen per Hand installiert und Anpassungen an dem Framework vor dem Compilieren durchgeführt werden. Unter anderem muss die LMDB-MAP-SIZE in src/caffe/util/db.cpp, auf Grund des 32-Bit Systems manuell angepasst werden. Weitere Tests zeigen, dass dieses Framework sehr speicherlastig und das Board dafür nicht geeignet ist. Auf der Suche nach Alternativen zeigt sich, dass Darknet sehr gut in das gesuchte Profil passt. Das Darknet Framework wurde von Joseph Redmon 2013 entwickelt und wird als Open-Source Software zur Verfügung gestellt. Dieses soll schnell, schmal und einfach zu bedienen sein [20, vgl.]. Außerdem gibt es im Internet Lösungen, die eine Portierung der Netzwerke zu Caffe ermöglichen.

Für die Bildverarbeitung wird die OpenCV Libary verwendet. Sowohl OpenCV als auch Darknet besitzen eine Python API, welche die Programmierung erheblich erleichtert. Somit fällt die Auswahl der Programmiersprache auf Python.

3. Grundlagen

Dieses Kapitel gibt einen Überblick über die Grundlagen, die in dem Entwurf der Anwendung mit einbezogen werden.

3.1. Jetson TK1

In diesem Abschnitt werden die Grundlagen der Software und die des Entwicklerboards dargestellt. In der folgenden Abbildung 3.1, ist die Architektur des Entwicklerboards dargestellt.

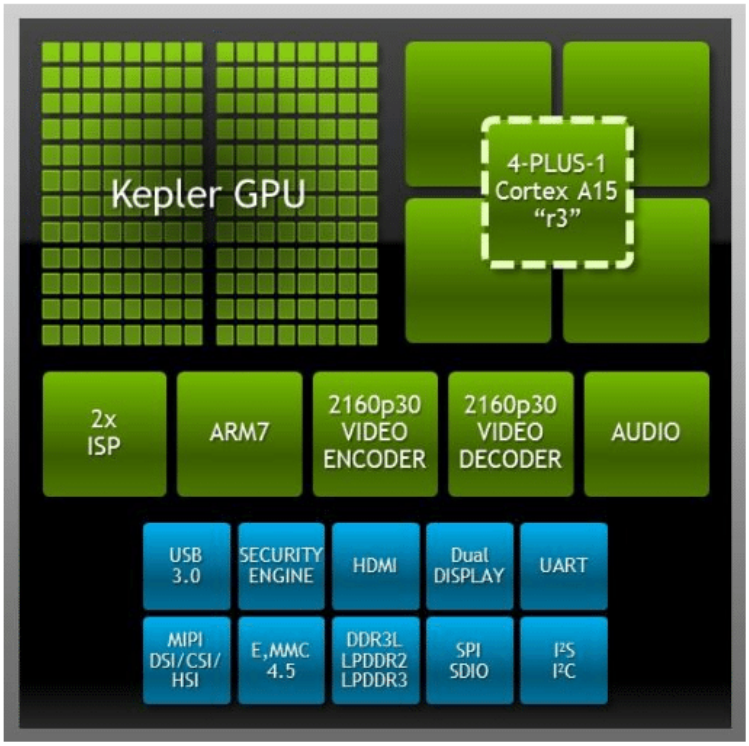

Abbildung 3.1.: Architecture of Tegra TK1 [17]

Als GPU besitzt das Board eine Kepler GK20A GPU, welche in 28 nm gefertigt ist. Kepler ist der Name für die GPU Mikroarchitektur, welche 2012 von NVDIA auf dem Markt gebracht wurde. Als CPU arbeitet ein Cortex A15 Quad-Core mit 2,3 GHZ. Zusätzlich gibt es noch einen Kern zum Batterie sparen. Das Board besitzt außerdem die gängigen Anschlussarten, um es als eigenständiges System zu verwenden. Diese wären z.b.: ein USB 3.0, ein HDMI Anschluss, sowie einen fest verbauten 16GB eMMC-Speicher. Nachfolgend ist in der Abbildung 3.2 das Jetson TK1 Entwicklerboard dargestellt. Um sich einen Eindruck über die Größe zu verschaffen, wurde ein handelsübliches Geodreieck mit einer Kantenlänge von 14cm daneben gelegt.

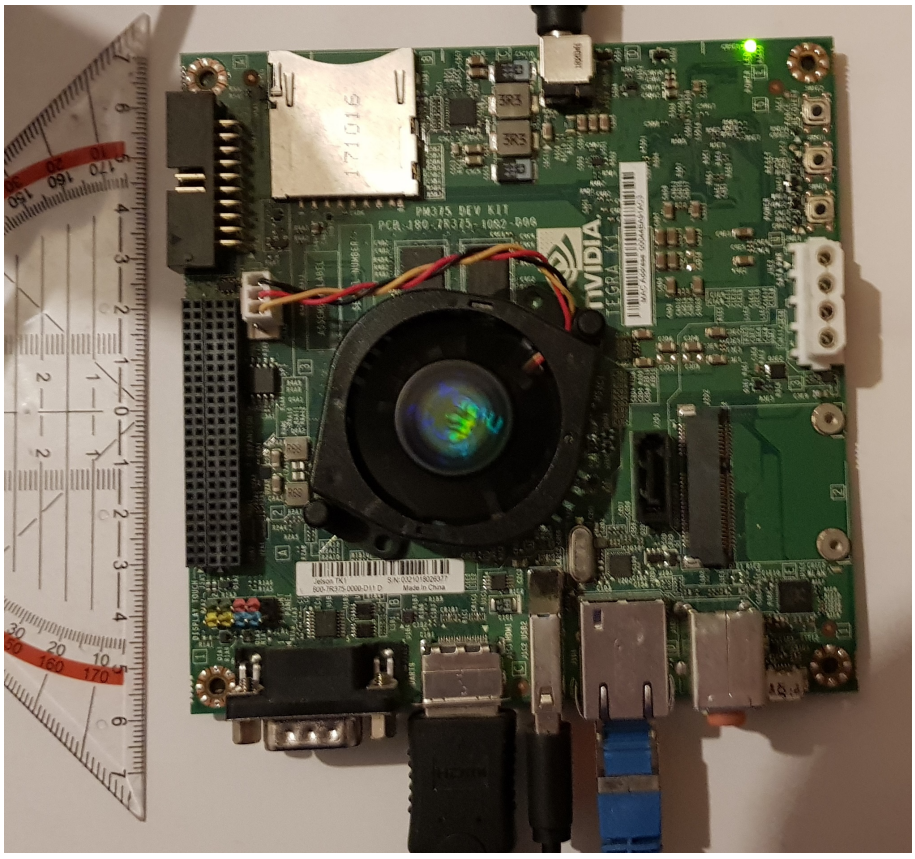

Abbildung 3.2.: Darstellung Jetson TK1 im Betrieb

3.1.1. Betriebssystem

Das Betriebssystem Linux For Tegra R21.5 (L4T R21.5) wird direkt von NVIDIA gestellt und basiert auf einem 32-Bit Ubuntu 14.04 System mit NVIDIA Treiber. Mit der Installation des All-in-one Paketes besitzt das Jetson TK1 bereits die benötigte CUDA Libary. Zusätzlich ist ein OpenCV (OpenCV4Tegra) installiert, welches speziell für die Hardware optimiert wurde.

3.1.2. CUDA

Wie bereits im Kapitel 2.2 beschrieben benötigt das System eine NVIDIA-CUDA Unterstützung. CUDA ist eine spezielle Architektur für parallele Berechnungen, welche von NVIDIA entwickelt wurde. Durch die Nutzung des Grafikprozessors entsteht eine deutliche Leistungssteigerung. Die große parallele Rechenleistung ergibt sich aus der Architektur der GPU. Die folgende Abbildung 3.3 zeigt einen Vergleich zwischen einer CPU und GPU.

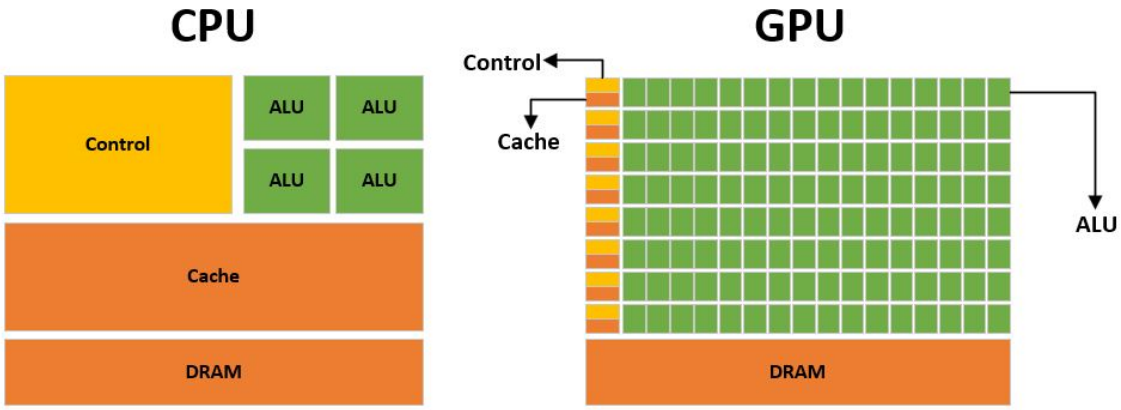

Abbildung 3.3.: Architektur CPU VS GPU

Die GPU besitzt einen kleineren Cache und Befehlssatz als die CPU. Jedoch besitzt die GPU im Vergleich zu einer modernen CPU ein Vielfaches an einfachen Recheneinheiten. Auf Grund dessen ist es möglich, mit ihr tausende parallelisierte Hardware-Threads zu benutzen. Der Prozessablauf von CUDA ist in der folgenden Abbildung 3.4 vereinfacht dargestellt.

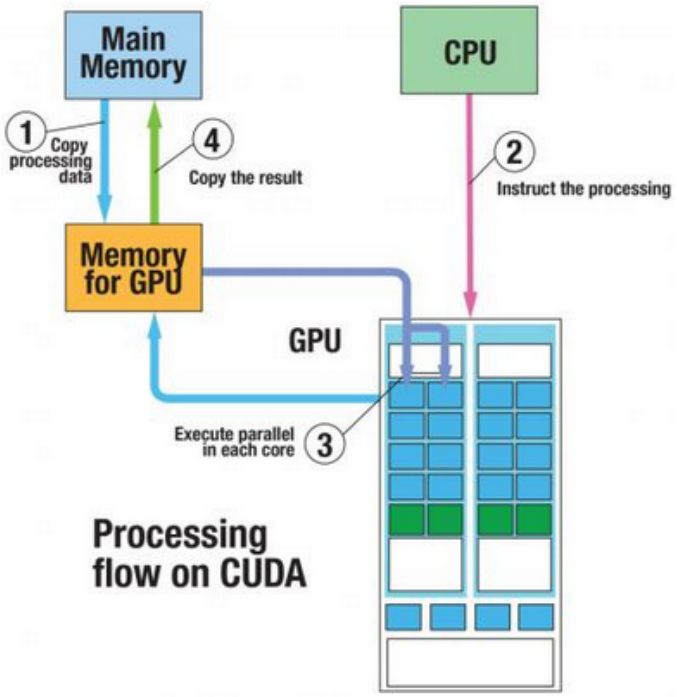

Abbildung 3.4.: CUDA Prozessablauf [27]

Zunächst werden die zu bearbeitenden Daten von dem Hauptspeicher in den Speicher für die GPU kopiert (1). Im zweiten Schritt (2) bekommt die GPU Anweisungen von der CPU zum bearbeiten der Daten. Im vorletzten Schritt (3) werden die Daten parallel in den einzelnen Kernen der GPU verarbeitet. Als letztes (4) wird das Ergebnis vom Speicher der GPU zurück in den Hauptspeicher geschrieben.

3.1.3. Programmiersprache

Als Programmiersprache wird die Skriptsprache Python verwendet. Wie bereits erwähnt, besitzt das Darknet Framework, welches in C geschrieben ist, eine Python API. Zusätzlich verfügt OpenCV eine Libary für Python. Diese wurde am Anfang der 1990er Jahre von Guido van Rossum in Amsterdam entwickelt. Er wollte eine einfach zu erlernende Programmiersprache entwickeln. Python wird ständig von vielen Entwicklern weltweit in der Python Software Foundation weiterentwickelt [12, vlg.]. Der Interpreter ist auf allen gängigen Betriebssystemen (Linux, MacOS und Windows) portiert. Außerdem besitzt Python

alle wichtigen Programmier-Paradigmen, welche bei der heutigen Softwareentwicklung vorkommen. Diese sind unter anderem: objektorientierte, aspektorientierte und strukturierte Programmierung. Ein wichtiger Punkt spielt dabei die Syntax. Anders als z.B. in C oder anderen Programmiersprachen, bei denen Blöcke durch Klammern oder Schlüsselwörter markiert werden, wird dies bei Python durch Einrücken geregelt. Das lässt den Code sehr leserlich und aufgeräumt wirken. Ein Beispiel dafür ist in den folgenden zwei Listings 3.1 und 3.2 zu sehen.

```
1 //Einfache For-Schleife in C 0-99
2 for (int i = 0; i < 100; i++) {
3   printf("Zahl %d\n", i+1);
4 }
```

Listing 3.1: For-Schleife C

```
1 #!/usr/bin/python
2
3 #Einfach For-Schleife in Pyton 0-99
4 for i in range(0,100):
5   print "Zahl "+i
```

Listing 3.2: For-Schleife Python

Beide Codes zeigen eine For-Schleife. Einmal in C 3.1 und sowie in Python 3.2. Wie zu erkennen ist die For-Schleife, welche in Python Programmiert wurde, einfacher, ausdrucksstärker und weniger fehleranfällig.

3.1.4. OpenCV

Die Open Source Computer Vision Library (OpenCV) ist eine der meist verwendeten frei verfügbaren Libarys, welche Algorithmen für Bildverarbeitung und Maschinen Learning besitzt. Als freie Software steht sie unter der BSD-Lizenz zur Verfügung. Die OpenCV-Libary steht für die Programmiersprachen C, C++ und Python zur Verfügung [18]. Wie bei dem Darknet Framework besitzt die OpenCV Libary eine Unterstützung für CUDA Grafikkarten.

3.2. Neuronale Netze

In diesem Abschnitt werden die benötigten Grundlagen für die Neuronalen Netze vermittelt.

3.2.1. Natürliche neuronale Netze

Natürliche neuronale Netze stammen aus der Biologie. Das menschliche Gehirn enthält cirka 87 Milliarden Neuronen. Die Nuklei der Nervenzellen sind durch Axone und Dendriten, wie in der Abbildung 3.5 zusehen, stark miteinander vernetzt.

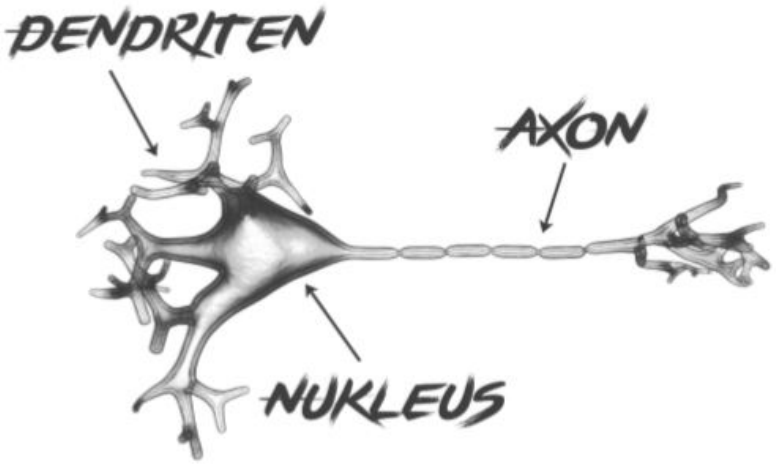

Abbildung 3.5.: Stark vereinfachter Aufbau einer Nervenzelle [1]

Über die Dendriten empfangen die Nervenzellen ihre Informationen von anderen Nervenzellen. Diese Informationen werden anschließend im Nukleus verarbeitet und über das Axon an andere Nervenzellen weitergeleitet. Diese Informationen werden über Chemische und elektrische Signale übertragen. Ein solches natürliches neuronales Netz lernt über die Entwicklung neuer Verbindungen zwischen den Nervenzellen dazu. [1, vgl.]

3.2.2. Künstliche neuronale Netze

Künstliche neuronale Netze besitzen viel Ähnlichkeit mit den natürlichen neuronalen Netzen. Ein wichtiger Grund warum neuronale Netze erst heute so großen Erfolg feiern können, ist

der Anstieg der Rechenleistung, mit dem viel größere Modelle betrachtet werden können. Anfänglich waren die Verbindungen zwischen Neuronen durch Hardware begrenzt. Heute ist es lediglich eine Designentscheidung. Einige dieser Netze weisen heutzutage nahe zu so viele Verbindungen pro Neuron wie eine Katze auf. [10, Vgl. S.24-25] Häufig wird in der Literatur lediglich von neuronale Netzen gesprochen, ohne das Wort „künstlich“ zu erwähnen. Der Grund hierfür ist, dass es in den Texten meistens offensichtlich ist, worum es sich handelt. In dieser Arbeit wird folgend mit neuronale Netze oder nur Netze ausschließlich das künstliche neuronale Netz gemeint.

3.2.3. Aufbau

Die Grundeinheit der Neuronalen Netze sind die Neuronen, auch Units genannt. Diese sind dafür da, Informationen von anderen Units oder aus der Umwelt aufzunehmen und sie an andere Units oder die Umwelt modifiziert weiterzugeben. In der folgenden Abbildung 3.6 ist der Aufbau eines einzelnen Neurons zu erkennen.

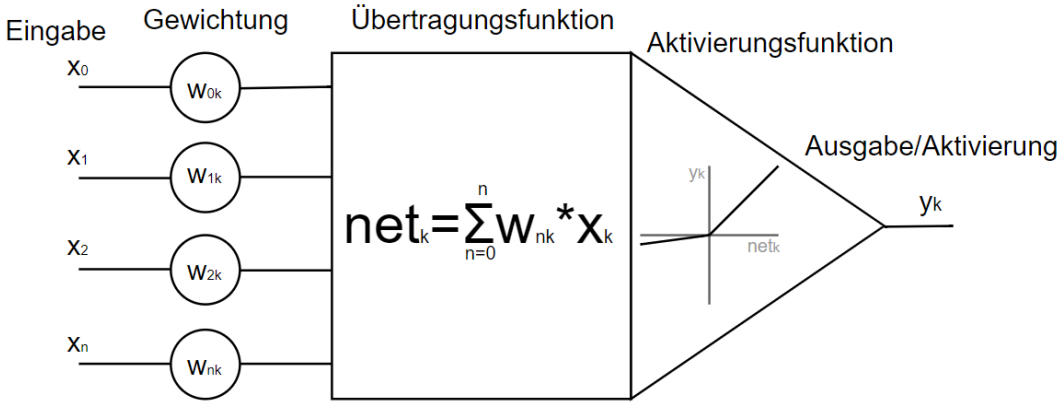

Abbildung 3.6.: Aufbau Neuron

Die Eingabeschicht nimmt die Information auf und leitet diese weiter. Mit der Gewichtung wird die Stärke der Verbindung zwischen den Neuronen ausgedrückt. Ist die Gewichtung positiv, hat ein Neuron auf ein anderes Neuron einen erregenden (exzitatorischen) Einfluss. Eine negative Gewichtung hingegen, hat einen hemmenden (inhibitorischen) Einfluss auf die anderen Neuronen. Bei einer Gewichtung von null, hat das Neuron keinen Einfluss auf die anderen Neuronen. Dieses Verhalten lässt sich mit der Formel 3.1 nachvollziehen. Die Aktivierungsfunktion wird im Kapitel 3.2.5 genauer erläutert.

$$net_k = \sum \lim_{n=0}^{n} W_{nk} \cdot X_k \tag{3.1}$$

Somit ist jedes einzelne Gewicht W_{nk} im Endeffekt nur eine reelle Zahl, welche mit dem Output-Wert X_k des entsprechenden Vorgängerneurons multipliziert wird. Jedes einzelne Neuron bildet eine gewichtete Summe aller Vorgängerneuronen.

3.2.4. Layer

In einem künstlichen neuronalen Netz sind die Neuronen bzw. Units schichtweise angeordnet in sogenannten Layern. Zwischen den Schichten ist jedes einzelne Neuronen mit allen Neuronen der darauf folgenden Schicht verbunden. Dies ist in der folgenden Abbildung 3.7 dargestellt.

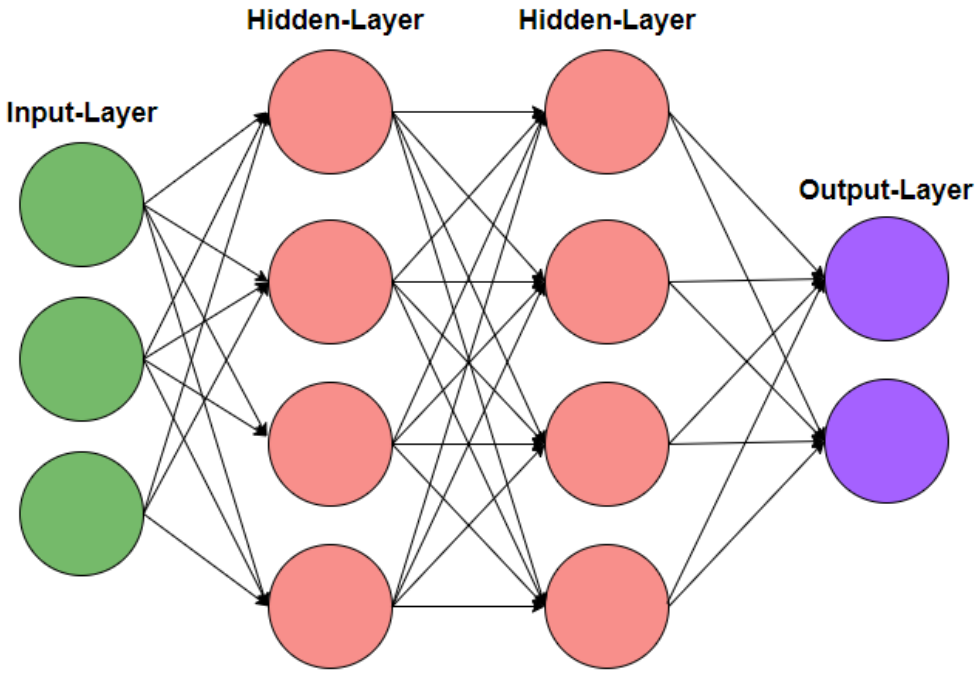

Abbildung 3.7.: Neuronales Netz

Beginnend mit der Eingabeschicht (Input-Layer). Dies ist der Startpunkt des Informationsflusses. Die einzelnen Signale werden von den Neuronen aufgenommen und gemäß der Gewichtung, Übertragungsfunktion und der Aktivierungsfunktion an die nächste Schicht (Hidden-Layer) weitergegeben. Auf Grund des Aufbaus gibt jedes Neuron seine Information an alle Neuronen der folgenden Schicht weiter. Jedes KNN hat mindestens eine Zwischenschicht (Hidden-Layer). Je mehr Hidden-Layer es in einem KNN gibt, desto tiefer und aufwendiger wird das Netz. Dann ist die Rede vom "Deep Learning". Die Anzahl von Hidden-Layern ist theoretisch unbegrenzt, erhöht jedoch die Anzahl der benötigten Berechnungen in

einem Netz. Nach den Zwischenschichten folgt als letztes die Ausgabeschicht, auch Output-Layer genannt. Diese enthält das Ergebnis der Informationsverarbeitung vom Netzwerk.

3.2.5. Aktivierungsfunktionen

Die Aktivierungsfunktion stellt die Verbindung zwischen dem Netzinput und dem Output (Aktivitätslevel) eines Neurons dar. Sie ermöglicht dem Netz Grenzwerte einer Entscheidung zu erlernen. Die Aktivierungsfunktion kann abhängig von der verwendeten Topologie des Netzes unterschiedlich gewählt werden. Es gibt unterschiedliche Arten von Aktivierungsfunktionen. Häufig werden folgende Funktionen verwendet: Sigmoid, TanH, ReLU und Varianten dieser Aktivierungsfunktionen. In den hier verwendeten Netzwerken wird in allen Layern, bis auf dem letzten, die Leaky ReLU Funktion verwendet. Diese ist eine Verbesserung der ReLU Funktion und vermeidet das sogenannte „Dying Problem“. Dying Problem bedeutet, dass alle negativen Werte bei dieser Funktion null sind. Dadurch können sogenannte „tote“ Neuronen entstehen. Neuronen sind tot, wenn diese auf der negativen Seite hängen bleiben und immer null ausgeben. Durch die Steigung von null im negativen Teil der RELU Funktion, ist es sehr unwahrscheinlich, dass sich so ein Neuron erholt. Solche Neuronen spielen keine Rolle bei der Diskriminierung der Eingabe und sind im wesentlichen unbrauchbar. Durch diesen Effekt kann es passieren, dass im Laufe der Zeit ein großer Teil des Netzwerks keine Funktion ausübt [4, vgl.]. Im letzten Layer wird eine lineare Funktion verwendet. In der folgenden Abbildung 3.8 ist die ReLU und die verbesserte Leaky ReLU Funktion dargestellt. Außerdem ist die einfache lineare Funktion dargestellt, welche im letzten Layer enthalten ist.

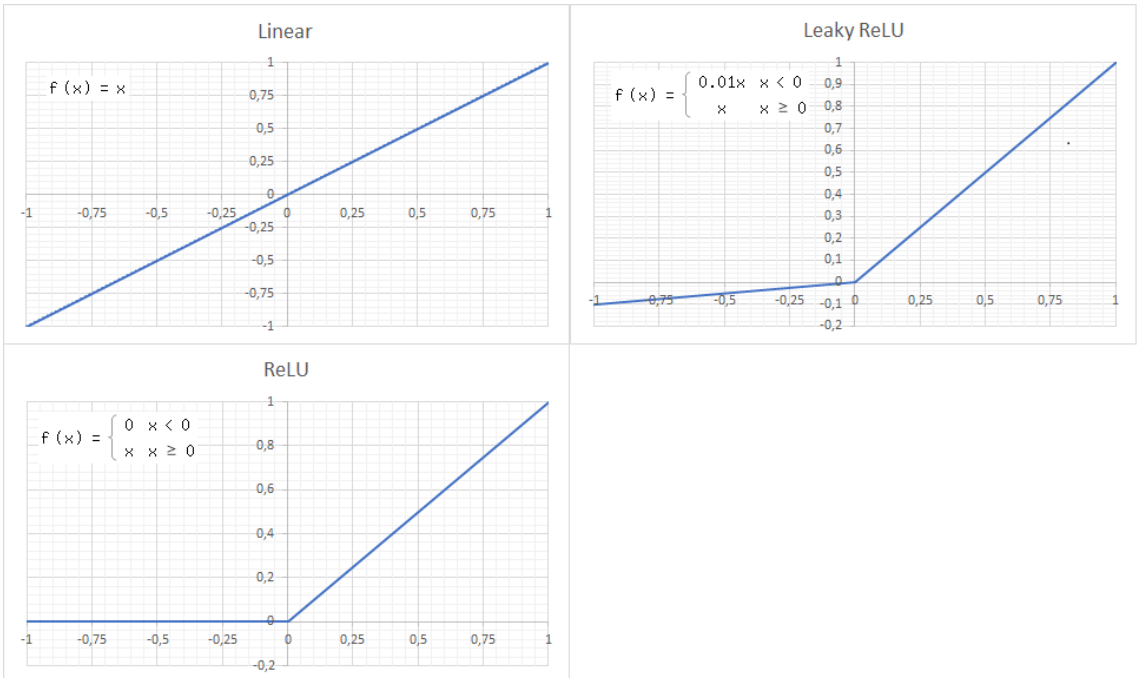

Abbildung 3.8.: Aktivierungsfunktionen

Wie in der Abbildung 3.8 zu erkennen, verlaufen alle drei Funktionen im positiven Teil linear, lediglich der negative Teil unterscheidet sich voneinander.

In der nachstehenden Abbildung 3.9 ist ein Beispiel für den Unterschied der ReLU und Leaky ReLU Funktion zu sehen. Als Beispiel wird eine 4x4 große Eingangsmatrix verwendet.

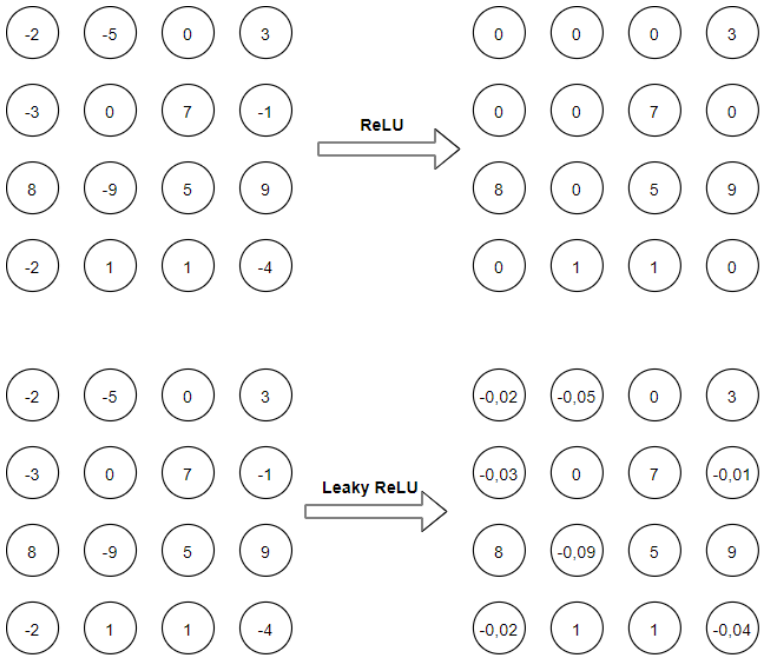

Abbildung 3.9.: Beispiel ReLU, Leaky ReLU

Wie in der obigen Abbildung 3.9 gut ersichtlich, ist der einzige Unterschied im negativen Bereich zu erkennen. Genau mit dieser kleinen Steigung im negativen Bereich wird das vorher erwähnte „dying Problem„verhindert.

3.3. Faltendes neuronales Netz

Es gibt viele verschiedene Arten von künstlichen neuronalen Netzen. Besonders gut für Bild- und Audioverarbeitung eignet sich ein faltendes neuronales Netz, auch Convolutional Neural Network (CNN/ConVNet) genannt. Ein CNN erkennt mit seinen unterschiedlichen Filtern ortsunabhängig Strukturen im Eingangsbild.

3.3.1. Unterschied zum einfachen Neuronalen Netz

Anders wie bei einem einfachen feedforward-Netz, verwenden die CNNs eine besondere Architektur, die sogenannten Convolutional und Pooling Layers. Der Sinn dabei ist, eine Betrachtung des Inputs aus verschiedenen Schichten zu haben. Jedes Neuron im Convolution Layer überprüft einen bestimmten Bereich der Input Matrix. Dies ist in der nachstehenden

Grafik 3.10 zu sehen. Aufgrund der Faltungen erhält diese Art der neuronalen Netze ihren Namen.

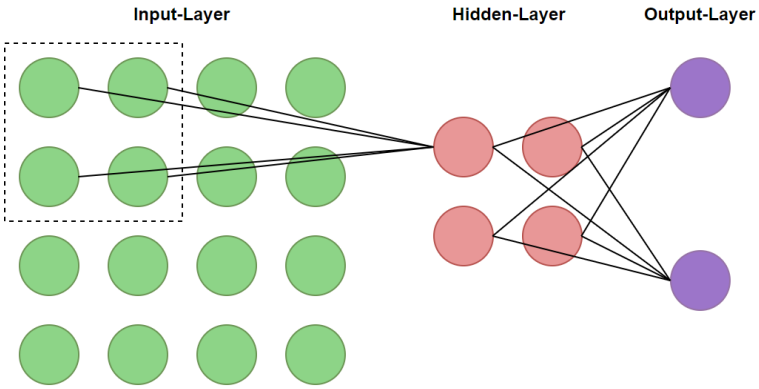

Abbildung 3.10.: Unterschied zum einfachen neuronalen Netz

3.3.2. Aufbau

In der nachstehenden Abbildung 3.11 ist der Aufbau und Ablauf eines CNNs zu sehen.

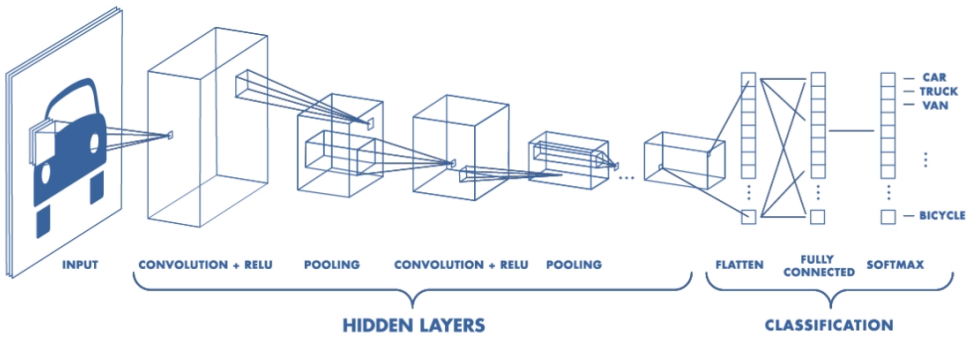

Abbildung 3.11.: Aufbau faltendes neuronales Netzwerk [19, Quelle]

Typisch für ein CNN sind fünf oder mehr Schichten. In jeder einzelnen Schicht wird eine Erkennung der Muster durchgeführt. Die erste Schicht erkennt primitive Muster, die zweite erkennt Muster von Mustern, die dritte Schicht wiederum Muster von den Mustern zuvor. Dieser Prozess wird fortlaufend durch die einzelnen Schichten weitergeführt. Dies passiert in den Hidden-Layers. Als letztes wird die Klassifizierung durchgeführt, sodass am Ende eine Wahrscheinlichkeit der einzelnen Klassen herauskommt. Dies wird Full Connected Layer

oder auch Dense Layer genannt. Es handelt sich um eine normale neuronale Netzstruktur, bei der alle Neuronen mit allen Inputs und Outputs verbunden sind. Die einzelnen Schritte werden in den nachfolgenden Abschnitten erklärt.

3.3.2.1. Convolution-Layer

Wie bereits in den Abschnitten 3.3.1 und 3.3.2 erwähnt, untersuchen die Neuronen im Convolution-Layer das Bild auf bestimmte Eigenschaften. Wie z.B. Farbzusammensetzung oder Kanten. Dies geschieht mit sogenannten Filtern. Diese Filter sind gewissermaßen nichts anderes als eine eigens dafür trainierte Schicht. Solche Filter werden mit Hilfe einer diskreten Faltung realisiert. Das Ergebnis eines Filters ist der gewichtete Input eines Bereiches und wird anschließend im Convolution-Layer gespeichert [11, vgl.]. Die Anzahlt der Convolution-Layer ist gleich die Anzahl der vorhanden Filter, da jeder Filter über das komplette Input Bild läuft. Die Formel 3.2 lässt sich anhand der nachstehenden Abbildung 3.12 leicht erklären.

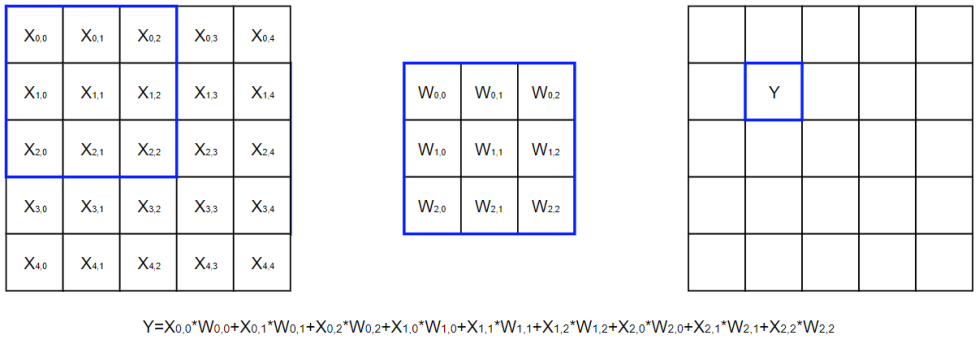

Abbildung 3.12.: Faltung mit 3x3 Filter

In der Abbildung 3.12 wird eine 5x5 Pixel große Eingabematrix mit einer 3x3 großen Filtermatrix gefaltet. Dazu wird das Skalarprodukt einer hier $3 \cdot 3$ Filtermatrix mit einem ebenfalls $3 \cdot 3$ großen Teil der Eingangsmatrix gebildet.

$$Y[i,j] = \sum \lim_{n=0}^{n} W_n k \cdot X_k \tag{3.2}$$

3.3.2.2. Pooling

Das Pooling ist ein Diskretisierungsprozess zwischen den unterschiedlichen Convolution-Layer. Dort wird mit Hilfe von Pooling überflüssige Information verworfen. Es gibt verschie-

dene Arten von Pooling. Die am meisten verbreitete Art ist das MaxPooling. Dabei laufen die Layer die durch die Filter erstellten Feature Map ab und komprimieren diese. Dies geschieht so wie in der Abbildung 3.13 dargestellt.

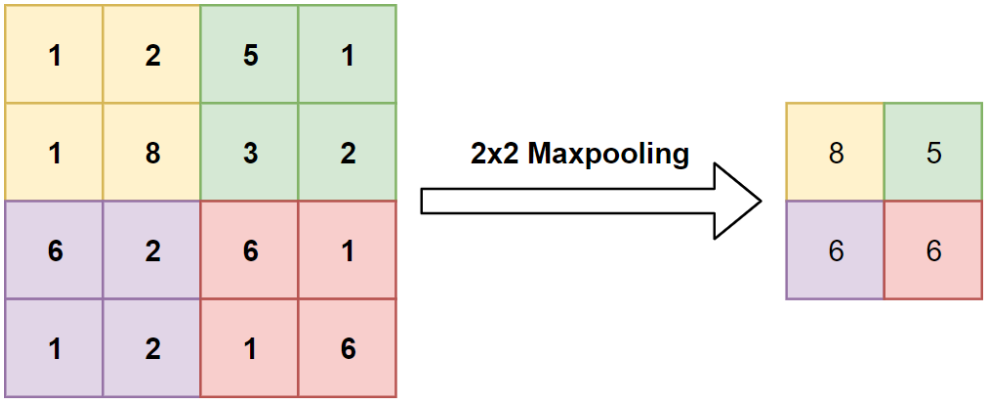

Abbildung 3.13.: Beispiel Maxpooling

$$Z_f = mean\{s\} = mean\{S_1, S_2, \ldots, S_n\} \tag{3.3}$$

Wie auch in der Abbildung zuvor 3.13, werden in dieser Arbeit 2x2 Maxpool-Layer verwendet. Durch diese Methode werden die Daten um 75 Prozent reduziert. Trotz dieser Reduzierung bleibt die Performance des Netzwerks in der Regel gleich. Das Pooling bietet einige wichtige Vorteile. Durch die Reduzierung der Daten wird das Netzwerk kleiner und der Rechenaufwand sinkt. Außerdem sorgt das Pooling präventiv gegen Überanpassung (Overfitting) vor. [9]

3.3.2.3. Flattening

Wie bereits im Absatz 3.3.2 erwähnt, wird am Ende der Hidden-Layer eine Klassifikation mit Hilfe einer normalen neuronalen Netzstruktur (MLP) durchgeführt. Dafür muss die Ausgabe des letzten Max Pooling Layers wieder in einen Vektor ausgerollt werden. Dies geschieht durch den Flatten Layer [2, vgl.].

3.3.2.4. Softmax

Der letzte Layer enthält bei Klassifizierungs-Problemen eine sogenannte Softmax-Aktivierung. Diese wandelt den Output aller Output-Neuronen in die unterschiedlichen Wahr-

scheinlichkeiten der einzelnen Klassen um. Sie bringt die Summe aller Output-Neuronen auf eins. Zu sehen in der nachstehenden Formel 3.4.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{3.4}$$

3.3.3. Trainingsablauf

Die Gewichte der einzelnen Filter und der Full Connected Layer werden zu Beginn entweder zufällig gewählt oder es werden vortrainierte Gewichte verwendet. Diese Gewichte werden durch das so genannte Backpropagation 3.3.3.1 während des Trainings ständig weiter optimiert.

Das untrainierte Netz berechnet für einen bestimmte Input-Datensatz ein Ergebnis und vergleicht dieses mit bekannten Ergebnissen des Beispiel-Datensatzes. Daraufhin wird der Fehler bzw. die Größe der Abweichung berechnet.

3.3.3.1. Backpropagation

Wie bereits im Abschnitt vorher beschrieben, geschieht das Training mit einem sogenannten Backpropagation-Algorithmus (Fehlerrückführung). Die Größe der berechneten Abweichung wird zurück bis zur Eingabeschicht in das Netz propagiert. Dabei wird diese anteilig auf das Neuron verteilt, welches maßgeblich am Fehler beteiligt war. Zusätzlich spielt die Learning Rate eine sehr große Rolle. Sie ist einer der wichtigsten Einstellgrößen von neuronalen Netzen. Viel genutzte Learning Raten sind 0,001 oder 0,01. Das bedeutet, es wird nur ein tausendstel bis ein hundertstel des errechneten Fehlers pro Durchlauf korrigiert. Dieser Zyklus wiederholt sich so lange, bis der Fehler klein genug ist. Mathematisch gesehen ist das Problem ein Finden von Minimalwerten (Fehlerminimierung). [2, vgl.]

3.3.3.2. IoU

IoU (Intersection over union) ist eine Auswertemethode zum Bestimmten der Genauigkeit einer Detektion. Diese vergleicht unter anderem beim Training die Bounding Boxes der Testbilder mit denen, die vom Modell ausgegeben werden. Dafür wird die Überlappung der beiden Boxen durch die Gesamtfläche dieser Boxen geteilt. Bildlich ist dies in der Abbildung 3.14 dargestellt.

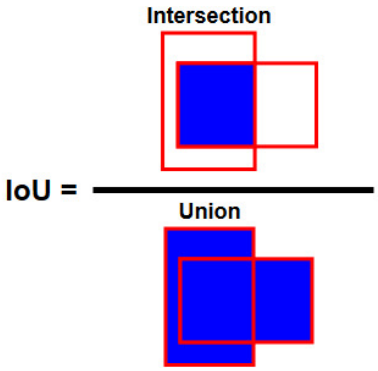

Abbildung 3.14.: Berechnung IoU

Anhand der Abbildung ergibt sich folgende Formel 3.5 für die Berechnung des IoU:

$$IoU = \frac{B_1 \wedge B_2}{B_1 \vee B_2} \tag{3.5}$$

Diese Werte liegen zwischen null und eins. Je näher der Wert an eins ist, desto genauer ist die Detektion. Ein IoU Wert von > 0.5 kann normalerweise als gute Vorhersage betrachtet werden. Ein Beispiel dafür ist in der nächsten Abbildung 3.15 dargestellt.

Abbildung 3.15.: Beispiel zur Berechnung von IoU [23]

3.3.4. Yolo

Ein einfaches CNN, welches oben in der Abbildung 3.11 beschrieben wird, eignet sich sehr gut zum Klassifizieren von Objekten. In dieser Arbeit soll jedoch eine zusätzliche örtliche Bestimmung der Objekte erfolgen (Detektion). Dafür wird das YOLO Verfahren von Redmon Et al. [22] verwendet. YOLO zählt zu den Single Shot Detectors. Der Vorteil von YOLO ist,

dass sowohl die Lokalisierung als auch die Klassifizierung in einem Durchgang geschieht und somit das Bild nur einmal betrachtet wird. Daher stammt der Name des Verfahrens: “You Only Look Once “(YOLO).

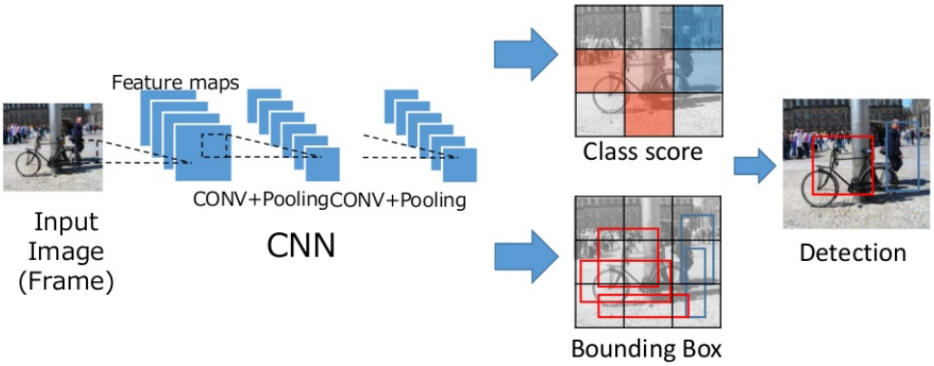

Abbildung 3.16.: YOLO Verfahren [14]

In der obigen Abbildung 3.16 ist der Ablauf eines YOLO-Netzwerkes dargestellt. Das Konzept von YOLO besteht darin, das Problem der Detektion als Regressionsproblem zu betrachten. Verglichen mit einem einfachen CNN besteht bei einem YOLO-Netz kein allzu großer Unterschied. lediglich der Letzte Layer Unterscheidet sich wesentlich. Während bei der Klassifizierung ein Vektor als Output entsteht, ist dies bei YOLO eine dreidimensionale Matrix. Dies ist in der blauen Markierung auf der folgenden Abbildung 3.17 dargestellt.

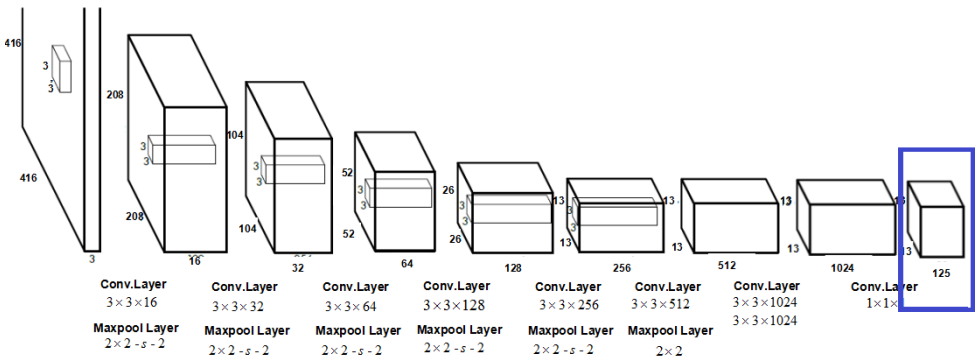

Abbildung 3.17.: tiny yolo Architektur [22]

Der Aufbau der Output-Matrix wird in der nachstehenden Abbildung 3.18 genauer erläutert.

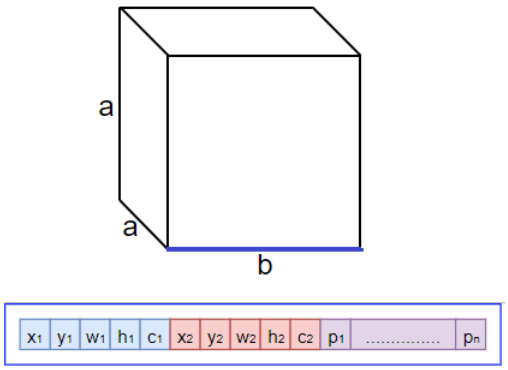

Abbildung 3.18.: Output-Box

Die Matrix aus der Abbildung 3.18 setzt sich wie folgt zusammen: die Dimension a ist abhängig von der Input-Größe des Bildes / 32. Der Standard ist ein Input von 448x448 und ergibt ein Grid von 7x7. Die Tiefe b besteht aus zwei Teilen. Zum ersten aus zwei Bounding Boxes und zum zweiten aus den Klassen. Eine Bounding Box besteht aus fünf Werten: die Position der Box (x,y), die Breite (w), Höhe (h) der Box und als letzter Wert eine Schätzung (c) der Intersection over Union (IoU). Der IoU besagt, wie bereits zuvor erläutert 3.3.3.2, wie gut eine Detektion mit der realen Position eines Objektes im Bild übereinstimmt [29, vgl.]. Mehr zum IoU ist aus dem Kapitel 3.3.3.2 zu entnehmen. Indem die Künstliche Intelligenz (KI) diesen Wert schätzt, entsteht ein Wert welcher ausdrückt, wie sicher sich die KI der Schätzung ist. Die letzten Werte entsprechen der Anzahl der vorhanden Klassen (p_n). Somit einspricht die Tiefe $b = 2 \cdot 5 + Klassen$. In der Nachstehenden Abbildung 3.19 sind die einzelnen Elemente dargestellt.

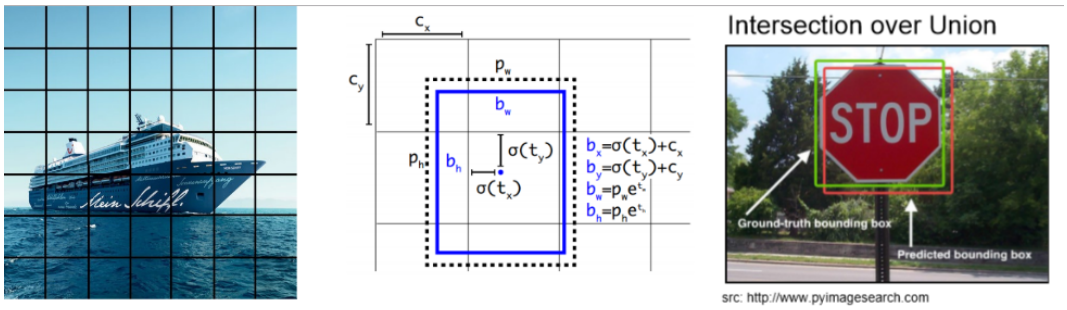

Abbildung 3.19.: Elemente der Output-Box

Somit ergeben sich für jedes einzelne Grid eine Klassifikation sowie Einschätzung der Bouding Boxes. Wenn der Mittelpunkt eines Objekts in ein Grid fällt, ist dies Grid für die Erkennung dieses Objekts verantwortlich.

4. Realisierung

In diesem Teil der Arbeit, geht es um die Realisierung und das Erstellen der Trainings- / Testdaten.

4.1. Framework

Wie bereits zuvor im Kapitel Software 2.3 erläutert, wird für diese Arbeit das Darknet Framework verwendet. Dies wird wie auf der Entwicklerseite [21] beschrieben für das Training auf dem PC und für die Detektion auf dem Jetson TK1 installiert. Damit das kompilieren funktioniert, ist es wichtig, dass die Makefiles auf die beiden Boards abgestimmt werden. Unter anderem muss die Flag für die GPU gesetzt werden, damit das Framework die Grafikkarte zum bearbeiten verwendet. In den nachfolgenden Listings 4.1 und 4.2 sind die Anpassungen der beiden Makefiles für die unterschiedlichen Systeme zu sehen.

```
GPU=1
CUDNN=1
OPENCV=1
OPENMP=0
DEBUG=0

ARCH= -gencode arch=compute_30,code=sm_30 \
      -gencode arch=compute_35,code=sm_35 \
      -gencode arch=compute_50,code=[sm_50,compute_50] \
      -gencode arch=compute_52,code=[sm_52,compute_52]
#      -gencode arch=compute_20,code=[sm_20,sm_21] \ This one is deprecated?

# This is what I use, uncomment if you know your arch and want to specify
ARCH= -gencode arch=compute_52,code=compute_52

ifeq ($(GPU), 1)
COMMON+= -DGPU -I/usr/local/cuda-9.2/include/
CFLAGS+= -DGPU
LDFLAGS+= -L/usr/local/cuda-9.2/lib64 -lcuda -lcudart -lcublas -lcurand
```

```
endif
```

Listing 4.1: Makefile Trainings-PC

```
GPU=1
CUDNN=0
OPENCV=1
OPENMP=0
DEBUG=1

ARCH= -gencode arch=compute_30,code=sm_30 \
      -gencode arch=compute_35,code=sm_35 \
      -gencode arch=compute_50,code=[sm_50,compute_50] \
      -gencode arch=compute_52,code=[sm_52,compute_52]
#      -gencode arch=compute_20,code=[sm_20,sm_21] \ This one is deprecated?

# This is what I use, uncomment if you know your arch and want to specify
ARCH= -gencode arch=compute_32,code=compute_32

ifeq ($(GPU), 1)
COMMON+= -DGPU -I/usr/local/cuda/include/
CFLAGS+= -DGPU
LDFLAGS+= -L/usr/local/cuda/lib -lcuda -lcudart -lcublas -lcurand
endif
```

Listing 4.2: Makefile Jetson-TK1

Werden diese beiden Listings 4.1 und 4.2 miteinander verglichen ist zu erkennen, dass bei dem Trainings-PC zusätzlich das CUDNN gesetzt ist. Dieses wird zum Trainieren der Netzwerke benötigt. Ein weiterer Punkt ist die Anpassung der Architektur (ARCH) an die unterschiedlichen Grafikkarten. Diese wird aus der CUDA Toolkit Dokumentation entnommen [15, 5.5. Virtual Architecture Feature List]. Die NVIDIA Geforce 970 GTX verwendet die Maxwell (compute_52) und das Jetson-TK1 die Kepler (compute_32) Architektur. Zusätzlich müssen die Pfade der CUDA Libary entsprechend angepasst werden. Wenn alles funktioniert, gibt die Eingabe von "*./darknet*"als Ausgabe "*usage: ./darknet <function>*"zurück. Zusehen in der Abbildung 4.1.

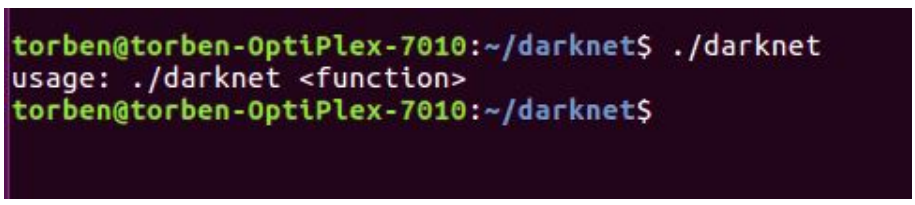

Abbildung 4.1.: Test des Darknet Frameworks

4.2. Aufbau der unterschiedlichen Netzwerke

In diesem Teil der Arbeit geht es um die Auswahl und den Aufbau der verwendeten Netzwerke (Detektoren). Wie bereits vorher erwähnt, fällt die Wahl des Netzwerks auf das Tiny-YOLOv2 Netzwerk. Die folgende Tabelle 4.1 begründet die Wahl des Netzwerks.

Model	Train	Test	mAP	FLOPS	FPS	required GPU-RAM
Old YOLO	VOC 2007+2012	2007	63,4	40,19 Bn	45	-
SSD300	VOC 2007+2012	2007	74,3	-	46	-
SSD500	VOC 2007+2012	2007	76,8	-	19	-
YOLOv2	VOC 2007+2012	2007	76,8	34,90 Bn	67	4GB
YOLOv2 544x544	VOC 2007+2012	2007	78,6	59,68	40	4GB
Tiny YOLO	VOC 2007+2012	2007	57,1	6,97	207	1GB

Tabelle 4.1.: Vergleich unterschiedlicher Netzwerke (Detektoren) [20]

Alle Netzwerke aus der Tabelle 4.1 wurden mit dem VOC 2007+2012 Datensatz trainiert. Dieser Datensatz enthält 11530 Bilder mit 20 Klassen. Ein Vergleich der Performance und Geschwindigkeit frames per second (FPS) der Netzwerke aus der Tabelle 4.1 zeigt, dass Tiny YOLO mit Abstand das schnellste Netzwerk ist und lediglich 1GB GPU Memory benötigt. Obwohl das Tiny-YOLO 10mAP ungenauer ist, fällt die Wahl auf dieses Netz. Bei der Realtime-Detektion hat die Geschwindigkeit einen höheren Stellenwert. „mAP“ ist die Metrik zum Messen der Genauigkeit von Objektdetektoren. Es ist der Durchschnitt der maximalen Genauigkeit bei verschiedenen Auswertungen.

Die Architektur des Tiny-YOLO Netzwerk, ist in der Abbildung 3.17 zu sehen. Das Tiny-YOLO ist eine einfachere Version des YOLOv2 Detektors. YOLOv2 besitzt 24 Converloution-Layer, gefolgt von zwei fully connected layers (FC). Das Tiny-YOLO Netzwerk dagegen, lediglich neun Converloution-Layer mit flacheren feature maps. Dadurch wird der Rechenaufwand erheblich reduziert. Für diese Arbeit werden insgesamt sechs verschiedene abgeänderte Tiny-YOLO Netzwerke verwendet. Diese sind wie folgt aufgebaut:

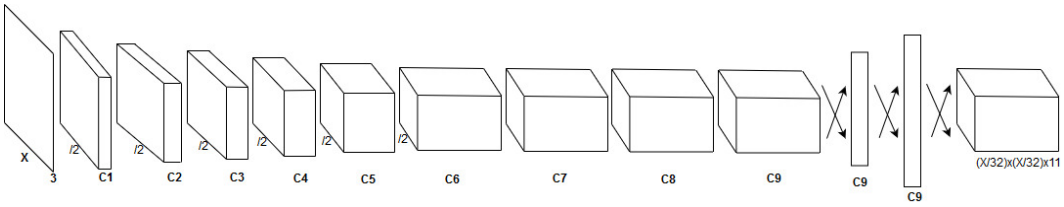

Abbildung 4.2.: Aufbau der zu trainierenden Netzwerke

Die oben gezeigte Abbildung 4.2, stellt den Aufbau der Netzwerke dar. Es werden folgende Werte angepasst:

- X für die Eingangsgröße des Netzwerks (X · X Pixel)
- Cn für die Anzahl an Filter

Die nachfolgende Tabelle 4.2 zeigt die unterschiedlichen zu verwendenden Netzwerke, die in dieser Arbeit verwendet und miteinander verglichen werden. Die Filtergröße beträgt bei allen Convolution Layer 3 · 3 · Cn.

Config	**X**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**
yolov2-tiny-voc_standart	416x416	16	32	64	128	256	512	1024	1024	1024
yolov2-tiny-voc_standart_half_filters	416x416	8	16	32	64	128	256	512	512	512
yolov2-tiny-voc_352x352	352x352	16	32	64	128	256	512	1024	1024	1024
yolov2-tiny-voc_352x352_half_filters	352x352	8	16	32	64	128	256	512	512	512
yolov2-tiny-voc_288x288	288x288	16	32	64	128	256	512	1024	1024	1024
yolov2-tiny-voc_288x288_half_filters	288x288	8	16	32	64	128	256	512	512	512

Tabelle 4.2.: Verwendete Netzwerke

Wie der Tabelle 4.2 zu entnehmen, gibt es sechs unterschiedliche Kombinationen der Netzwerke. Es gibt drei Eingangsgrößen, mit je zwei unterschiedlichen Anzahlen an Filtern bei den Convolution Layern. Einmal die Standardgröße und sowie die halbierte Filteranzahl in jedem Convolution Layer. Die Idee dabei ist, je kleiner die Eingangsgröße und je weniger Filter benötigt werden, desto weniger Berechnungen werden benötigt und somit wird die Geschwindigkeit des Netzwerks erhöht. Diese Einstellungen werden in der Config-Datei der einzelnen Netzwerke eingestellt. Den Aufbau einer solchen Datei, ist in dem nachfolgenden Listing 4.3 gezeigt. Es zeigt Ausschnitte aus der yolov2-tiny-voc_standart Config von der Tabelle 4.2.

```
1 [net]
2 # Testing
3 #batch=1
4 #subdivisions=1
5 # Training
6 batch=64
```

```
7  subdivisions=8
8  width=416
9  height=416
10 channels=3
11 momentum=0.9
12 decay=0.0005
13 angle=0
14 saturation = 1.5
15 exposure = 1.5
16 hue=.1
17
18 learning_rate=0.001
19 max_batches = 40200
20 policy=steps
21 steps=-1,100,20000,30000
22 #steps=-1,100,20000,30000
23 scales=.1,10,.1,.1
24
25 [convolutional]
26 batch_normalize=1
27 filters=16
28 size=3
29 stride=1
30 pad=1
31 activation=leaky
32
33 [maxpool]
34 size=2
35 stride=2
36
37 [convolutional]
38 batch_normalize=1
39 filters=32
40 size=3
41 stride=1
42 pad=1
43 activation=leaky
44
45 [maxpool]
46 size=2
47 stride=2
```

Listing 4.3: Aufbau yolov2-tiny-voc_standart

In den Zeilen acht und neun aus dem Listing 4.3 wird die Eingangsgröße X definiert. Dieser Wert muss durch 32 teilbar sein, wie bereits in den Grundlagen 3.3.4 erwähnt, da die Größe der Ausgangsmatrix abhängig der Eingangsgröße ist. Die Anzahl an Filtern wird z.B. in den Zeilen 27 (C1) und 39 (C2) definiert.

4.2.1. Unterschied SSD vs 2-Stage Detecor

Wie bereits im Kapitel 3.3.4 erwähnt, gehört YOLO zu den Single Shot Detectors kurz SSD. Diese Variante ist auf Grund der niedrigeren Anzahlen an Rechendurchläufen schneller. Dennoch bringt diese Variante negative Aspekte mit sich. Die Erkennungsraten sind bei den SSDs etwas schlechter. Alternative Netze für YOLO, die ebenfalls auf das Prinzip von SSD basieren, sind folgende:

- SSD [Google]
- DSSD [Amazon]
- RetinaNet [Facebook]

Eine andere Möglichkeit für die Aufgabe der Detektion sind so genante 2-Stage Detektoren. Diese bestehen aus zwei Schritten:

- Algorithmus zum Regionen finden
- Convolutional Neural Network zum Klassifizieren der Region

Anders als z.B. beim YOLO kann der Algorithmus zur Bestimmung des Ortes auch ein einfaches neuronales Netz sein. Für jede mögliche Detektion wird anschließend eine extra Klassifizierung durchgeführt. Siehe Abbildung 4.3. Dementsprechend ist die Geschwindigkeit des Netzwerks schlechter. Die Genauigkeit hingegen besser.

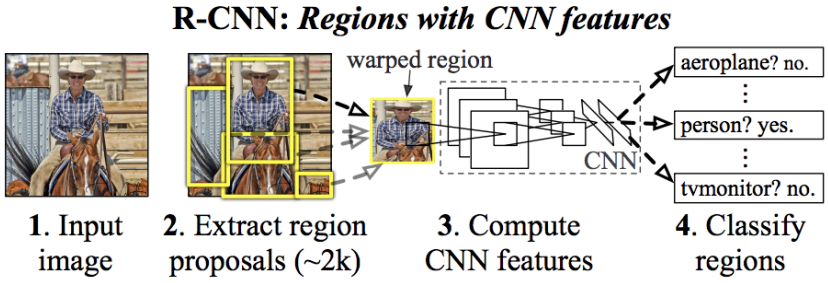

Abbildung 4.3.: Ablauf eines R-CNN [8]

Beispiele für 2-Stage Detektoren sind folgende:

- R-CNN, Fast R-CNN, Faster R-CNN, Mask R-CNN [Facebook]
- R-FCN [Microsoft Research]

4.3. Erstellen der Trainings- und Testdaten

In diesem Abschnitt der Arbeit geht um den Aufbau, Beschaffung und der Vorbereitung der Trainings und Testdaten. Ein guter Datensatz ist essenziell für eine gute Erkennungsrate. Je mehr unterschiedliche Bilder zum Training verwendet werden, desto besser wird die Detektion. Der verwendete Datensatz für das Training in dieser Arbeit ist verhältnismäßig klein. Dieser besteht aus circa 600 unterschiedlichen Bildern. Diese sind aufgeteilt in ungefähr 540 Bilder zum trainieren und 60 Bilder zum testen. Zum Trainieren eines YOLOv2 Netzwerkes werden bestimmte Dateien benötigt, damit das Framework weiß, was trainiert werden muss. Diese benötigten Dateien sind folgende:

- cfg/obj_ship.data
- cfg/obj_ship.names
- Und die Config-Datei von dem zu trainierenden Netz

Die „obj_ship.data“ sowie die „cfg/obj_ship.names“ Dateien, sind für alle sechs zu trainierenden Netze gleich. Lediglich die Config-Datei wird angepasst. Zuerst wird die „obj_ship.data“ Datei betrachtet, welche nachfolgend im Listing 4.4 dargestelt ist.

```
1 classes= 1
2 train  = /home/torben/darknet/cfg/train.txt
3 valid  = /home/torben/darknet/cfg/test.txt
4 names = /home/torben/darknet/cfg/obj_ship.names
5 backup = backup/
```

Listing 4.4: Aufbau obj_ship.data

In der ersten Zeile der Datei 4.4 wird die Anzahl der zu detektierenden Klassen angegeben. In diesem Fall sollen nur Schiffe erkannt werden. Somit muss dort eine eins stehen. Zeile zwei und drei geben den Pfad zu der Trainings- bzw. Testdatei an. In diesen Dateien sind die Pfade der einzelnen Trainings und Testbilder enthalten. In der Zeile vier steht der Pfad zu der zweiten Datei, der „obj_ship.names“. In dieser sind die Namen der Klassen enthalten. In diesem Fall nur eine und zwar ship. Dies ist im nachgehenden Listing 4.5 zu sehen.

```
1 ship
```

Listing 4.5: Aufbau obj_ship.names

Ausschnitte aus einer Config-Datei wurde bereits im Abschnitt 4.2 erläutert und werden später noch im Abschnitt 4.4 fortgeführt.

4.3.1. Beschaffung der Daten

Zunächst werden die Bilder zum Trainieren und Testen benötigt. Es gibt einige bekannte Datenbanken wie z.B. ImageNet, in der bereits eine große Menge solcher Bilder vorhanden sind. In dieser Arbeit werden die Bilder jedoch aus der Google Bildersuche bezogen. Dafür wird das Tool "google-images-download"verwendet. Dieses Tool kann einfach über das Python Paketverwaltungsprogramm PIP mit „pip install google_images_download“ installiert werden. Nun kann einfach mit dem Befehl „googleimagesdownload --keywords "Schiffe, boats, ship, ships"--limit "100"“eine Bildersuche gestartet werden. Das Tool sucht daraufhin in Google nach den Keywords und speichert die gefundenen Bilder in einzelnen Verzeichnissen, die nach den Keyboards benannt sind, unter „downloads“ in dem ausgeführten Verzeichnis ab.

4.3.2. Aufbereitung der Daten

Als nächstes müssen diese Bilder aufbereitet werden. Es gibt einige doppelte und ungeeignete Bilder. Ungeeignete Bilder sind z.B. Bilder welche gar kein Schiff zeigen, selbstgemalt, viel zu klein bzw. unscharf sind oder Bilder die komplett von einer Schrift bedeckt sind. Beispielhaft für ein geeignetes und zwei ungeeignete Bilder zeigt die nachstehenden Abbildung 4.4.

Abbildung 4.4.: Beispiele für geeignete und ungeeignete Trainingsbilder

4.3.2.1. Labeln

Als nächstes müssen diese Bilder gelabelt werden, damit das Framework später weiß, wo genau in dem Bild sich welches Objekt befindet und welcher Teil von dem Bild nur Hintergrund darstellt. Dafür wird für jedes einzelne Bild eine Textdatei erstellt, in der alle Informationen vorhanden sind. Das verwendete Format ist ein spezielles YOLO-Format und ist wie folgt in Listing 4.6 dargestellt.

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

Listing 4.6: YOLO annoation Format

Das YOLO-Format ist simpel aufgebaut. Als erstes steht die Nummer der Objekt Klasse mit null beginnend. Da bei dem Beispiel nur Schiffe erkannt werden, gibt es nur eine Klasse. Diese ist somit immer null. X und Y sind die Mittelpunktkoordinaten des Objekts. Die letzten beiden Werte width und height geben die Breite in X und Y Richtung an. Diese sind auf eins normiert. Falls mehrere Objekte in einem Bild sind, werden diese untereinander in gleicher Art geschrieben. In der Folgenden Abbildung 4.5 ist ein gelabeltes Bild mit der dazugehörigen Datei im Listing 4.7 zu sehen.

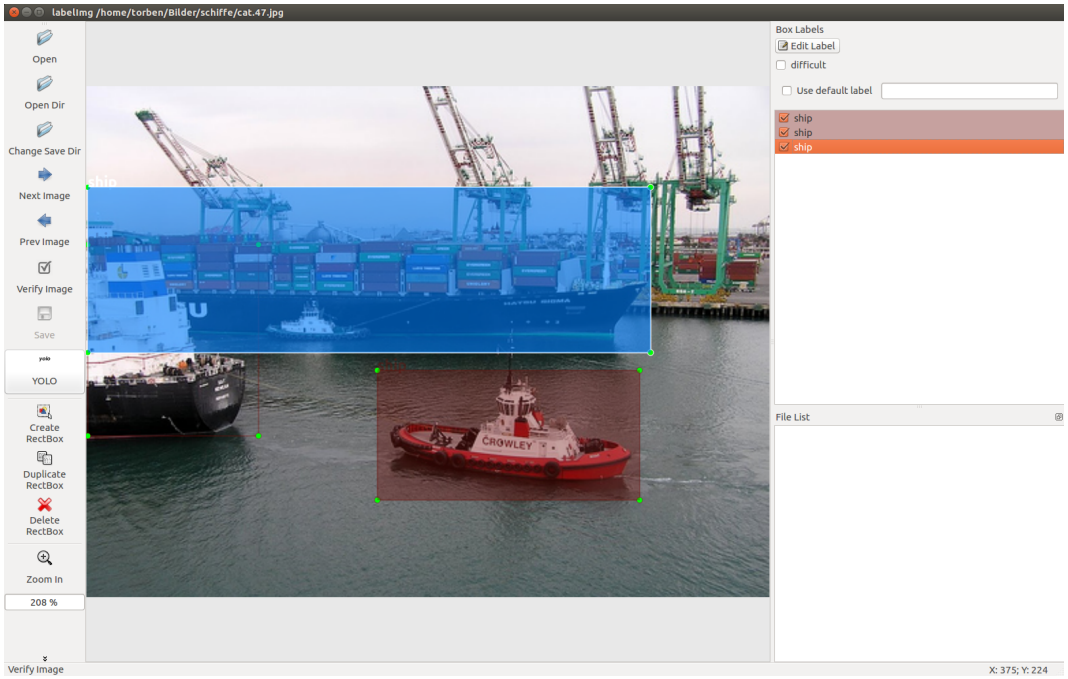

Abbildung 4.5.: LableImg

```
0 0.618000 0.684492 0.384000 0.256684
0 0.127000 0.498663 0.250000 0.377005
0 0.414000 0.362299 0.824000 0.323529
```

Listing 4.7: YOLO Labeldatei passend zur Abbildung 4.5

Als Hilfe für das Labeln wurde das Python Tool labelImg von Tzutalin verwendet. Das Tool ist ebenso in der Abbildung 4.5 zu sehen. Bei dem Tool werden die Objekte markiert und die dazugehörige Textdatei automatisch mit dem richtigen Format erstellt. Dies vereinfacht und beschleunigt das Labeln, sowie das Erstellen der Dateien erheblich.

4.3.2.2. Einordnen in Traings- / Testdaten

Wie bereits im Teil 4.3 beschrieben werden die Bilder in Trainings- und Testbilder eingeteilt. Der Faktor beträgt zirka zehn zu eins. Auf zehn Trainingsbilder kommt ein Testbild. Der Aufbau der train.txt und test.txt ist in dem nachfolgenden Listing 4.8 dargestellt.

```
/home/torben/Downloads/obj/cat.59.jpg
/home/torben/Downloads/obj/4. 4681986_20140410083142702_1_xlarge.jpg
/home/torben/Downloads/obj/52. w29coho_0228.jpg
/home/torben/Downloads/obj/10. saver-620-002-schuetze-boote-berlin.jpg
/home/torben/Downloads/obj/cat.11.jpg
/home/torben/Downloads/obj/71. superior-cruise-ships.jpg
/home/torben/Downloads/obj/cat.3.jpg
```

Listing 4.8: Aufbau train.txt und test.txt

Diese Listen wurden mit der Hilfe des folgenden Python-Skripts 4.9 erstellt.

```
import glob, os

# Aktuelles Verzeichnis
current_dir = os.path.dirname(os.path.abspath(__file__))

# Pfad zu den Bildern
path_data = '/home/torben/Downloads/obj/'

# Anzahl an Trianingsbilder fÃ¼r 1 Testbild
percentage_test = 10;

# Erstelle train.txt und test.txt
train_data = open('train.txt', 'w')
test_data = open('test.txt', 'w')

# Alle .jpg Bilder aus dem Ordner in die Listen train.txt und test.txt eintragen
```

```
17 counter = 1
18 index_test = round(100 / percentage_test)
19 for pathAndFilename in glob.iglob(os.path.join(current_dir, "*.jpg")):
20     title, ext = os.path.splitext(os.path.basename(pathAndFilename))
21
22  if counter == index_test:
23         counter = 1
24         test_data.write(path_data + title + '.jpg' + "\n")
25     else:
26         train_data.write(path_data + title + '.jpg' + "\n")
27         counter = counter + 1
```

Listing 4.9: Python-Skript train.txt und test.txt erstellen

Dieses Skript sortiert alle Bilder, welche im angegebenen Pfad in Zeile sieben enthalten sind automatisch in die Benötigten Listen. Dies geschieht mit dem vordefinierten Faktor von zehn zu eins.

4.4. Trainieren der Netzwerke

In diesem Abschnitt geht es um das Trainieren der unterschiedlichen Netze. Zunächst müssen einige Werte in der Config-Datei 4.3 der Netze für das Training angepasst werden. Dafür wird in Zeile 6 batch auf 64 und in Zeile 7 subdivisons auf 8 gesetzt. Die Variable batch gibt an, wie viele Bilder für jeden Trainingsschritt (Iteration) verwendet werden sollen. Der Punkt subdivison gibt den Teiler für die batch Größe an. Somit wird der batch von 64 Bildern in 8 kleine batches aufgeteilt $64/8$. Daraus ergeben sich acht kleine batches zu je acht Bilder pro Iteration, diese werden der GPU zur Verarbeitung übergeben. Sind diese 8 bachtes abgearbeitet, beginnt eine neue Iteration. Steht eine starke GPU mit viel VRAM zur Verfügung, kann die batch size erhöht oder die subdivsion reduziert und somit effektiver und schneller trainiert werden. Später, zum Testen der Netzwerke sollten, diese Werte auf eins gesetzt werden. Ein weiterer wichtiger Punkt ist die learning_rate, welche bereits im Abschnitt 3.3.3.1 erläutertet wurde. Diese ändert sich durch folgende Parameter im laufe des Trainings:

- policy=steps: Bedeutet, dass die nachstehenden steps und scales Parameter die learning rate anpasst im laufe des Trainings.
- steps=1,100,20000,30000: Gibt an, nach wie vielen Iterationen die learning rate angepasst wird.
- scales=.1,10,.1,.1: Mit diesen Werten wird die rate nach den oben gewählten steps angepasst.

Max_batches gibt an, wie viele Iterationen maximal trainiert werden sollen. Als nächstes werden die einzelnen Netzwerke trainiert. Der Trainingsprozess wird wie folgt gestartet:

```
torben@torben-OptiPlex-7010:~/darknet$ ./darknet detector train /path/datafile.data
/path/yolo.cfg darknet19_448.conv.23 > /path/trainoutput.txt
```

Abbildung 4.6.: Traning der Netze

Bei dem Trainingsbefehl von 4.6 werden mit darknet19_448.conv.23 vortrainierte Gewichte übergeben. Dies soll laut den Entwicklern [20] ein schnelleres Erreichen des Trainingsziels bewirken. Damit ist das Trainieren des Netzes gestartet. Der Trainingsfortschritt wird in die Textdatei weitergeleitet und nach Fertigstellung des Trainings analysiert. Nachstehend im Listing 4.10 ist ein Ausschnitt der Trainingsdatei zu sehen.

```
40199: 0.420257, 0.345440 avg, 0.001000 rate, 1.004303 seconds, 2572736 images
40200: 0.246859, 0.335582 avg, 0.001000 rate, 1.029924 seconds, 2572800 images
```

Listing 4.10: Training output

Der Ausschnitt 4.10 zeigt nur den wichtigen Teil der letzten beiden Iterationen einer Trainingsdatei. Zum Auswerten des Trainings sind folgende drei Werte wichtig [26, vgl.]:

- 40199 und 40200 geben den Iterationsschritt an.
- 0.420257 und 0.236859 zeigen den totalen Fehler, auch total loss genannt an.
- 0.345440 und 0.335582 Zeigen den Durchschnitt des total loss an. Dieser ist der ausschlaggebende Wert beim Training und sollte möglichst gering sein. Je geringer dieser ausfällt, desto höher ist die Erkennungsrate des Netzes. Ideal wäre ein Wert von unter 0.06.

Die loss Funktion zum berechnen des Fehlers ist in der Formel 4.1 dargestellt.

$$loss = loss_{localization} + loss_{confidence} + loss_{classification} \tag{4.1}$$

Die Formel 4.1 kann in drei einzelne Abschnitte unterteilt werden. Diese sind in der Abbildung 4.7 ersichtlich und sind wie folgt unterteilt:

- classification loss (rot) zum Bestimmen des Fehlers bei der Klassifikation.
- localization loss (grün) zum Bestimmen des Fehlers bei der Ortsbestimmung und Größe der Boundin Boxes.
- confidence loss (blau) zum Bestimmen des Fehlers bei der Abschätzung des Modells.

Der gesamte Loss ist somit die Summer aller einzelnen Funktionen, die zur Bestimmung des Fehlers dienen [3, vgl.].

$$\mathbf{loss}_{\text{localization}} = \lambda_{\mathbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

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

$$\mathbf{loss}_{\text{confidence}} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

$$\mathbf{loss}_{\text{classification}} = \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2$$

Abbildung 4.7.: loss Trainingsfunktion [22]

Genauer wird in dieser Arbeit nicht auf diese Funktion eingegangen, da dies den Rahmen dieser Arbeit übersteigen würde.

In den folgenden Abbildungen sind die verschiedenen Trainingsergebnisse grafisch dargestellt. Dafür werden die wichtigen Werte aus den Trainingsdateien extrahiert, in eine CSV-Datei übertragen und anschließend geplottet. Dies wird mit den Python-Skripten sliceandcreatplotdata.py und PlotV2.py aus dem Anhang realisiert.

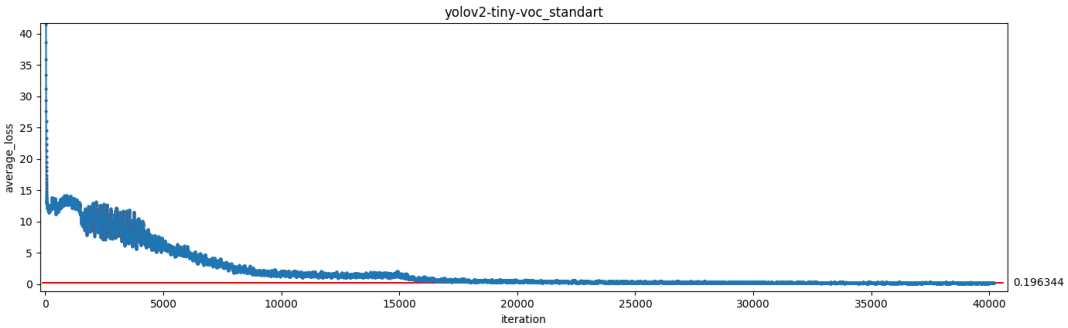

Abbildung 4.8.: yolov2-tiny-voc_standart Trainingskurve

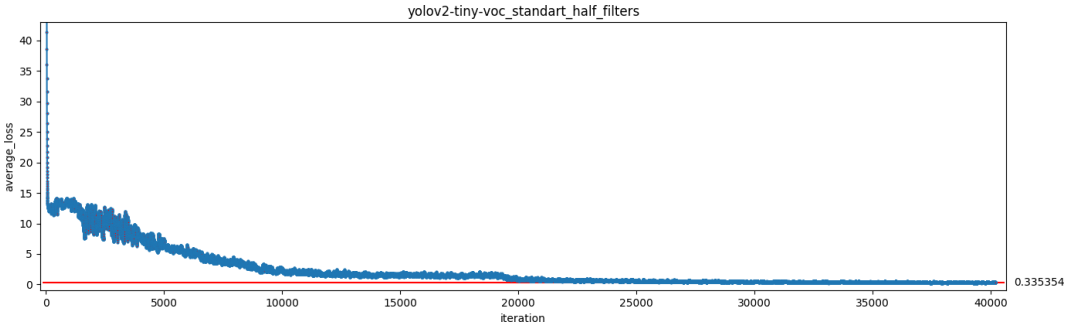

Abbildung 4.9.: yolov2-tiny-voc_standart_half_filters Trainingskurve

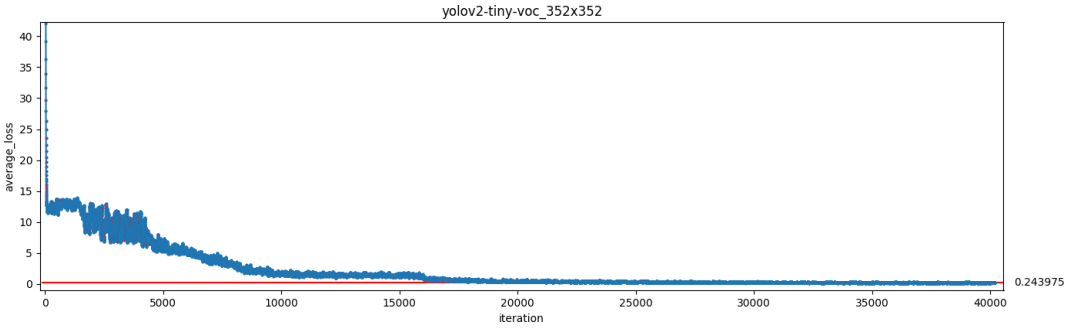

Abbildung 4.10.: yolov2-tiny-voc_352x352 Trainingskurve

![h]

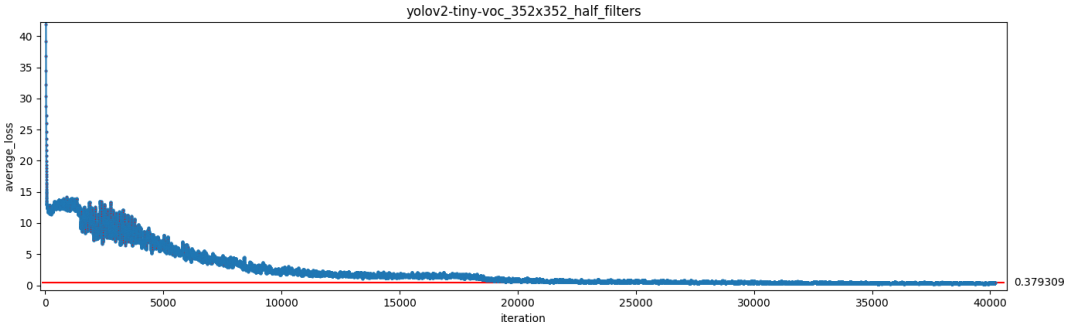

Abbildung 4.11.: yolov2-tiny-voc_352x352_half_filters Trainingskurve

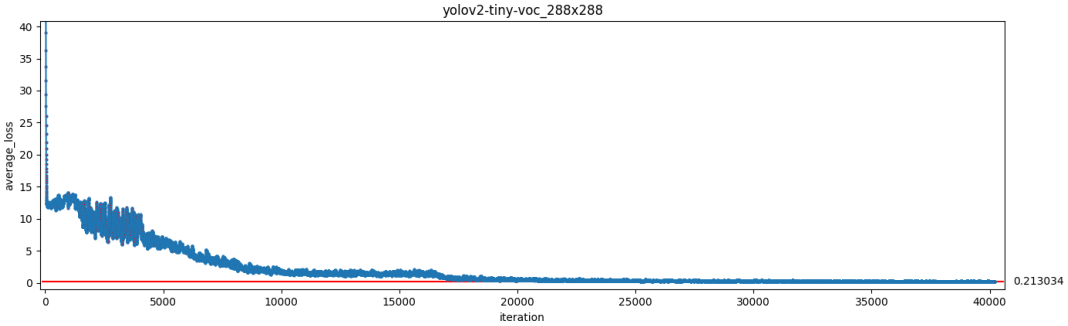

Abbildung 4.12.: yolov2-tiny-voc_288x288 Trainingskurve

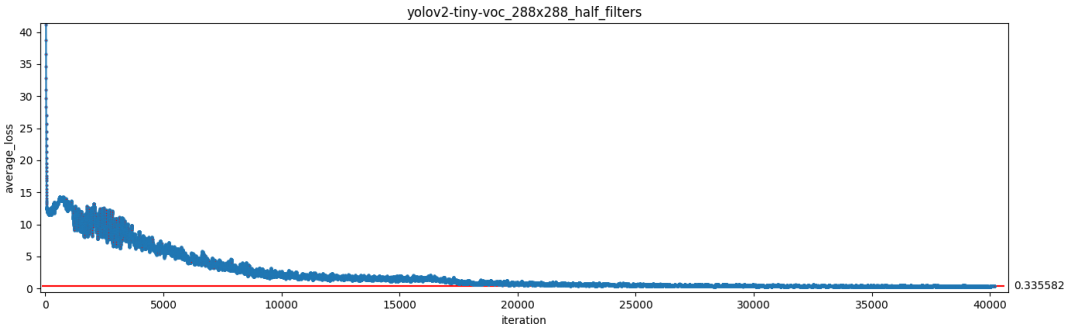

Abbildung 4.13.: yolov2-tiny-voc_288x288_half_filters Trainingskurve

In den oberen Abbildungen ist der durchschnittliche Fehler im Verlauf der Iterationen dargestellt. Die rote Linie stellt den letzten Wert des Trainings dar. An Hand der Verläufe ist zu sehen, dass das Training bereits ab 20000 Iterationen keine signifikante Besserung mehr bringt. Außerdem fällt auf, dass die Netze mit halbierter Filteranzahl länger brauchen um

auf ihren Ausgangswert zu kommen als die anderen. In der folgenden Tabelle 4.3 sind die einzelnen Endwerte des Trainings noch einmal zusammen gefasst.

Config	Eingangsgöße[X]	loss
yolov2-tiny-voc_standard	416x416	0,196344
yolov2-tiny-voc_standard_half_filters	416x416	0,335354
yolov2-tiny-voc_352x352	352x352	0,243975
yolov2-tiny-voc_352x352_half_filters	352x352	0,379309
yolov2-tiny-voc_288x288	288x288	0,213034
yolov2-tiny-voc_288x288_half_filters	288x288	0,335582

Tabelle 4.3.: Übersicht Trainingsergebnis

Ein Blick in die Tabelle 4.3 zeigt auf, dass das Standart-Netz mit der größten Auflösung die niedrigste Fehlerrate mit ca. 0,196 besitzt. Auffällig ist, das Netz mit der Auflösung von 288x288. Dies besitzt die zweit beste Fehlerrate mit 0,213, nicht wie erwartet das Netz mit der zweit größten Auflösung. Dies kann auf Grund von einer Überanpassung durch zu langes Ausführen eines iterativen Trainingsalgorithmus entstehen. Beziehungsweise durch für das Problem oder für den verfügbaren Datenumfang zu komplexen Netzen [24, vgl.]. Ein Beispiel dafür ist in der nachstehenden Abbildung 4.14 zu entnehmen.

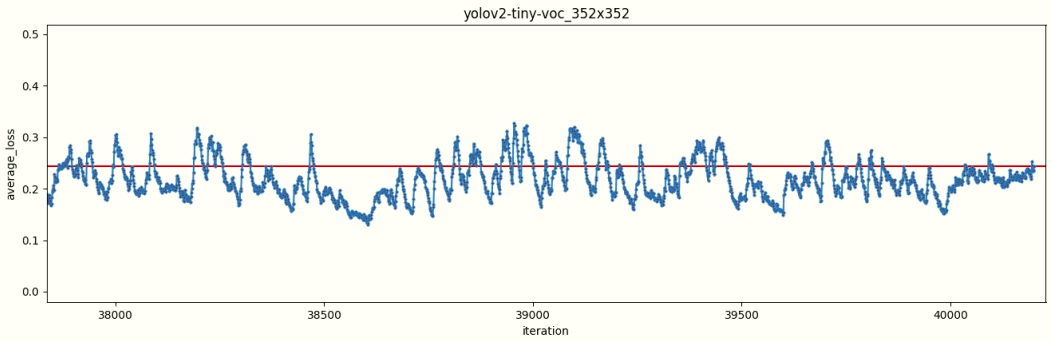

Abbildung 4.14.: yolov2-tiny-voc_352x352 Trainingskurve gezoomt

Es ist sehr gut zuerkennen, dass bereits schon vorher das Netz mit der Auflösung von 352x352 einen erheblich besseren loss besitzt. Keines der hier gezeigten Netze hat den gewünschten Wert von 0,06 unterschritten. Dies kann unterschiedliche Gründe haben. Zum einen ist der Datensatz mit insgesamt 600 Bildern recht klein. Ein anderer Grund können die unterschiedlichen verwendeten Typen von Schiffen sein. Für das Training wurden unterschiedliche Arten von Schiffen verwendet unter, anderem Containerschiffe, Segelschiffe und Sportboote, welche teils sehr unterschiedliche Formen aufweisen. Dies hat vermutlich einen

negativen Einfluss auf das Netz. Eine mögliche Lösung wären unterschiedliche Klassen für die einzelnen Typen von Schiffen oder eine Baumstruktur. Dies hat den Vorteil, dass es eine Oberklasse gibt und mehrere Unterklassen. Falls ein Schiff nicht genau klassifizierbar wäre, aber visuelle Gemeinsamkeiten mit einem Schiff aufweist, wird für dieses Objekt die nächste Überklasse verwendet. Der Einfachheit halber wurden für diese Arbeit alle Netze bis zum Schluss (Iteration 40200) trainiert und verwendet.

5. Verifikation der neuronalen Netzwerke und Auswertung der Ergebnisse

In diesem Kapitel werden die unterschiedlich trainierten Netzwerke auf ihre Geschwindigkeit und Performance getestet.

5.1. Test und Implementierung auf dem PC und Jetson TK1

Als erstes wird die Funktion der Netzwerke mit Hilfe des Frameworks getestet. Darkent liefert bereits eine fertige Lösung zum Testen und kann mit dem Befehl aus der Abbildung 5.1 gestartet werden.

```
torben@torben-OptiPlex-7010:~/darknet$ ./darknet detector demo cfg/obj_ship.data
 cfg/yolov2-tiny-voc_352x352.cfg backup/yolov2-tiny-voc_352x352_final.weights  /
home/torben/Downloads/smaler360p.mp4 -thresh 0.20
```

Abbildung 5.1.: Befehl zum Testen der Netzwerke

Für den ersten Test, wird ein Video als Eingangsquelle verwendet. Dieses Video besitzt eine Auflösung von 360p, 25FPS und liegt in dem Format Moving Pictures Experts Group kurz MPEG IV vor. Dasselbe Video wird zusätzlich später bei den Geschwindigkeitsmessungen in dieser Arbeit verwendet.

In der hierauf folgenden Abbildung 5.2 ist ein Ausschnitt aus dem im Abbildung 5.1 gestarteten Test auf dem Trainingscomputer mit der NVIDIA GeForce 970 zusehen. Der hier ausgewählte thresh von 0.20 gibt an, ab wie viel Prozent das Schiff erkannt werden soll.

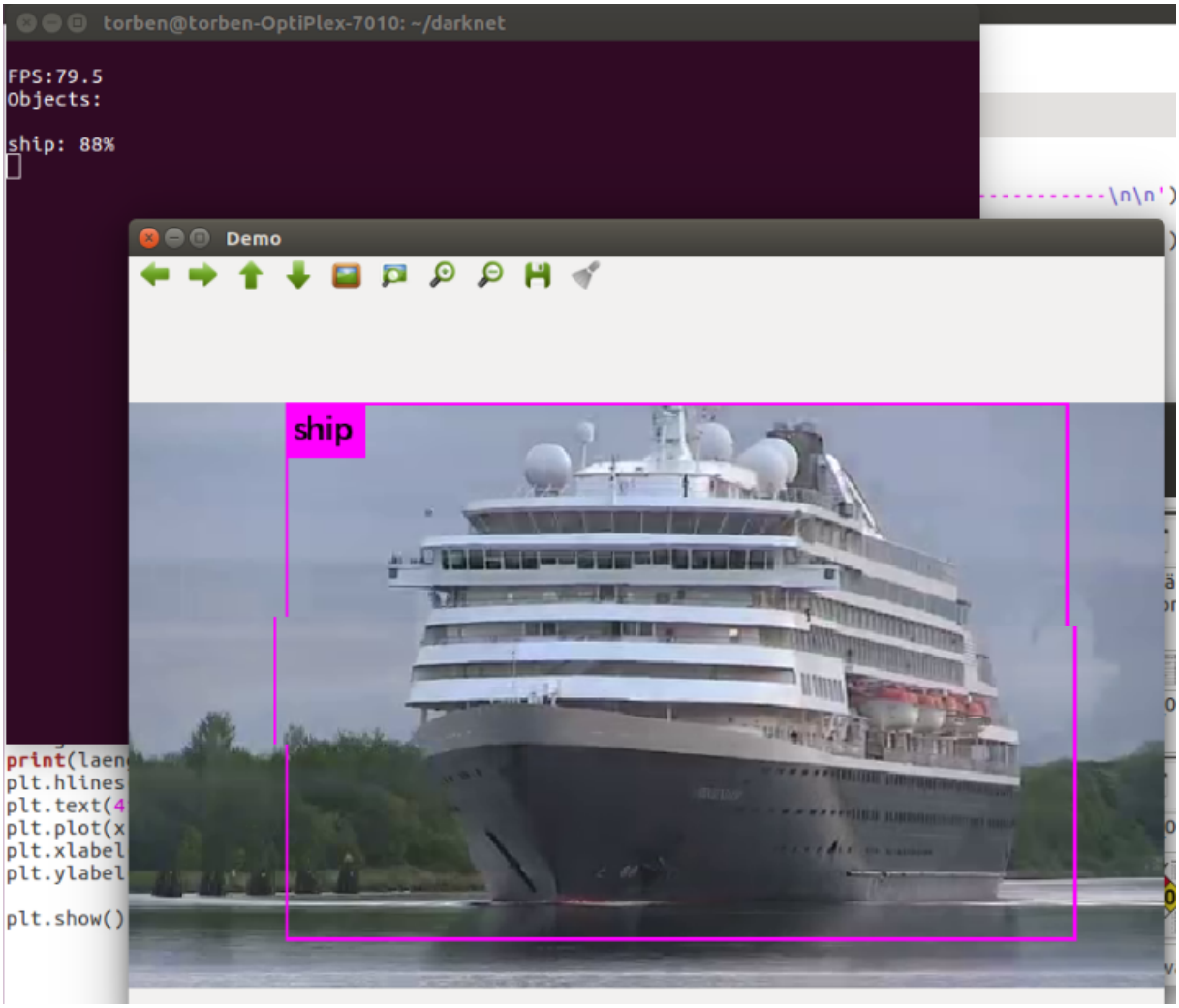

Abbildung 5.2.: Darknet Test Auflösung 352x352 mit der NVIDIA GeForce 970

Wie in der Abbildung 5.2 zu erkennen, wird das Schiff korrekt mit einer Wahrscheinlichkeit von 88% detektiert. Die Bilder pro Sekunde, kurz FPS variieren zwischen 70 FPS und über 100 FPS. Dies deutet auf eine sehr gute Echtzeitfähigkeit des Systems hin. Eine handelsübliche Webcam gibt in der Regel 30 bzw. 60 FPS wieder und kann somit ohne Probleme vom Netz verarbeitet werden. Als nächstes wird der gleiche Test auf dem Jetson TK1 Entwicklerboard durchgeführt. Es stellt sich heraus, dass es Probleme bei der Portierung der Netze von einem 64-Bit System auf ein 32-Bit System gibt und keine Objekte erkannt werden. Damit eine erfolgreiche Detektion auf dem Jetson TK1 möglich ist, muss zunächst die Datei „src/parser.c“ angepasst werden. Alle sizeof(size_t) müssen mit sizeof(size_t)*2 ersetzt und anschließend neu compiliert werden. Dies ist notwendig, da bei dem 64-Bit System 8 By-

tes als Ergebnis rauskommen und bei einem 32-Bit System nur 4 Bytes. Das Ergebnis der Detektion ist in der folgenden Abbildung 5.3 zu sehen.

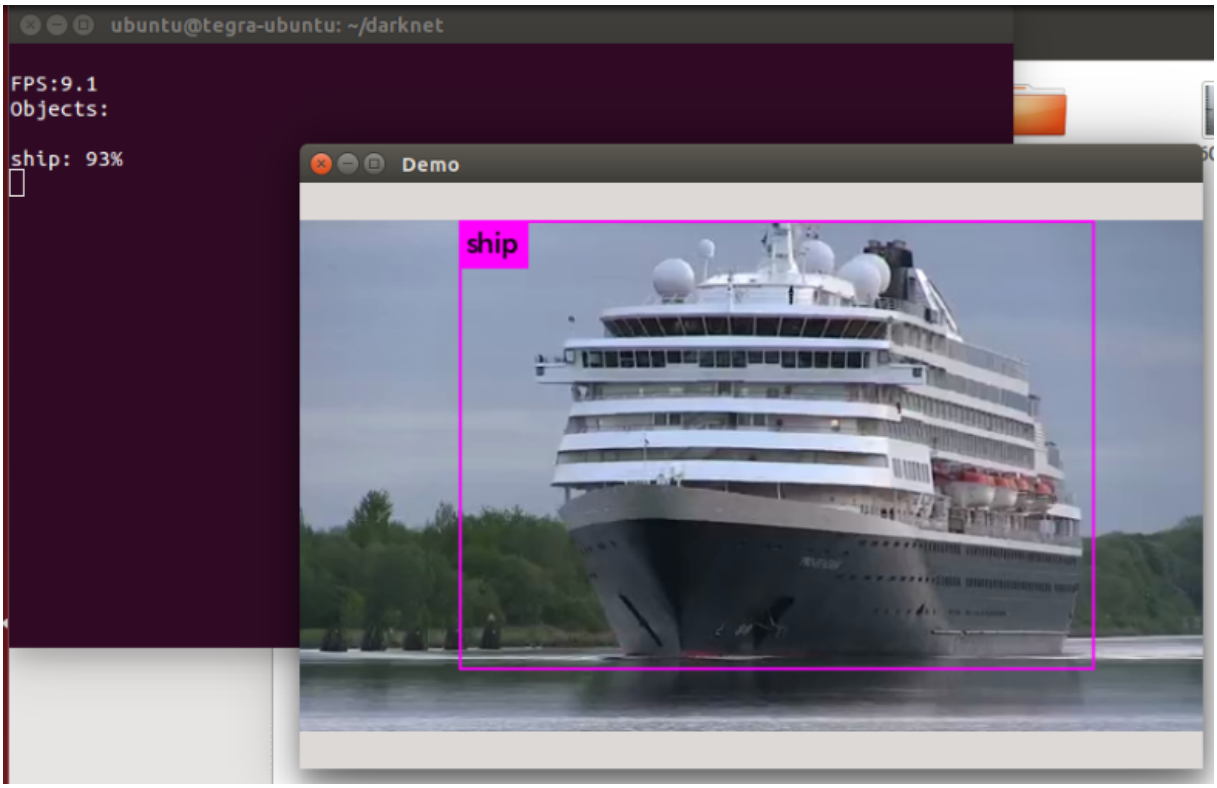

Abbildung 5.3.: Darknet Test Auflösung 352x352 mit dem Jetson Tk1

Wie in der Abbildung 5.3 zu entnehmen, wird wie zuvor das Schiff erfolgreich mit einer Wahrscheinlichkeit von 93% detektiert. Die geringe Abweichung ist mit dem nicht 100 prozentig übereinstimmenden Zeitpunkt bei der Aufnahme zu erklären. Auffällig ist jedoch, dass die FPS bei dem Jetson TK1 zwischen 8 und 10 FPS liegen. Werden die Mittelwerte bei diesem Test miteinander verglichen, ergibt sich ein Faktor von $80FPS/9FPS = 8,89$. Wird nun die Anzahl der vorhanden CUDA-Cores miteinander verglichen, ist auch dort ein sehr ähnlicher Faktor zu erkennen $1664CORES/192CORES = 8,7$. Daraus lässt sich die Vermutung bestätigen, dass die Anzahl der CUDA-Cores signifikant für die Geschwindigkeit sind. Es stellt sich heraus, dass die hier vorgefertigte Lösung, welche vom Darknet Framework direkt zur Verfügung gestellt wird, nicht unbedingt für eine Real-Time Detektion geeignet ist. Das Programm arbeitet jeden Frame sequentiell ab. Solange die Quelle weniger FPS aufweist als das Netz wiedergeben kann, ist eine echtzeitfähige Detektion mit diesem Programm möglich. Ist jedoch, wie beim Jetson TK1, die Quelle (25 FPS) schneller als die Verarbeitung und Wiedergabe (9 FPS), wird die Quelle ausgebremst und dementsprechend langsamer

wiedergegeben. Der Buffer, in dem die Frames zwischen gespeichert werden, wächst somit immer weiter an und es droht die Gefahr eines Überlaufens des Buffers. Um dieses Problem zu lösen und eine Echtzeitfähigkeit auf den Jetson TK1 zu erreichen, wird ein eigenes Programm entwickelt. Dies wird in dem nachfolgenden Kapitel 5.1.1 genauer erläutert.

5.1.1. Programmablauf

Da eine sequenzielle Verarbeitung jedes einzelnen Frames auf dem Jetson TK1 nicht in Echtzeit möglich ist, wird ein extra dafür entwickeltes Programm verwendet. Damit das möglich ist, wird die Multithreading Fähigkeit des Systems genutzt. Das Programm besteht aus zwei Threads. Der erste Thread übernimmt die Verarbeitung der Bildquelle, während der zweite Thread sich aus dem ersten Thread den aktuellen Frame holt und die Detektion durchführt. Der Vorteil besteht darin, dass die Detektion auch bei langsamen Netzen noch echtzeitfähig ist. Der Nachteil bei dieser Variante ist, dass einige Frames verloren gehen. Dies hat jedoch bei langsamen Objekten wie Schiffen keinen erheblichen Einfluss, da sie eine längere Zeit im Bild sind. Den Ablauf des oben erklärten Programms kann der nachstehenden Abbildung 5.4 entnommen werden.

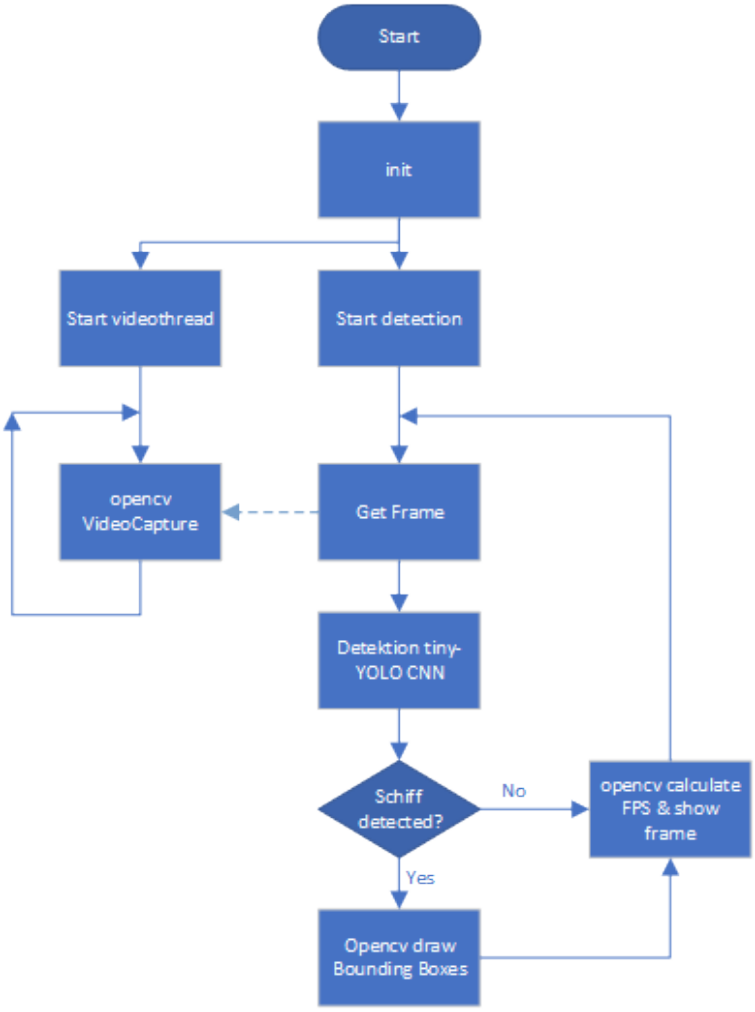

Abbildung 5.4.: Flussdiagramm threaddetection.py

Ein weiterer Test dieser Arbeit ist die Untersuchung der Performance von den unterschiedlichen Netzen. Um dies zu testen, wird eine x beliebige Anzahl an bekannten Testbildern in einen Ordner gelegt. Das Programm arbeitet diese Bilder sequenziell ab und untersucht diese nach den gesuchten Objekten. Werden Objekte erkannt, wird dies gelabelt und in einen extra Order abgespeichert. Dieser Programmablauf ist der nächsten Abbildung 5.5 zu entnehmen.

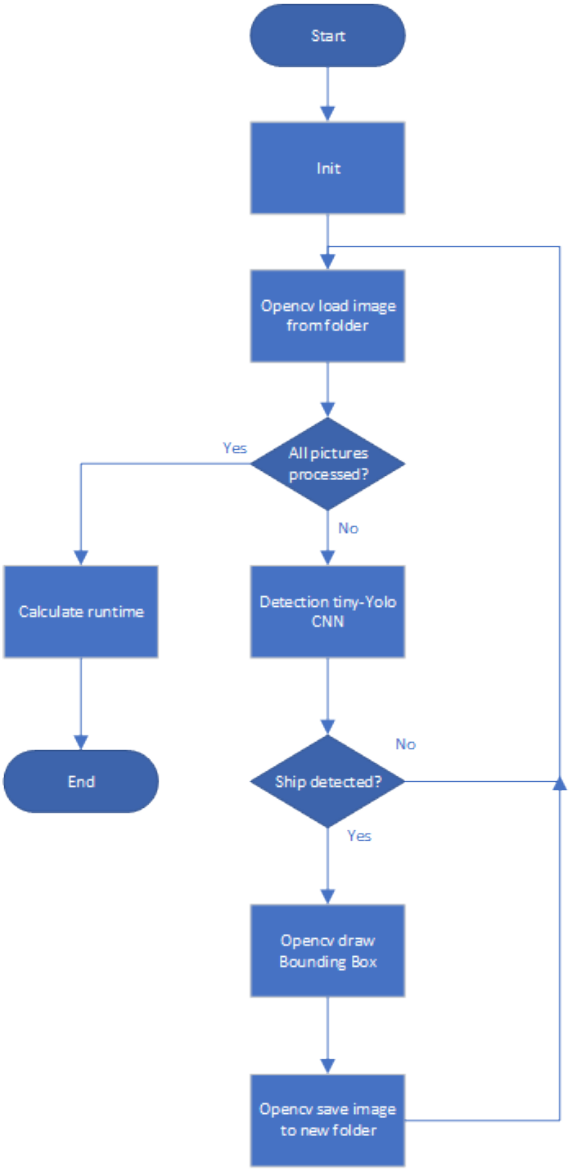

Abbildung 5.5.: Flussdiagramm processpics.py

Das Programm endet, sobald alle vorhandenen Bilder aus dem Ordner abgearbeitet sind. Das Programm befindet sich auf der CD, die im Anhang zu finden ist.

5.2. Testen der Unterschiedlichen Netze

In diesem Kapitel geht um das Testen der unterschiedlichen Netze auf dem Jetson TK1. Als Tests werden die im zuvor erklärten Abschnitt 5.1.1 verwendeten Python Skripte verwendet. Die unterschiedlichen Skripte sind im Anhang zu finden.

5.2.1. Geschwindigkeit

Im ersten Test werden mit dem Python Skript threaddetection.py 5.4 die Geschwindigkeit und Echtzeitfähigkeit der verschiedenen Netze getestet. Dafür wird einmal das bereits im 5.1 verwendete Video mit der Auflösung von 360p und 25 FPS verwendet. Als zweites wird eine Logitec C170 USB-Webcam mit einer Auflösung von 640x480 getestet und zu guter Letzt wird als Quelle eine IP Kamera verwendet. Als IP Kamera dient ein Samsung Galaxy S8 mit der kostenlosen APP IP WEBCAM. Dort werden die Netze mit vier unterschiedlichen Auflösungen getestet. Diese sind: 1024x768, 720x480, 640x480 und 352x288 Pixel. Als Basis vom Python Skript dient das Beispiel Skript darknet.py von Darknet. Dieses wurde unter anderem mit multithreading erweitert.

In der nachstehenden Tabelle 5.1, sind die Ergebnisse der Tests dargestellt.

Netz	**Video 360p**	**IP Webcam 352x288**	**IP Webcam 640x480**	**IP Webcam 720x480**	**IP Webcam 1024x768**	**USB Webcam 640x480**
yolov2-tiny-voc_standart	2,23	1,76	1,21	1,43	1,21	1,37
yolov2-tiny-voc_standart_half_filters	2,40	1,85	1,82	1,78	1,49	1,78
yolov2-tiny-voc_352x352	3,46	2,77	1,84	2,07	1,25	2,21
yolov2-tiny-voc_352x352_half_filters	3,22	2,92	2,38	2,41	1,80	2,27
yolov2-tiny-voc_288x288	4,69	3,21	2,38	2,26	1,47	2,35
yolov2-tiny-voc_288x288_half_filters	4,48	4,23	3,20	3,35	2,11	3,03

Tabelle 5.1.: "Übersicht Geschwindigkeitstests der Netze

In der Tabelle 5.1 ist gut zu erkennen, dass die Auflösung eine gewisse Auswirkung auf die Geschwindigkeit ausübt. Je größer die Auflösung der Eingangsquelle, desto langsamer wird das Netz. Werden die FPS vom ersten Test des Frameworks 5.3 (9 FPS) mit dem aktuellen Test 352x352 (3,46 FPS) verglichen, wird deutlich, dass die FPS geringer ausfallen. Dafür gibt es zwei Möglichkeiten: Die erste ist, dass durch das Multithreading das gesamte System des Jetson TK1 ausgebremst wird. Die zweite Möglichkeit kann die verwendete Programmiersprache Python sein. Im Gegensatz zu c++, muss Python von einem Interpreter interpretiert werden. Dadurch entstehen höhere Laufzeiten der Programme.

Für einen besseren Überblick ist die Tabelle 5.1 in der nachfolgenden Abbildung 5.6 als Balkendiagramm dargestellt.

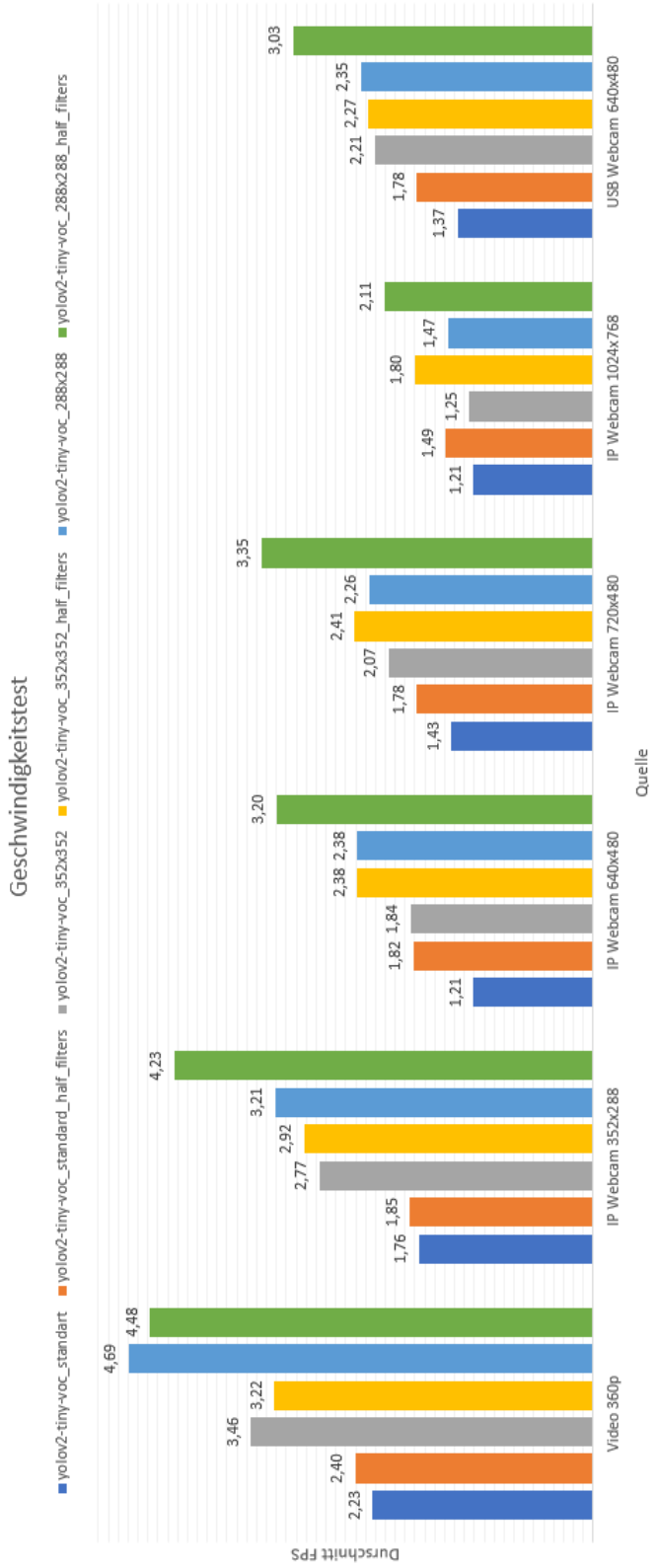

Abbildung 5.6.: Übersicht der Geschwindigkeit der Netze

Da es um Echtzeitdetektion geht, sind die interessanten Werte die, der IP und USB Kamera. Wie in der Abbildung 5.6 zu erkennen, liegen die FPS zwischen 1,21 FPS und 4,23 FPS. Wird ein Vergleich zwischen der IP Kamera mit einer Auflösung von 640x480 und der USB Kamera mit derselben Auflösung gezogen, zeigt sich, dass dort kein signifikanter Unterschied besteht. Wird davon ausgegangen, dass eine FPS Rate von 1,5 FPS für die Detektion von Schiffen reicht, stehen noch 23 Kombinationen aus dem Test zur Verfügung. Damit kann das System Schiffe in Echtzeit detektieren.

Nachstehend sind die einzelnen Kurven der verschiedenen Netze mit der IPWebcam und der Auflösung von 640x480 dargestellt 5.7.

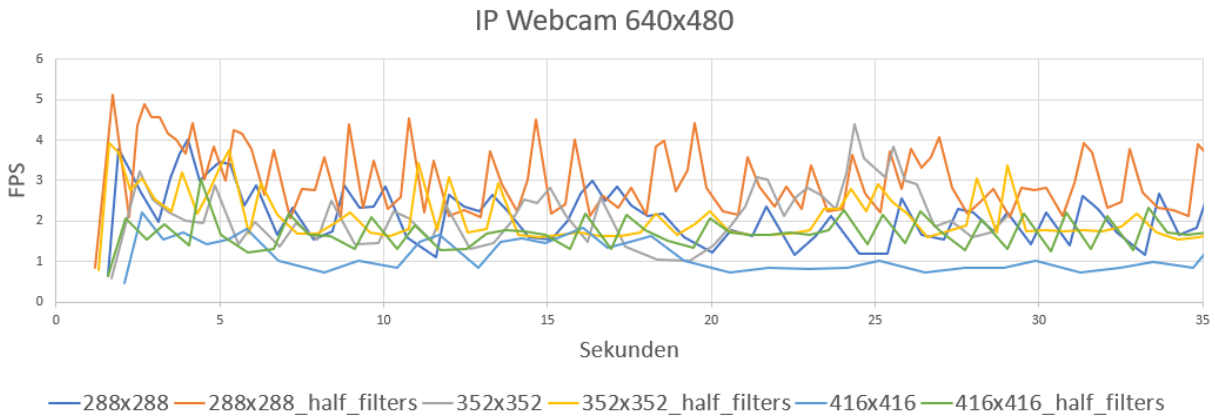

Abbildung 5.7.: Testergebnis IPWebcam 640x480

Auffällig in der Abbildung 5.7, sind die großen Schwankungen der FPS. Je größer das Netzwerk ist, desto kleiner fallen diese Schwankungen aus.

5.2.2. Performance

In diesem Abschnitt geht es um das Testen der Detektionsraten von den unterschiedlichen Netzen. Dafür wird das zuvor erläuterte Python Skript 5.5 processpics.py verwendet. Alle zu testenden Bilder sind unterschiedlich und enthalten Schiffe verschiedener Art. Die Anzahl der zu testenden Bilder beträgt 50. Jedes der Netze wird mit dem exakt gleichen Datensatz getestet um einen ordentlichen Vergleich zu erzielen. Außerdem besitzen die zu testenden Bilder unterschiedliche Auflösungen. Anschließend wird die Detektionsrate wie folgt gebildet 5.1.

$$Detektionsrate = \frac{Erkannte\ Bilder}{Anzahl\ Bilder} \tag{5.1}$$

Die Ergebnisse der Tests sind in der nachstehenden Tabelle 5.2 zu sehen.

Netz	Anzahl Bilder	Erkannte Bilder	Detektionsrate
yolov2-tiny-voc_standart	50	23	0,46
yolov2-tiny-voc_standart_half_filters	50	20	0,40
yolov2-tiny-voc_352x352	50	23	0,46
yolov2-tiny-voc_352x352_half_filters	50	20	0,40
yolov2-tiny-voc_288x288	50	14	0,28
yolov2-tiny-voc_288x288_half_filters	50	12	0,24

Tabelle 5.2.: Performance der Netzwerke

Bei einem Blick in die Tabelle 5.2 ist zu erkennen, dass die Netze mit der Auflösung von 416x416 die selbe Detektionsrate besitzen, wie die Netze mit 352x352. Werden die Netze hingegen mit der Auflösung von 288x288 betrachtet fällt auf, dass diese eine ca. 40% schlechtere Rate aufweisen. Der Unterschied zwischen der normalen Anzahl an Filtern und der halbierten, beträgt circa 13%. Um sich einen kleinen Einblick zu verschaffen, sind auf den folgenden Abbildungen 5.8 und 5.9 zwei Ergebnisse des Tests dargestellt.

Abbildung 5.8.: Beispiel Detektion mit yolov2-tiny-voc_standart

Die obere Abbildung 5.8 zeigt, das Ergebnis einer Detektion mit Hilfe des yolov2-tiny-voc_standart Netzes. Das Eingangsbild besitzt eine Auflösung von 1620x1080 Pixel. Es wurden beide Schiffe erfolgreich erkannt.

Abbildung 5.9.: Beispiel Detektion mit yolov2-tiny-voc_352x352_half_filters

Die Abbildung 5.9 ist das Ergebnis vom yolov2-tiny-voc_352x352_half_filters Netz. In diesem Fall hatte das Bild eine Auflösung von 262x192 Pixel und ist somit bedeutend kleiner als das erste. Trotz der kleinen Auflösung wird ein Schiff erfolgreich erkannt. Die Tatsache, dass nur eines der drei Schiffe erkannt wird, ist entweder auf die geringe Auflösung des Eingangbildes oder auf den Trainingsdatensatz zu schließen. Das yolov2-tiny-voc_standart Netz erkennt ebenso wie das yolov2-tiny-voc_352x352 Netz, nur eins der drei Schiffe. Weitere Testergebnisse können dem Anhang entnommen werden.

6. Fazit und Ausblick

Im Rahmen dieser Arbeit sollte eine Implementierung von faltenden neuronalen Netzen für Detektionsaufgaben von Schiffen auf einer NVIDIA-CUDA-Plattform erfolgen und anschließend die Echtzeitfähigkeit des Systems untersucht werden. Die Hardware sollte unter den Aspekten möglichst kompakt und kostengünstig gewählt werden.

Diese Zielsetzung wurde vollständig umgesetzt. Die Hardware wurde unter den vorher definierten Aspekten gewählt und die passende Software für die Aufgaben implementiert. Dabei stellte sich heraus, dass ein großer Aufwand der Arbeit darin bestand, die richtige Software vernünftig zum laufen zu bekommen. Wichtig dabei war die Auswahl der unterschiedlichen Netzwerke mit verschiedenen Eingangs- und Filtergrößen. Des Weiteren war es wichtig, alle Netze mit demselben Datensatz zu trainieren, um einen vernünftigen Vergleich erzielen zu können. Es zeigte sich, dass die Netze mit der Auflösung von 352x352 für die Aufgabe am besten geeignet sind. Zusätzlich stellte sich heraus, dass es keinen signifikanten Unterschied im Bezug auf die Quelle gibt. Die USBWebcam, als auch die IPWebcam mit der gleichen Auflösung, erzielten ähnliche Ergebnisse. Die Auflösung der Quelle hat in diesem Fall die größte Auswirkung auf die Geschwindigkeit der Netze gehabt. Weiterhin war die Implementierung von Multithreading wichtig, um eine Echtzeitfähigkeit des Systems zu erlangen. Das Ergebnis zeigt, dass das System für eine Echtzeitdetektion bedingt geeignet ist. Vor allem wenn es um die Erkennung von langsamen Objekten geht, wie z.B. Schiffen, sind die erreichten FPS ausreichend. Sobald schnellere Objekte wie, z.B. Tiere, erkannt werden sollen oder extrem hohe Auflösungen untersucht werden, ist die Implementierung auf eine schnellere Hardware sinnvoll.

Durch das stetig steigende Interesse in dem Bereich des Deep-Learnings, wird die Hardware stets weiterentwickelt. Es gibt immer mehr Chips, die speziell für solche Aufgaben entwickelt werden. Beispiele dafür, sind der Orange Pi AI Stick 2801 Features Lightspeeur 2801S Neural Processor, welcher am Ende des letzten Jahres (2018) vorgestellt wurde und der Intel Movidius Neural Compute Stick. Beides sind USB Sticks im unteren Preissegment und sind für unter 100 Euro erhältlich. Diese sind mit speziellen Chips für die Berechnung von neuronalen Netzen bestückt. Der Intel Stick besitzt sogar die Eigenschaft zu kaskadieren, um eine besser Leistung mit mehreren Sticks zu erzielen. Dies wäre ein interessantes Thema, für eine Weiterentwicklung dieses Systems um höhere Geschwindigkeiten zu erreichen und

somit schnellere Objekte detektieren zu können. Eine weitere Anregung wäre eine Langzeitprüfung des hier vorgestellten Systems unter realen Bedingungen.

Literaturverzeichnis

[1] AZEVEDO, Frederico: *Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain*. 2009. – URL https://www.ncbi.nlm.nih.gov/pubmed/19226510

[2] BECKER, Roland: *MACHINE DEEP LEARNING - WIE LERNEN KÜNSTLICHE NEURONALE NETZE?*. – URL https://jaai.de/machine-deep-learning-529/. – Zugriffsdatum: 2019-01-01

[3] CHABLANI, Manish: *YOLO - You only look once, real time object detection explained*. – URL https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9 – Zugriffsdatum: 2019-01-17

[4] DANQING, Liu: *A Practical Guide to ReLU*. 2017. – URL https://medium.com/tinymind/a-practical-guide-to-relu-b83ca804f1f7. – Zugriffsdatum: 2019-01-09

[5] ELINUXTK1: *Jetson TK1*. – URL https://elinux.org/Jetson_TK1. – Zugriffsdatum: 2018-12-06

[6] ELINUXTX1: *Jetson TX1*. – URL https://elinux.org/Jetson_TX1. – Zugriffsdatum: 2018-12-06

[7] ELINUXTX2: *Jetson TX2*. – URL https://elinux.org/Jetson_TX2. – Zugriffsdatum: 2018-12-06

[8] GANDHI, Rohith: *R-Cnn, Fast R-CNN, Faster R-CNN, YOLO - Bject Detection Algortihms*. 2018. – URL https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d5 – Zugriffsdatum: 2019-01-09

[9] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[10] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning - Das umfassende Handbuch*. 1. Aufl. Frechen : mitp Verlag GmbH & Co. KG, 2018. – ISBN 978-3-95845-700-3

[11] HEINZ, Sebastian: *Deep Learning - Teil 1 Einführung*. – URL https://www.statworx.com/de/blog/deep-learning-teil-1-einfuehrung/. – Zugriffsdatum: 2019-01-16

[12] KLEIN, Bernd: *Python-Kurs Geschichte*. – URL https://www.python-kurs.eu/entstehung_python.php. – Zugriffsdatum: 2019-01-16

[13] LITZEL, Nico: *Was ist ein Neuronales Netz?*. – URL https://www.bigdata-insider.de/was-ist-ein-neuronales-netz-a-686185/. – Zugriffsdatum: 2018-11-29

[14] NAKAHARA, Hiroki: *A Lightweight YOLOv2: A binarized CNN with a parallel support vector regression for an FPGA*. – URL https://www.slideshare.net/HirokiNakahara1/fpga2018-a-lightweight-yolov2-a-binarized-cnn-with-a-parallel-suppo – Zugriffsdatum: 2019-01-02

[15] NVIDIA: *CUDA TOOLKIT DOCUMENTATION*. – URL https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. – Zugriffsdatum: 2019-01-06

[16] NVIDIA: *WAS IST GPU COMPUTING?*. – URL https://www.nvidia.de/object/back-gpu-computing-de.html. – Zugriffsdatum: 2018-12-10

[17] NVIDIA-WHITEPAPER: *Darknet: Open Source Neural Networks in C*. – URL https://www.nvidia.com/content/pdf/tegra_white_papers/tegra-k1-whitepaper.pdf. – Zugriffsdatum: 2018-12-23

[18] OPENCV.ORG: *About-OpenCV*. – URL https://opencv.org/about.html. – Zugriffsdatum: 2018-12-26

[19] PATEL, Shyamal ; PINGEL, Johanna: *Introduction to Deep Learning: What Are Convolutional Neural Networks?*. – URL https://de.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-network html. – Zugriffsdatum: 2018-12-30

[20] REDMON, Joseph: *Darknet: Open Source Neural Networks in C*. – URL https://www.computerwoche.de/a/entwickler-frameworks-fuer-machine-learning,3224484. – Zugriffsdatum: 2018-12-20

[21] REDMON, Joseph ; DIVVALA, Santosh ; GIRSHICK, Ross ; FARHADI, Ali: *Installing Darknet*. – URL https://pjreddie.com/darknet/install/. – Zugriffsdatum: 2019-01-04

[22] REDMON, Joseph ; DIVVALA, Santosh ; GIRSHICK, Ross ; FARHADI, Ali: *You Only Look Once: Unified, Real-Time Object Detection*. – URL https://pjreddie.com/media/files/papers/yolo.pdf. – Zugriffsdatum: 2019-01-01

[23] ROSEBROCK, Adrian: *Intersection over Union (IoU) for object detection*. – URL https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/. – Zugriffsdatum: 2019-01-08

[24] STATSOFT: *Übertrainieren von neuronalen Netzen*. – URL https://www.statsoft.de/glossary/O/OverlearninginNeuralNetworks.htm. – Zugriffsdatum: 2019-01-09

[25] SUAREZ-PANIAGUA, Victor ; SEGURA-BEDMAR, Isabel: *Evaluation of pooling operations in convolutional architectures for drug-drug interaction extraction*. – URL https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-018-2195-1. – Zugriffsdatum: 2019-01-01

[26] TIJTGAT, Nils: *Understanding YOLOv2 training output*. 2017. – URL https://timebutt.github.io/static/understanding-yolov2-training-output/. – Zugriffsdatum: 2019-01-09

[27] WILLIAMS, Tom: *CUDA*. 2014

[28] YEGULALP, Serdar ; HÜLSBÖMER, Simon: *Entwickler-Frameworks für Machine Learning*. – URL https://www.computerwoche.de/a/entwickler-frameworks-fuer-machine-learning,3224484. – Zugriffsdatum: 2018-12-20

[29] ZIMMERMANN, Bruno: *Individualisierte Gesichtsdetektion mit YOLOv2 in Darknet*. – URL https://digitalcollection.zhaw.ch/bitstream/11475/7114/2/YOLO-Face_ECC_Presentation.pdf. – Zugriffsdatum: 2019-01-16

A. Anhang

A.1. Inhalt der beiliegenden CD

Auf der beiliegenden CD befinden sich folgende Inhalte:

- die vorliegende Arbeit als PDF
- Ordner cfg: alle benötigten Config-Dateien
- Ordner Paper: alle PDF Dateien der Quellen
- Ordner Skripte: alle benötigten Python Skripte
- Ordner Test
  - Unterordner ErgebnisGeschwindigkeit: Ergebnisse des Geschwindigkeitstest
  - Unterordner ErgebnisPerformance: Ergebnisse des Performancetest
  - Unterordner Testbilder: verwendete Bilder des Performancetest
  - 360ptest.mp4 Testvideo der Arbeit
- Ordner Trainierte_Netzwerke: fertig trainierte Netze
- Ordner Training
  - Unterordner Trainigsbilder_mit_label: Trainingsdatensatz inklusive Labels
  - Unterordner Trainingskurven: Ergebnis des Trainings

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 21. Januar 2019	
Ort, Datum	Unterschrift