Hochschule Karlsruhe – Technik und Wirtschaft
Fakultät für Maschinenbau und Mechatronik

Verkehrszeichenerkennung für autonome Fahrzeuge

Bachelorarbeit (B.Eng.)

von
Tobias Busch
geb. am 26. 02. 1993
in Rastatt
Matr.-Nr.: 44299

Betreuer der Hochschule Karlsruhe
Referent: Prof. Dr.-Ing. Ferdinand Olawsky
Korreferent: Dipl.-Ing. (FH) Bernhard Beck

Karlsruhe, 21. 11. 2016 bis 21. 03. 2017

Erklärung

Ich versichere hiermit wahrheitsgemäß, die Abschlussarbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles einzeln kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 15. März 2017

Unterschrift:

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, die mich bei der Erstellung dieser Bachelorarbeit unterstützt haben. Besonderer Dank gebührt meinem Betreuer, Herrn Prof. Dr.-Ing Ferdinand Olawsky, welcher mir diese Abschlussarbeit ermöglicht und mich bei Fragen und auftretenden Problemen stets motiviert unterstützt hat. Weiterer Dank gilt Herrn Dipl.-Ing. Bernhard Beck, an den ich mich bei fachspezifischen Fragen jederzeit wenden konnte. Abschließend möchte ich meiner Familie und meinen Freunden für die Unterstützung während meines Studiums danken.

Kurzfassung
Verkehrszeichenerkennung für autonome Fahrzeuge

Die vorliegende Arbeit beschäftigt sich mit den Herausforderungen von autonomen Fahrzeugen im Straßenverkehr. Betrachtet wurde die autonome Fahrt an sich, wie auch das Erkennen von Verkehrszeichen und Straßenmarkierungen in Kamerabildern. In dieser Arbeit wird zum einen die Entwicklung eines durch einen Laptop autonom gesteuerten Modellautos beschrieben, zum anderen die Integration und Validierung zweier auf maschinellen Lernen basierender Objektdetektionssysteme, welche auf die Detektion von Verkehrszeichen trainiert wurden. Es wurde sich dafür an ähnlichen Projekten [1, 2] orientiert und die Erkenntnisse dieser genutzt und darauf aufgesetzt. Das Ergebnis dieser Arbeit ist ein Fahrzeug, dem es möglich ist, autonom gesteuert durch Kameradatenauswertung mit Hilfe eines trainierten Künstlichen Neuronalen Netzes, eine Teststrecke abzufahren. Zudem ist eine Verkehrszeichenerkennung durch eine auf Haar-ähnlichen Merkmalen basierende Kaskade aus Klassifikatoren, wie auch durch eine spezielle CNN-Architektur namens YOLO ermöglicht worden. Durch Bewertung der Bildverarbeitungsrate, Detektionsqualität und Speicheranforderungen stellte sich heraus, dass nur das Haar-Kaskaden-Klassifikator-Verfahren für eine effiziente Verkehrszeichenerkennung geeignet ist.

Abstract
Detection of Traffic Signs for Autonomous Vehicles

The work in hand deals with the challenges of autonomous vehicles in road traffic. The autonomous driving itself, as well as the detection of traffic signs and street markings in camera images, was considered. This work describes the development of a model car autonomously controlled by a laptop, as well as the integration and validation of two object detection systems based on machine learning, which were trained to detect traffic signs. It was orientated on similar projects [1, 2] and the knowledge of these were used for further development. The result of this work is a vehicle which can autonomously follow a test track, by camera image evaluation through an trained artifial neural network. In addition, traffic sign recognition has been made possible by a cascade of boosted classifiers based on haar-like features as well as by a special CNN architecture called YOLO. By evaluating the image processing rate, detection quality, and memory requirements, it was found that only the haar cascade classifier method is suitable for an efficient traffic sign detection.

Inhaltsverzeichnis

Nomenklatur

Abkürzungen

CNN	Convolutional Neural Network
CNTK	Computational Network Toolkit
CSI	Camera Serial Interface
cuDNN	NVIDIA CUDA Deep Neural Network library
DC	Direct current
FPS	Frames per second
GPIO	General-purpose input/output
GPU	Graphics processing unit
GTSDB	The German Traffic Sign Detection Benchmark
GTSRB	The German Traffic Sign Recognition Benchmark
HOG	Histogram Of Oriented Gradients
IC	Integrated circuit
ILSVRC	Large Scale Visual Recognition Challenge
KNN	Künstliches Neuronales Netz
MLP	Multilayer Perceptron
RAM	Random Access Memory
RCNN	Regions with CNN features
ROI	Region of Interest
ROS	Robot Operating System
PWM	Pulsweitenmodulation
SIFT	Scale Invariant Feature Transform
SSH	Secure Shell
SURF	Speeded Up Robust Features
SVM	Support Vector Machine

1 Einleitung

Das autonome Fahren im Straßenverkehr ist ein hochaktuelles Forschungsgebiet. Bei autonomen Fahrzeugen handelt es sich um intelligente Systeme, die sich fahrerlos bewegen und durch die Auswertung ihrer Sensoren auf ihre Umwelt reagieren können. Firmen wie Tesla, BMW, Audi, Mercedes oder Google haben das Potential dieses Forschungsgebietes erkannt und sind in diesem Bereich sehr engagiert. Autonome Fahrzeuge könnten Menschen nicht nur das Steuern abnehmen, sie könnten auch für mehr Sicherheit und weniger Emissionen im Straßenverkehr sorgen. Voraussetzung dafür ist eine effiziente Auswertung von Sensordaten. Die künstliche Intelligenz und vor allen Dingen der Bereich des maschinellen Lernens, hat sich bei der autonomen Fahrt als sehr effektiv erwiesen. Maschinen können durch Algorithmen lernen die Welt zu verstehen und richtige Entscheidungen zu treffen. Das Halten der Fahrspur und die Erkennung und Reaktion auf Verkehrszeichen sind wichtige Aufgaben der autonomen Fahrt. Durch Aufnahme der Fahrstrecke mit einer Kamera und Auswerten der Bilder mit Algorithmen des maschinellen Sehens und Lernens können diese beiden Aufgaben erfüllt werden. Die Aufgabenstellung dieser Arbeit kann in die vorgestellte Thematik eingeordnet werden.

1.1 Aufgabenstellung

Ziel dieser Arbeit ist es, unterschiedliche Objekterkennungsalgorithmen zu betrachten und eine Verkehrszeichenerkennung mit diesen zu realisieren. Es soll eine durch einen Laptop gesteuerte autonome Fahrt eines Modellfahrzeugs realisiert werden, welches eine beliebige Teststrecke abfahren und durch Objekterkennungsalgorithmen auf Verkehrszeichen reagieren kann. Die eingesetzten Methoden sollen getestet und bewertet werden.

1.2 Gliederung

Die vorliegende Arbeit gliedert sich in neun Kapitel.

Kapitel 1 beinhaltet eine Einleitung in die Thematik und stellt die Aufgabenstellung dar.

Das Kapitel 2 geht auf den Aufbau und die manuelle Steuerung des Modellfahrzeugs ein. Es wird auf die Hardwarekomponenten und deren Funktion eingegangen. Zudem wird das Robot Operating System vorgestellt. Im letzten Teil des Kapitels wird die Funktion der manuellen Steuerungssoftware des Fahrzeugs beschrieben.

Kapitel 3 beschreibt die Schritte und theoretischen Grundlagen zur Realisierung einer autonomen Fahrt des Modellfahrzeugs. Dies beinhaltet die hard- und softwareseitige Integration eines Kameramoduls in den Fahrzeugaufbau. Es folgt eine Einführung in die Grundlagen von Künstlichen Neuronalen Netzen. Daraufhin wird aufgezeigt, wie Trainingsdaten für das KNN gesammelt werden konnten und wie diese vor dem Training aufbereitet wurden. Anschließend

wird auf die Realisierung des Trainings, der ausgewählten KNN-Architektur, mit Tensorflow eingegangen. Das Kapitel wird durch die Beschreibung des Softwareprogramms abgeschlossen, welche das Fahrzeug autonom fahren lässt.

Kapitel 4 soll als Einführungskapitel, der in dieser Arbeit genutzten Objektdetektionssysteme dienen. Außerdem wird auf den Stand der Technik, der computergestützten visuellen Objekterkennung in 2-D Bilddaten eingegangen.

Kapitel 5 beschäftigt sich mit dem ersten ausgewählten Objektdetektionssystem. Es wird die Funktion des Algorithmus beschrieben. Hinzukommend wird aufgezeigt, wie eine Verkehrszeichenerkennung mit diesem Objektdetektionssystem, ermöglicht werden kann. Der letzte Teil des Kapitels beschreibt, wie der Algorithmus in das ROS integriert wurde, um während der autonomen Fahrt Verkehrszeichen zu detektieren.

Kapitel 6 beschreibt das zweite ausgewählte Objektdetektionssystem. Zuerst werden die Grundlagen zu faltenden neuronalen Netzen erläutert. Darauf folgt eine Beschreibung der Funktion des Algorithmus. Im nächsten Teilabschnitt wird erklärt, wie das Verfahren auf eine Erkennung von Verkehrszeichen trainiert werden kann. Im abschließenden Teil des Kapitels wird erklärt, wie sich dieser Algorithmus in das ROS einbinden lässt, um damit Verkehrszeichen bei der autonomen Fahrt zu detektieren.

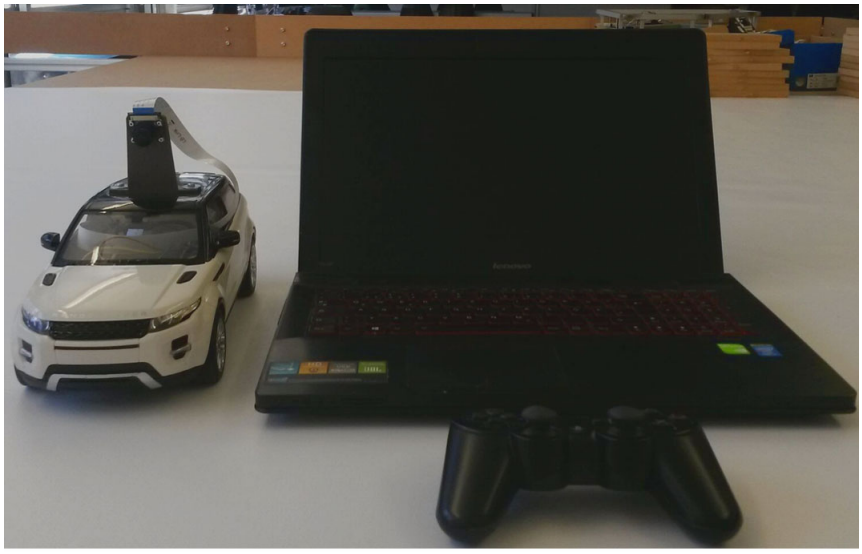
Kapitel 7 beschreibt das Testen der Objektdetektionssysteme und des KNN auf Bildverarbeitungszeiten und Speicheranforderungen.

In Kapitel 8 werden die eingesetzten Methoden bewertet und deren Probleme behandelt.

Kapitel 9 gibt eine Zusammenfassung der Bachelorarbeit und geht auf die erarbeiteten Ergebnisse ein. Zudem wird ein Ausblick, für das Aufsetzen weiterer Untersuchungen auf die Ergebnisse dieser Arbeit gegeben.

2 Aufbau und manuelle Steuerung des Fahrzeugs

Für die spätere Entwicklung eines autonomen Fahrzeugs wurde zuerst ein Modellauto so umgebaut, dass es über einen Laptop manuell gesteuert werden kann. Die manuelle Steuerung dient dabei zur Unfallvermeidung bei der autonomen Fahrt und zudem zum Sammeln von Traningsdaten für ein KNN. Um ein effizientes Fahrzeug zu entwerfen, wurden verschiedene Einheiten ausgewählt und in einem Modellauto kombiniert verbaut. Hierbei ging es nicht darum den bestmöglichen Aufbau zu erschaffen, sondern aus vorhandenen und einfach zu erwerbenden Komponenten eine gut funktionierende Lösung zu finden. Dieses Kapitel beschreibt den Aufbau dieses Steuerungssystems.



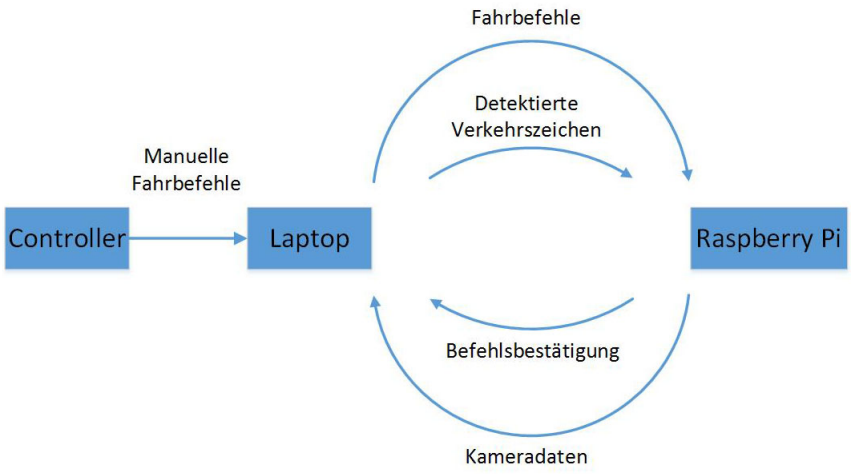


Abbildung 2.1. Das Steuerungssystem.

2.1 Aufbau des Fahrzeugs

Das Grundgerüst des Fahrzeugs ist ein ferngesteuertes Modellauto, welches bei einem Versandhandel erstanden wurde. Das Fahrzeug ist ein Modell des Range Rover Evoque, im Maßstab 1:14. Die Wahl fiel auf dieses Fahrzeug, da es ein gutes Preis-Leistungs-Verhältnis besitzt und sehr viel Stauraum für die vorgesehene Elektronik im Inneren bietet. Angetrieben wird das Fahrzeug durch einen 5V DC Motor. Die Lenkung ist auch durch einen 5V DC Motor realisiert, wobei dadurch nur Kurvenfahrten nach links und rechts mit einem festen Lenkwinkel möglich sind. Das bedeutet die Lenkung kann nur drei verschiedene Positionen einnehmen, was Nachteile beim Fahren mit sich bringt. Jedoch reichen diese Fahreigenschaften für erste Tests im Bereich des autonomen Fahrens aus.

Die gesamte Elektronik zur Ansteuerung der Motoren und zum Empfang von Steuersignalen, wie auch andere unnötige Komponenten, wurden aus dem Fahrzeug entfernt.

Um die Motoren des Fahrzeugs elektrisch steuern zu können, wird der Motortreiber L293D von Texas Instruments verwendet. Durch die in dem Motortreiber befindlichen H-Brücken, können zwei DC Motoren bidirektional gesteuert werden. Zudem ist eine variable Geschwindigkeit der Motoren mit Hilfe von Pulsweitenmodulation (PWM) möglich. Der L293D Motortreiber wurde auf einer Lochrasterplatine mit zusätzlichen Stifleisten verlötet, um mit Steckbrückenkabeln eine einfache Verbindung zu den Steuerungs-, Ansteuerungs- und Versorgungskomponenten zu ermöglichen. Des Weiteren wurde ein Kondensator an den Versorgungseingang des ICs vorgeschaltet, um ihn beim Anlauf der Motoren vor Spannungspitzen zu schützen. Anschließend wurde die Platine mit Schrauben am Fahrgestell fixiert.

Als Steuerungseinheit des Fahrzeugs wird ein Raspberry Pi Model 2 B verwendet. Dieser besitzt einen 900MHz Quad-Core-Prozessor mit einem Cortex-A7 Design, basierend auf der ARM Architektur. Der Arbeitsspeicher beläuft sich auf 1GB RAM. Neben vier USB 2.0 Ports sind unter anderem 40 GPIO Pins verfügbar, sowie ein Ethernet Port, ein CSI Kamera-Interface und ein microSD-Kartenslot. Das Betriebssystem Ubuntu 14.04 LTS wurde gewählt, da dieses in Kombination mit der ROS-Distribution indigo igloo aus Kompatibilitätsgründen empfohlen wird. Als Speicher dient eine 32GB SanDisk HCI microSD-Karte, sie bietet ausreichend Speicherplatz für das Betriebssystem und die benötigte Software. Zudem bietet sie eine schnelle Lesegeschwindigkeit von 80 MB/s. Der WLAN-Stick Ultra-Nano EDIMAX EW-7811Un ermöglicht es, eine Verbindung zum Internet aufzubauen, dies ist essentiell um benötigte Softwarepakete zu installieren. Zudem ist damit ein SSH-Zugang auf den Raspberry Pi möglich, wie auch der Datenaustausch über Netzwerke.

Eine Powerbank der Marke RAVPower versorgt den Raspberry Pi und die Motoren des Fahrzeugs mit ausreichend Energie. Sie besitzt eine Ladekapazität von 16750mAh und ein Gewicht von 281 Gramm. Somit ist der Akku die schwerste Komponente der gesamten Plattform. Die Powerbank hat zwei 5V DC USB-Ausgangsbuchsen, eine mit 2,1A welche an den Raspberry Pi angeschlossen ist und eine mit 2,4A die an den Motortreiber angeschlossen ist. So viel Strom ist notwendig, um den Antriebsmotoren ausreichend Drehmoment beim Anfahren zur Verfügung zu stellen. Durch einen Schalter der unten am Fahrgestell verbaut ist, kann der Stromkreis, der die Elektronik versorgt, unterbrochen werden.

Der Raspberry Pi wurde im Fahrzeuginnenraum platziert und mit allen elektrischen Komponenten verdrahtet. Zum einen ist er mit der Energieversorgung über Micro-USB verbunden. Des Weiteren sind seine GPIO-Pins mit den Ansteuerungspins des Motortreibers verbunden, um diese über PWM ansteuern zu können. In Abbildung 2.2 ist der Gesamtaufbau der elektrischen Schaltung des Fahrzeugs dargestellt.

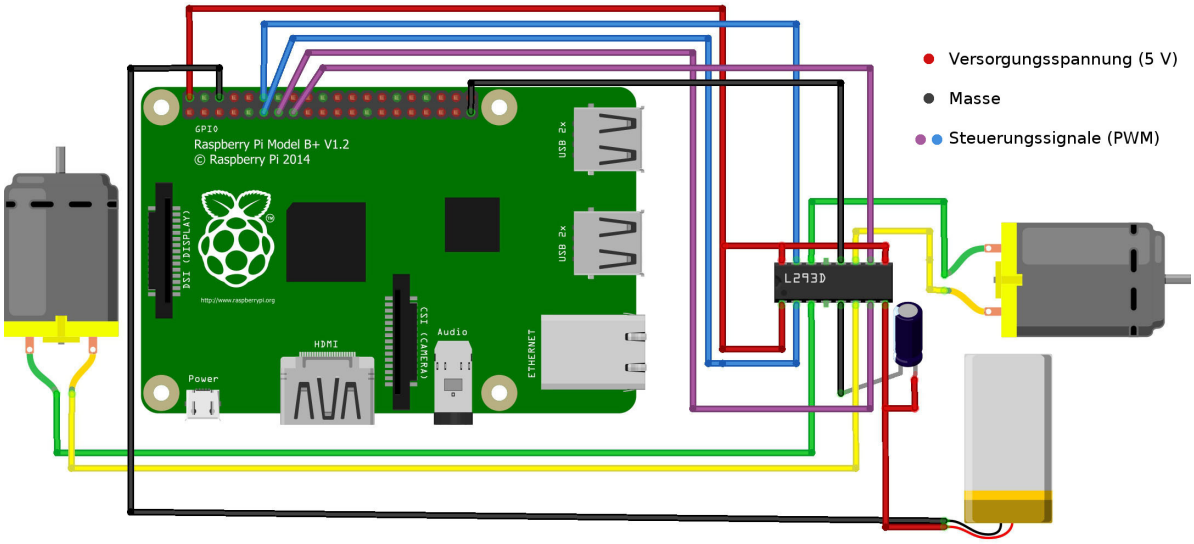


Abbildung 2.2. Schaltungsaufbau.
(Raspberry Pi, Motortreiber, Powerbank, Antriebs- und Steuerungsmotor)

2.2 Die Steuerungseinheit

Da für die Generierung von Fahrbefehlen bei einer autonomen Fahrt sehr viel Rechenleistung benötigt wird, werden die Fahrbefehle von einem Laptop generiert, welche mit einer leistungsfähigeren CPU und zusätzlicher Grafikkarte ausgestattet ist. Deshalb wird das Fahrzeug bei der manuellen Fahrt mit einem PS3 Controller, der über USB mit dem Laptop verbunden ist, gesteuert. Die Fahrbefehle werden dabei über WLAN, mit Hilfe des Robot Operating Systems, an den Raspberry Pi gesendet. Der Lenovo Y510P Laptop wurde verwendet. Er besitzt eine i7-4700MQ CPU, zwei GeForce GT 755M GPUs im SLI-Verbund und 16 GB RAM. Auch auf ihm läuft das Betriebssystem Ubuntu 14.04 LTS.

2.3 Einführung in das Robot Operating System

Das Robot Operating System (ROS) ist ein Open Source Framework, das primär für die Robotik-Forschung entwickelt wurde. Es ist eine Ansammlung von Tools, Programmbibliotheken und Konventionen zum Erleichtern der Entwicklung komplexer Robotikanwendungen. Das Hauptziel dieses Frameworks ist die Förderung der Wiederverwendung von Code in der Robotik-Forschung und Entwicklung. Verwendbare Programmiersprachen sind hierbei C++

und Python. Die ROS-Distribution indigo igloo wurde auf beiden Systemen, dem Lenovo Y510P wie auch auf dem Raspberry Pi, integriert. Auf die in dieser Arbeit genutzten ROS-Features wird in diesem Abschnitt eingegangen.

2.3.1 Nodes

ROS ist ein verteiltes System bestehend aus Prozessen, sogenannten Nodes, welche es ermöglichen individuell ausführbare Programme zur Laufzeit einfach miteinander kommunizieren zu lassen. Diese Prozesse sind freilaufend über die ROS-Kommunikationsinfrastruktur verbunden.

2.3.2 Topics

Topics sind Datenbusse über welche ROS-Nachrichten asynchron ausgetauscht werden können. Nodes können Daten auf verschiedene Topics senden (publish), während andere Nodes diese Topics auslesen (subscribe) können, um die Daten weiterzuverarbeiten. Dadurch lässt sich die Produktion und der Konsum von Informationen entkoppeln. Mit einer solchen Softwarearchitektur können unabhängig entwickelte Softwarekomponenten verwendet und in bestehende Systeme integriert werden.

2.3.3 Messages

Um den Datenaustausch über Topics zu vereinfachen, nutzt ROS die Standardisierung von Daten und deren Format. Dies wird mit sogenannten ROS-Messages umgesetzt. Diese Messages beschreiben die Datenstrukturen welche auf ROS-Topics gesendet werden und ermöglichen es ROS-Tools, automatisch Quellcode für den Message-Typ in verschiedenen Programmiersprachen zu generien. Als Beispiel kann die ROS-Message *sensor_msgs/Image* genannt werden, welche die Breite und Höhe wie auch die Pixelcodierung eines auf eine Topic gesendeten Kamera-Frames definiert.

2.3.4 Parameter Server

Auf dem ROS Parameter Server können Parameter aus einer großen Variation von Datentypen abgelegt werden, auf welche Nodes zur Laufzeit zugreifen und die Parameter auslesen oder verändern können. Dieser Server ist dabei nicht auf High-Performance ausgelegt, er wird für statische Daten, wie etwa Konfigurations-Parameter, benutzt.

2.3.5 Packages

In ROS wird Software in Packages verarbeitet. Ein Package kann eine ROS-Node enthalten, eine von ROS unabhängige Programmbibliothek, ein Datenset von Konfigurations-Dateien, die Software eines Drittanbieters, kurz gesagt jede mögliche Art von nützlichen Software-Modulen. Dabei ist das Ziel dieser Packages ein einfaches Wiederverwenden von Softwarefunktionalitäten zu ermöglichen.

2.3.6 Multiple Machines

Mit ROS ist es zudem möglich, eine Kommunikation zwischen Nodes herzustellen die auf verteilten Plattformen ausgeführt werden. In dieser Arbeit sind die verteilten Plattformen der Raspberry Pi und der Laptop, die zusammen ein Steuerungssystem bilden sollen. Benötigt wird dafür ein gemeinsames Netzwerk über welches Daten ausgetauscht werden können. Im Fall dieser Arbeit ist dies ein WLAN-Netzwerk der Hochschule. Den beiden Plattformen wurden feste IP-Adressen in diesem Netzwerk zugewiesen.

Der Laptop kann mit folgendem Eintrag in der .bashrc als Master dieses Kommunikationsnetzwerkes initialisiert werden:

export ROS_MASTER_URI=http://<Laptop IP-Adresse>:<Port>

export ROS_IP=<Laptop IP-Adresse>

Mit dem folgenden Eintrag in der .bashrc des Raspberry Pi kann selbiger als Slave des Kommunikationsnetzwerkes initialisiert werden:

export ROS_MASTER_URI=http://<Laptop IP-Adresse>:<Port>

export ROS_IP=<Raspberry Pi IP-Adresse>

Mit dem Starten von roscore und der zusätzlichen Festlegung des Ports auf dem kommuniziert werden soll, wird der Datenaustausch zwischen den verteilten Systemen über das Netzwerk freigegeben.

roscore -p<Port>

2.4 Manuelle Steuerung des Fahrzeugs

Um das Fahrzeug von dem Laptop aus über einen PS3-Controller steuern zu können, wurde das ROS-Package *joy* benutzt. Zudem wurde ein eigenes Package, in C++ implementiert. Dieses nennt sich *origami_bot*. Auf die Funktion der in den Packages enthaltenen Nodes und deren Auswirkungen auf die Steuerung wird nun näher eingegangen.

2.4.1 Die *joy_node* Node

Die *joy_node* Node welche im *joy* Package enthalten ist, stellt einen ROS-Treiber für einen auf Linux basierenden Joystick dar. In diesem Fall ist dieser Joystick ein PS3 Controller, der mit dem Laptop über eine USB-Schnittstelle verbunden ist. Die Node sendet eine *Joy* Message auf die Topic */joy*, welche den gegenwärtigen Zustand der Befehlstaster und Analogsticks des Controllers repräsentiert.

2.4.2 Die *motor_controller* Node

Die *motor_controller* Node, welche sich auf dem Raspberry Pi befindet, greift auf die Topic */joy* zu und liest somit die betätigten Kommandos auf dem PS3 Controller aus. Diese werden anschließend durch die inkludierte Programmbibliothek *WiringPi*, welche es ermöglicht die GPIO-Pins des Raspberry Pi anzusteuern, in PWM-Signale umgewandelt. Die GPIO-Pins steuern somit die Input-Pins des Motortreibers über PWM an. Ist ein GPIO-Pin auf High gesetzt, schaltet der Motortreiber den angesteuerten Stromkreis durch. Es wird ein bestimmter Motor mit der Spannung der Powerbank versorgt und das Fahrzeug kann sich fortbewegen. Durch den PS3 Controller ist es zusätzlich möglich, den Tastgrad des PWM-Signals zu verändern. Die Veränderung des Tastgrads führt dabei zu einer Geschwindigkeitsänderung des Fahrzeugs.

3 Autonome Steuerung des Fahrzeugs

Dieses Kapitel beschreibt den Umbau des manuell steuerbaren Fahrzeugs zu einem autonomen Fahrzeug. Da diese Arbeit ein sehr weitreichendes und komplexes Themengebiet umfasst, wurde die Realisierung einer einfachen autonomen Kurvenfahrt durch Künstliche Neuronale Netze als ausreichend angesehen, um anschließend darauf aufbauen zu können. In diesem Kapitel wird zuerst auf die Integration des Kamerasensors eingegangen, welcher die Fahrweginformationen generiert. Es folgt eine Einführung zu Künstlichen Neuronalen Netzen, um ein theoretisches Grundverständnis über die Funktion dieser zu bekommen. Dann wird wird auf das Sammeln und Aufbereiten von Trainingsdaten eingegangen. Anschließend wird gezeigt wie mit den Fahrweginformationen, in Kombination mit einem KNN und dem Robot Operating System, eine autonome Fahrt ermöglicht werden kann.

3.1 Hard- und softwareseitige Integration eines Kamerasensors

Um auf die Umwelt während der autonomen Fahrt reagieren zu können, braucht es Informationen über die Umgebung. In diesem Unterkapitel wird beschrieben wie ein Kamerasensor in das Fahrzeug integriert wurde und wie dessen Bildinformationen verarbeitet und zur Verfügung gestellt werden.

3.1.1 OpenCV

Die Programmierbibliothek OpenCV in der Version 2.4.8 ist in dem ROS bereitgestellt. OpenCV enthält Algorithmen für die Bildverarbeitung und das maschinelle Sehen. OpenCV ist dabei auf die Bildverarbeitung in Echtzeit ausgelegt. Intel leitete die Entwicklung dieser Open Source Bibliothek ein, heute wird sie hauptsächlich von den Entwicklern von Willow Garage gepflegt, einem Robotik Forschungsinstitut, welches sich auch um das Robot Operating System kümmerte. Alle in dieser Arbeit erwähnten Bildverarbeitungsoperationen wurden dabei mit Hilfe von OpenCV realisiert.

3.1.2 Einbau des Kamerasensors

Um ein autonomes Fahren zu realisieren, wurde ein Kamerasensor in das Fahrzeug eingebaut, über dessen Daten gute Rückschlüsse auf geeignete Fahrkommandos gezogen werden können und somit das Fahrzeug in der gewünschten Spur gehalten werden kann. Das SainSmart 5MP Kamera-Modul wurde dabei genutzt. Es besitzt einen 5MP OV5647 Sensor und eine Bildauflösung von 2592×1944 Bildpunkten. Es ist zudem eine Bildwiederholungsrate von 90 Bildern pro Sekunde möglich. Ausgestattet ist das Kamera-Modul mit einem Fischaugenobjektiv, welches einen größeren Sichtwinkel ermöglicht und somit auch mehr Informationen über die Umwelt

in einem Frame zur Verfügung stellt. Der Sichtwinkel der Kamera wird dabei mit 160° angegeben.

Die Kamera wurde mit Hilfes eines 15 Pin Flachbandkabels mit der CSI-Schnittstelle des Raspberry Pi verbunden. Diese CSI-Schnittstelle erlaubt eine hohe Datenübertragungsrate, denn sie überführt die Pixeldaten direkt zum BCM2835 Prozessor des Raspberry Pi. Montiert wurde sie an einer eigens angefertigten Kamerahalterung, welche auf dem Dach des Fahrzeugs befestigt ist. Sie wurde hoch über dem Fahrzeug positioniert, um einen besseren Überblick über den Fahrweg zu erhalten. In Abbildung 3.1 ist die montierte Kamera auf dem Fahrzeug zu sehen.

Abbildung 3.1. Fahrzeug mit Kamera.

Bei Kameras im niedrigen Preissegment ist zu beachten, dass Linsenverzerrungen auftreten können, welche das Bild verformen und somit die Bestimmung von Abständen aus dem Kamerabild unmöglich machen. Durch eine Kamerakalibrierung können jedoch kameraspezifische Parameter bestimmt werden. Das installierte Robot Operating System Framework bietet Packages zur Kalibrierung von Kameras an. Mit dem *camera_calibration* Package wurde die Kamera nach einer gegeben Anleitung [3] kalibriert. Dafür muss ein ausgedrucktes Schachbrettmuster auf ein ebenes Brett geklebt und die Kalibrierung durchgeführt werden. Nach der erfolgreichen Kalibrierung wird ein sogenanntes Yaml-Datei erzeugt, welche Parameter zum Herausrechnen der Linsenverzerrung enthält. Der Pfad zu dieser Datei muss der ROS-Node, welche die Kameradaten ausliest, zur Verfügung gestellt werden. Dies erlaubt es mit der entzerrten Version des Bildes zu arbeiten.

3.1.3 Die *raspicam_node* Node

Die *raspicam_node* Node welche im ROS-Package *raspicam* enthalten ist, dient als ROS-Treiber für das SainSmart 5MP Kamera-Modul. Sie liest die Pixelinformationen aus der Kamera aus und sendet die einzelnen Frames im JPEG-Format auf die Topic */camera/image/compressed*. Der Laptop ist dadurch in der Lage die Kameradaten über das bestehende ROS-Netzwerk (Abschnitt 2.3.6) auszulesen und weiterzuverarbeiten. Beim Starten der Node können dieser noch definierte Parameter beigefügt werden. Das Bildformat kann durch die Parameter *_height* für die Höhe und *_width* für die Breite, gewählt werden. Außerdem kann durch den Parameter *_framerate* die Anzahl der aufgenommenen Einzelbilder pro Sekunde festgelegt werden. Um die Latenzzeiten beim Übertragen der Daten so gering wie möglich zu halten, wird ein 240×320 Bildpunkte großer Frame mit 3 Channeln (RGB-Farbraum) und einer Bildfrequenz von 15 Einzelbildern pro Sekunde verwendet. Zum Starten der Node mit diesen genannten Parametern muss folgender Befehl in der Kommandozeile eingegeben werden:

rosrun raspicam raspicam_node _framerate:=15 _height:=240 _width:=320

3.2 Einführung in Künstliche Neuronale Netze

KNN sind durch das Neuronennetz des menschlichen Gehirns inspiriert. Es wird dabei versucht die Vorgänge im Zentralnervensystem des Gehirns mit Computern zu simulieren. Realisiert wird dies durch Software, welche das Verhalten von Neuronen untereinander simuliert. KNNs sind lernfähig, sie generieren Informationen aus zugeführten Daten-Beispielen indem sie durch Training die Verbindungen der einzelnen Neuronen anpassen. Durch das Training können anschließend gelernte Folgerungen aus unbekannten Daten gezogen werden. Um aus den Bildinformationen des Fahrwegs passende Fahrkommandos erzeugen zu können, welche das Fahrzeug in der gewünschten Spur halten, wurde auch ein Künstliches Neuronales Netz verwendet. Dieses Unterkapitel dient als Einführung in das Gebiet der KNNs und soll die Grundfunktion dieser verständlich machen.

3.2.1 Biologische und künstliche Neuronen

Neuronen sind Zellen oder Nervenzellen im Gehirn von Lebewesen. Die Hauptaufgabe der Neuronen besteht im Weiterleiten von Signalen beziehungsweise der Kommunikation mit anderen Neuronen. Abbildung 3.2 zeigt eine Veranschaulichung eines solchen biologischen Neurons. Die Dendriten eines Neurons fungieren als Informations-Eingang. Diese Informationen werden anschließend im Zellkern gebündelt. Wenn ein bestimmter Schwellwert an Information überschritten wird, dann feuert das Neuron, das bedeutet es sendet ein Signal über sein Axon aus. Dieses Axon ist nun mit den Dendriten anderer Neuronen verbunden, welche die Informationen weiterverarbeiten können. Dabei kann ein Neuron mit bis zu 10000 anderen Neuronen Verbunden sein. Im menschlichen Gehirn sind circa 100 Milliarden dieser Neuronen im Zusammenschluss. Dieser Zusammenschluss wird als neuronales Netz bezeichnet und ermöglicht es dem Menschen zu denken, zu fühlen und zu erkennen.

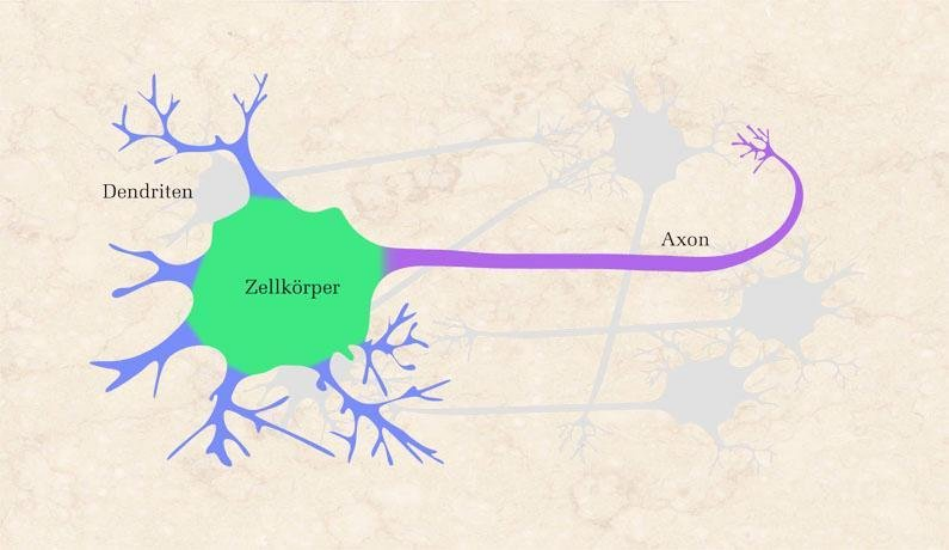


Abbildung 3.2. Biologisches Neuron. [4]

Das Modell eines künstlichen Neurons ist ein simples Konzept. Die Idee besteht darin durch mathematische Beschreibung ein biologisches Neuron nachzuahmen, welches eine informationsverarbeitende Einheit im Gehirn darstellt. Das mathematische Modell eines Neurons ist in Abbildung 3.3 dargestellt.

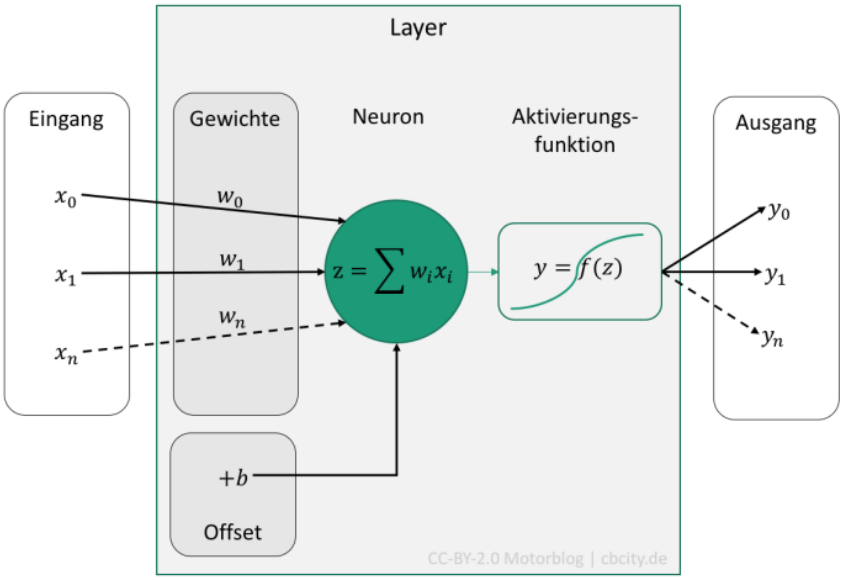


Abbildung 3.3. Künstliches Neuron. [5]

Die Verarbeitungen der eintreffenden Informationen wird dabei in folgenden Schritten realisiert:

1. Die Informationen am Eingang des künstlichen Neurons werden mit veränderbaren Gewichten multipliziert, jeder Eingang besitzt dabei sein eigenes Gewicht.

2. Diese einzelnen gewichteten Informationen werden anschließend summiert.

3. Der Summe dieser gewichteten Informationen wird daraufhin noch ein veränderlicher Wert addiert, der sogenannte Bias.

4. Das Ergebnis wird in eine Schwellwertfunktion auch Aktivierungsfunktion genannt eingespeist, je nach genutzter Funktion wird nun ein entsprechender Ausgabewert an die mit diesem Ausgang verbundenen Neuronen gesendet.

3.2.2 Architektur eines Künstlichen Neuronalen Netzes

Ein Künstliches Neuronales Netz ist aus Schichten aufgebaut, sogenannten Layers. Auf jedem dieser Layer ist eine gewisse Anzahl an Neuronen angesiedelt. Ab einer gewissen Anzahl von Layern ist auch von einem *Deep Neural Network* die Rede. Diese Layer und deren enthaltende Neuronen können auf verschiedene Weise untereinander verbunden sein, jede Verbindungsarchitektur besitzt dabei ihre eigenen Vor- und Nachteile. Eines haben jedoch alle Architekturen gemeinsam, sie besitzen eine Eingangs- und eine Ausgangsschicht. Die Schichten zwischen der Eingangs- und Ausgangsschicht werden *Hidden Layers* genannt. Beim Multilayer Perceptron (MLP) sind alle Neuronen der aufeinanderfolgenden Layer sequentiell miteinander verbunden, diese werden dann *Fully Connected Layer* genannt. Die Architektur eines MLPs wurde in dieser Arbeit zur Realisierung einer autonomen Fahrt verwendet.

3.2.3 Überwachtes Lernen

Es gibt verschiedene Arten von Trainingsvarianten eines KNNs. Um im Rahmen dieser Bachelorarbeit zu bleiben wird hier nur auf das sogenannte *überwachte Lernen* eingegangen. Beim Training eines KNNs können Neuronen entfernt, hinzugefügt, der Bias verändert oder die Aktivierungsfunktion geändert werden. Meist wird unter Lernen aber das Ändern der Gewichte verstanden, welche mit den Informationen am Eingang des Neurons multipliziert werden. Dem Netz wird dabei eine Eingabe an der Eingangsschicht präsentiert und die darauf gewünschte Ausgabe an der Ausgangsschicht. Im Fall der autonomen Fahrt sind dies die Bildinformationen der Fahrbahn am Eingang und der entsprechende Lenkwinkel, der gefahren werden soll, als Ausgabe. Beim Lernen werden die genannten Parameter des KNNs nun so angepasst, dass die Ausgabe des Netzes immer mehr auf die Zielausgabe passt. Wurde das Netz mit einer ausreichenden Anzahl an Trainingsdaten trainiert, ist es ihm möglich aus Unbekannten Eingangsdaten entsprechende Ausgangsdaten zu generieren.

3.2.4 Verlustfunktion

Um den Lernfortschritt bewerten zu können, wird eine sogenannte Verlustfunktion verwendet. Sie stellt beim Lernen das Maß der Diskrepanz zwischen der gewünschten Zielausgabe und der vorhergesagten Ausgabe des KNNs dar. Während der Optimierung des KNNs auf den Trainingsdatensatz, durch Veränderung der Parameter, wird dabei immer versucht die Verlustfunktion zu minimieren und dadurch die Wahrscheinlichkeit der korrekten Vorhersage zu erhöhen. Für verschiedene Anwendungsprobleme gibt es zudem auch unterschiedliche Verlustfunktionen. Ein Beispiel für ein Anwendungsproblem wäre ein Klassifizierungsproblem, bei dem geprüft werden soll, ob Daten einer bestimmten Klasse angehören. Das Auseinanderhalten von Verkehrszeichen in Bildern könnte ein solches Klassifizierungsproblem sein. Auch häufig vorkommend sind Regressionsprobleme, hier soll keine Klasse, sondern ein bestimmter Wert geschätzt werden. Die Positionierung eines Begrenzungsrahmens um ein Objekt in einem Bild kann ein Regressionsproblem sein. Für tiefere Informationen zu Verlustfunktionen sei auf [6] verwiesen und in [7] wird aufgezeigt, für welches Problem, welche Verlustfunktion verwendet werden kann.

3.2.5 Backpropagation

Das Backpropagation-Verfahren oder auch Fehlerrückführungsverfahren wird für das Einlernen von Künstlichen Neuronalen Netzen genutzt. Es gehört zur Gruppe der überwachten Lernverfahren und wurde in dieser Arbeit auf das MLP, welches die autonome Fahrt realisiert angewandt. Dieses Lernfahren arbeitet in drei Schritten:

1. Im ersten Schritt werden den Neuronen Reize präsentiert und dadurch der Output des neuronalen Netzes berechnet.
2. Anschließend wird mit der Verlustfunktion der Verlust jedes Neurons des Output-Layers berechnet. Ist der Verlust unter einer definierten Güteschwelle, wird die Trainingsphase beendet. Falls dies nicht der Fall ist, folgt Schritt drei.
3. Der Verlust wird anschließend vom Output-Layer zum Input-Layer zurückgeführt, für jedes Neuron wird dabei ein Verlustwert berechnet der seinen ungefähren Beitrag zum Gesamtverlust repräsentiert. Der Einfluss jedes einzelnen Gewichts auf die Verlustfunktion kann nun durch die Anwendung der Kettenregel errechnet werden. Dieser Einfluss ist der Gradient des Gewichts, der durch Differentiation berechnet wurde. Die Gradienten der Gewichte werden nun einer Optimierungsmethode übergeben, welche versucht die Verlustfunktion zu minimieren, indem sie die Gewichte verändert.

Für das Vorgehen zur Berechnung der Gradienten sei auf [6] verwiesen.

3.2.6 Optimierungsmethoden

Das Vermindern der Verlustfunktion durch Optimierung, wird durch die Suche eines Minimalwertes im Fehlerraum realisiert. Die Fehlerfunktion setzt sich dabei aus den unterschiedlichen optimierbaren Parametern des KNNs zusammen. Backpropagation wird in Verbindung mit einer Optimierungsmethode genutzt, um das globale Minimum im Fehlerraum des KNNs zu finden. Das Gradientenverfahren ist dabei eines der einfachsten Optimierungsmethoden. Bei

diesem Verfahren werden die Gewichte des KNNs je nach Steigung und Richtung ihres durch Backpropagation berechneten Gradienten verändert. Nach jeder Propagation von Trainingsdaten wird dabei in die Richtung des steilsten Abstiegs, mit einer gewissen Lernrate, optimiert.

In der praktischen Anwendung werden jedoch weitaus komplexere Versionen des Gradientenverfahrens genutzt. Weitere häufig genutzte Optimierungsmethoden sind zum Beispiel:

- Momentum
- Adagrad
- RMSProp
- Adam

Genauere Informationen über diese Methoden liefern [8, 6].

3.2.7 Batch Training

Das in dieser Arbeit genutzte MLP bekommt während des Trainings das gesamte Trainingsdatenset nicht auf einmal (online), sondern immer nur in kleinen Partien zugeführt (Batching). Diese einzelnen Partien werden Batches genannt. Nach jeder Propagation eines Batches durch das KNN wird dieses nach den genannten Verfahren optimiert. Das Arbeiten mit Batches hat den Vorteil, dass weniger RAM beim Training benötigt wird. Batches werden meist bei Trainingsdatensätzen mit hohen Speicheranforderungen und Deep Neural Networks genutzt. Durch dieses Batching wird eine Überschreitung der maximalen Speicherressourcen verhindert, was zum Abbruch des Trainings führen würde.

3.3 Sammeln der Trainingsdaten

Um ein überwachtes Lernen eines KNNs zu ermöglichen, ist es nötig Trainingsdaten zu sammeln. Dafür musste eine Teststrecke aufgebaut werden, mit Eigenschaften, die ein gutes Training begünstigen. Außerdem mussten die Ein- und Ausgangsdaten festgelegt werden, mit denen trainiert werden sollte. Um diese Trainingsdaten aufnehmen zu können wurde zudem eine ROS-Node geschrieben, welche diese Daten generieren und anschließend abspeichern kann.

3.3.1 Die Teststrecke

Als Teststrecke wurde eine einfache Kurvenfahrt gewählt. Diese Strecke diente dazu, Trainingsdaten zu sammeln und das mit diesen Trainingsdaten eintrainierte MLP auf der Teststrecke autonom Fahren zu lassen. Abbildung 3.4 zeigt diese Teststrecke. Der Boden der Teststrecke wurde dabei einfarbig gestaltet, um die Fahrbahnbegrenzungen deutlicher im Bild hervorheben. Zudem wurde darauf geachtet, gleichbleibende Lichtverhältnisse über den gesamten Streckenverlauf zu haben. Durch diese konstanten Teststreckenverhältnisse sollte die Qualität des Trainings erhöht werden. Die Wahl der Fahrbahnbegrenzung fiel auf Pylonen, da das Formula Student Team der Hochschule Karlsruhe auch bei Wettbewerben im Bereich des autonomen Fahrens teilnehmen möchte und deren Fahrbahnen durch Pylonen abgegrenzt sind. Es wurden dabei 120 Pylonen im gleichen Maßstab wie des Fahrzeugs (1:14) gekauft.

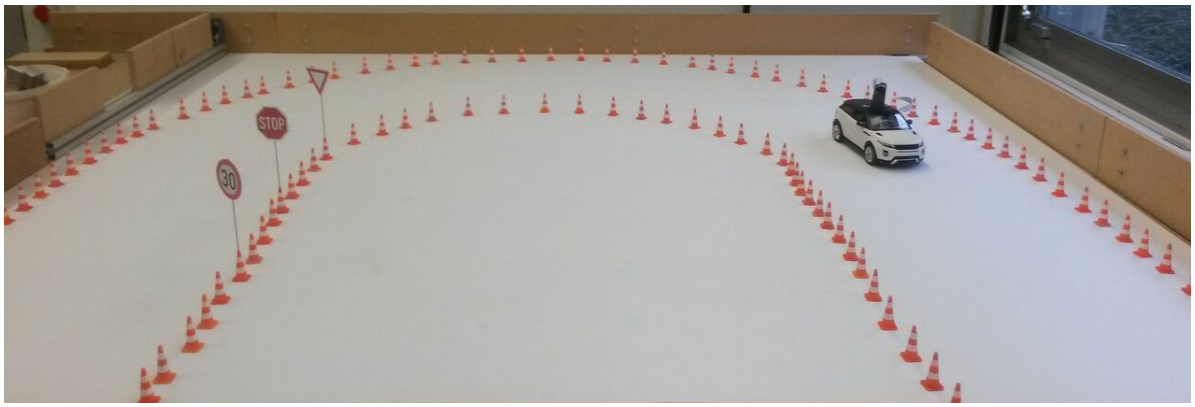


Abbildung 3.4. Die Teststrecke.

3.3.2 Trainingsdaten

Zur Realisierung eines überwachten Lernens des MLPs, mussten die sogenannten *Input-Features* und *Output-Labels*, welche für das Training genutzt werden sollten, festgelegt werden. Nach einiger Recherche wurde der Ansatz aus einem gut dokumentierten und sehr ähnlichen Projekt [1] gewählt. Als *Input-Features* dienen die Pixelinformationen der Kamera auf dem Fahrzeug. Die *Output-Labels*, welche sich durch das Propagieren der Pixelinformationen durch das MLP ergeben sollen, sind die drei möglichen Lenkwinkel des Fahrzeugs.

3.3.3 Die *collect_training_data* Node

Die *collect_training_data* Node wurde mit der Programmiersprache Python geschrieben und in das Robot Operating System integriert. Sie ermöglicht es Trainingsdaten zu sammeln und diese zu speichern. Dabei liest sie zum einen die Topic */camera/image/compressed* (Kapitel 3.1.3) aus, auf welche die Pixelinformationen des Fahrwegs gesendet werden. Zum anderen liest sie auch die Topic */joy* (Kapitel 2.4.1) aus, auf welche die aktuellen Fahrbefehle des PS3-Controllers gesendet werden. Zudem greift die Node auf die Topics */train_starter* und */train_saver* des ROS-Parameter Servers (Kapitel 2.3.4) zu. Nach Starten der Node, kann durch die Ausführung des folgenden Befehls in der Kommandozeile, die Aufnahme von Trainingsdaten gestartet werden:

rosparam set /train_starter „Start“

Je nach betätigtem Fahrmanöver auf dem Controller, werden nun die Pixelinformationen mit den zugehörigen Fahrbefehlen in Arrays gestapelt. Wurden genug Trainingsdaten gesammelt, können diese mit folgendem Befehl in einem Zip-Ordner abgespeichert werden:

rosparam set /train_saver „Save“

3.4 Aufbereitung der Trainingsdaten

Um die Genauigkeit des MLP durch das Training mit dem Trainingsdatenset weiter zu erhöhen, wurden Methoden der Datenaufbereitung auf dieses angewandt. Dadurch sollte eine Steigerung der Lenkwinkelvorhersagegenauigkeit erreicht werden. Diese Methoden wurden jedoch nur nach Empfehlungen aus [6, 1, 2] angewandt und nicht auf ihren wahren Nutzen getestet. Die nun folgenden Methoden wurden mit Python-Code realisiert und auf das Trainingsdatenset angewandt.

3.4.1 Speicherung

Um den benötigten Speicherplatz des Trainingsdatensets zu reduzieren, wurden die gespeicherten Frames mit Hilfe der Programmbibliothek OpenCV in ein 8-Bit Graustufenformat konvertiert. Zudem wurde der Frame so zugeschnitten, dass nur die relevanten Informationen auf diesem enthalten sind. Als relevante Informationen wurden hierbei der Fahrweg und dessen Begrenzungen (Pylonen) betrachtet. Der daraus entstehende Frame wird auch *Region of Interest* (ROI) genannt. In Abbildung 3.5 ist ein solcher Frame, gekennzeichnet mit seinem zugehörigen Label, dargestellt.

Abbildung 3.5. Trainingsframe.

Der zugeschnittene Frame besitzt eine Breite von 320 Pixeln und eine Höhe von 140 Pixeln. Um dieses Bild effektiv speichern und dem MLP zur Verfügung stellen zu können, musste es noch transformiert werden. Sieht man den Frame als eine Matrix aus 140 Zeilen und 320 Spalten an, könnte diese Matrix auch in eine Matrix mit einer Zeile und 140×320 Spalten transformiert werden. Dies wurde durch ein einfaches aneinanderreihen der einzelnen Zeilen realisiert. Diese 44800 ($140 \cdot 320$) aneinandergereihten Pixel konnten somit einfach in einem Array gespeichert und dem Input-Layer des MLPs zugeführt werden.

3.4.2 Mischung

Um das Training zu verbessern, wurde das Trainingsdatenset vorher zufällig durchgemischt. Dieses Mischen sollte Batches (Kapitel 3.2.7) mit hochkorrelierten Bildern vermeiden, da dies die Optimierungsqualität des MLPs beeinflussen könnte.

3.4.3 Normalisierung

Die Pixelwerte jedes Frames liegen in dem Skalenbereich von 0 bis 255 (8-Bit). Jeder einzelne dieser Pixelwerte wurde auf einen Skalenbereich zwischen 0 und 1 normalisiert, indem die Werte durch 255 dividiert wurden. Nach dieser ausgeführten Normalisierung des Datensets sollte eine 40%ige Verbesserung der Vorhersagegenauigkeit des MLPs erreicht werden [2]. Diese Normalisierung wird jedoch von anderen Quellen für unnötig empfunden, da die Bilddaten schon in dem Skalenbereich von 0 bis 255 normalisiert sind [6]. Das Verfahren wurde trotz Zweifel angewandt, da die Normalisierung einfach zu realisieren war und es keine Hinweise auf Nachteile dieses Verfahrens gab.

3.4.4 Nullzentrierung

Nach [6] ist die Nullzentrierung des Trainingsdatensets, vor allem bei Bilddaten, eine sehr effektive Methode, um die Trainingsleistung eines KNNs zu erhöhen. Dabei werden alle Frames des Trainingsdatensets, durch dessen Mittelwert, auf null gemittelt. Die einzelnen Matrizen des Datensets wurden zu einer Gesamtmatrix addiert, anschließend wird jeder einzelne Wert der Gesamtmatrix durch die Anzahl der im Datenset enthaltenen Matrizen dividiert. Diese Operation führte zu einem gemittelten Bild des gesamten Datensets. Um nun das Datenset auf null zu zentrieren, wurde dieses mittlere Bild von allen Bildern des Datensets subtrahiert. Abbildung 3.6 soll dies bildlich veranschaulichen. Jedoch stimmt das nullzentrierte Bild nicht ganz mit der Realität überein, da sich keine negativen Pixel darstellen lassen.

Abbildung 3.6. Nullzentrierung.

3.5 Architektur und Training des MLPs

In diesem Unterkapitel wird auf die Schritte eingegangen welche das Training des MLP möglich machten. Um ein MLP durch Software zu realisieren mussten verschiedene mit ROS kompatible Programmbibliotheken in das Framework eingebunden werden. Diese werden in dem Abschnitt *Aufbau der Programmierumgebung* genannt. Zudem musste eine effektive Netzarchitektur gefunden werden, worauf in *Festlegen der Netzarchitektur* eingegangen wird. Zuletzt wird die Funktion des Python-Codes beschrieben, welcher das Training des MLP ermöglicht.

3.5.1 Aufbau der Programmierumgebung

Tensorflow ist eine Open Source Programmierbibliothek, speziell entwickelt für Anwendungen des maschinellen Lernens. Entwickelt wurde diese von dem Google Forschungsteam Google Brain. In dieser Arbeit wurde dabei die Python API von Tensorflow genutzt. Durch die NVIDIA-CUDA Erweiterung von Tensorflow können Berechnungen durch GPUs hoch parallelisiert werden und mit der zusätzlichen Programmierbibliothek NVIDIA-cuDNN wird es ermöglicht auch Künstliche Neuronale Netze GPU-beschleunigt zu verarbeiten. Durch diese GPU-Beschleunigung kann die benötigte Trainingszeit eines KNNs stark reduziert werden. Folgende Versionen der genannten Software wurden in das ROS-Framework integriert, um dadurch ein MLP zu erzeugen, welches es erlaubt das Fahrzeug autonom zu steuern.

- Tensorflow 0.10.0
- Python 2.7
- Cuda 7.5
- cuDNN 4.0

3.5.2 Festlegen der Netzarchitektur

In Abbildung 3.7 ist das MLP dargestellt, welches zur Voraussage der Lenkwinkel verwendet wurde. Der Input-Layer des MLP besteht aus 44800 Neuronen. Für jedes Pixel aus dem Frame (Kapitel 3.4.1) steht somit ein Neuron zur Verfügung. Anschließend folgt ein Hidden-Layer mit 32 Neuronen. Der Output-Layer enthält drei Neuronen. Dies sind die Labels, welche für die drei möglichen Lenkwinkel des Fahrzeugs stehen. Die Wahl fiel auf diese Architektur, da diese auch in ähnlichen Projekten [1, 2] genutzt wurde und gute Ergebnisse damit erzielt wurden.

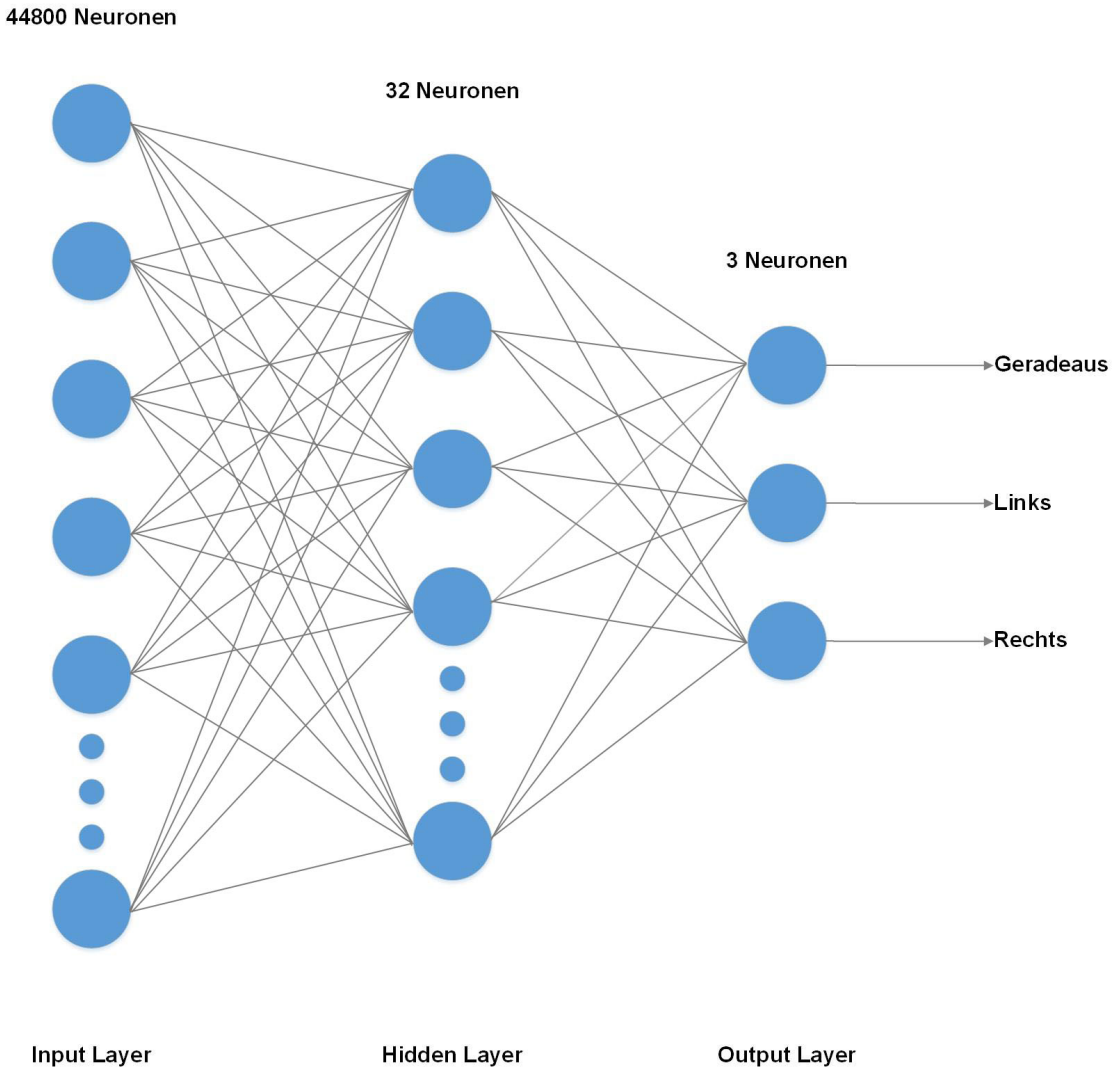


Abbildung 3.7. MLP Architektur.

3.5.3 Beschreibung des Trainings-Codes

Das Training des MLP wird mit dem eigens programmierten Python-Skript *NN_train_HSKA* ausgeführt, welches mit der Tensorflow Programmbibliothek arbeitet. Dieses Skript liest die Trainingsdaten, welche von der *collect_training_data* Node (Kapitel 3.3.3) generiert wurden, ein. Anschließend bereitet es die Trainingsdaten nach den genannten Aufbereitungsmethoden (Kapitel 3.4) auf. Diese Trainingsdaten werden nun in kleinen Batches (Kapitel 3.2.7) dem MLP zugeführt und es findet eine Optimierung des MLPs auf diesen Datensatz statt. Ist das Training des MLP fertiggestellt, werden die optimierten Parameter in einer speziellen Datei gespeichert. Das Training des MLP fand auf einem hochschuleigenen PC stattt, welcher mit einer NVIDIA GeForce TITAN X Grafikkarte ausgestattet war. Die Trainingszeit betrug ungefähr 30 Minuten, bei rund 9800 Frames mit zugehörigem Fahrbefehl.

3.6 Die autonome Steuerung

Damit das Fahrzeug mit dem eintrainierten MLP gesteuert werden kann, wurde die Node *NN_use_HSKA* geschrieben. Diese ermöglicht es, Kameradaten in die entsprechenden Fahrbefehle des Fahrzeugs umzuwandeln. Zudem wurde die vorhandene *motor_controller* Node (Kapitel 2.4.2) erweitert, um die von dem MLP erzeugten Fahrbefehle in Steuerungssignale umwandeln zu können. Die Funktion dieser beiden Nodes, wird in diesem Abschnitt beschrieben.

3.6.1 Die *NN_use_HSKA* Node

Die *NN_use_HSKA* Node wandelt die Frames, generiert aus dem Kamerasensor, in Fahrmanöver des Fahrzeugs um. Um dies zu ermöglichen greift die Node auf die Topic */camera/image/compressed* zu. Anschließend werden die in Kapitel 3.4 erwähnten Methoden der Datenaufbereitung auf diese Frames angewandt. Die Parameter werden aus der durch die *NN_train_HSKA* Node erzeugten Datei geladen und das MLP wird mit diesen initialisiert. Jeder einkommende und anschließend aufbereitete Frame wird durch das MLP propagiert. Dieses erzeugt daraus eine Prognose für das aktuell beste Fahrmanöver. Diese Prognose wird auf eine Topic namens */drive_prediction* gesendet.

3.6.2 Erweiterung der *motor_controller* Node

Die *motor_controller* Node, zur Steuerung des manuellen Fahrzeugs, wurde mit neuen Funktionen erweitert. Es wurde zum einen ein ROS-Subscriber integriert, der die Fahrmanöver aus der Topic */drive_prediction* ausliest. Außerdem wurde ein Auslesen des Start- und Kreistasters des PS3-Controllers eingerichtet (siehe Abbildung 3.8). Mit dem Betätigen des Start-Tasters, können Fahrmanöver aus der */drive_prediction* Topic in PWM-Signale umgewandelt werden, welche es ermöglichen das Fahrzeug autonom fahren zu lassen. Um das Fahrzeug vor Beschädigungen zu schützen, kann die Fahrt dabei jederzeit mit dem Kreis-Taster unterbrochen werden.

Abbildung 3.8. PS3-Controller.

4 Einführung in die Objekterkennung in Bildern

Für Menschen scheint es einfach zu sein, Objekte in einem Bild zu identifizieren, jedoch dauerte die Entwicklung des menschlichen Sehapparates über 540 Millionen Jahre und bis heute kann nicht genau gesagt werden, wie Bildinformationen im Gehirn verarbeitet werden. Computer können Bildinformationen nur in einer Darstellung aus Matrizen mit Pixelwerten verarbeiten. Um in diesen Matrizen Objekte zu erkennen und die Position derer im Bild abzuleiten, werden diese durch eine Kombination verschiedener komplexer Verfahren aufbereitet und ausgewertet. Um eine Verkehrszeichenerkennung in Bilddaten zu ermöglichen, müssen interessante Objekte im Bild erst gefunden (lokalisiert) und anschließend einer Klasse zugeordnet (klassifiziert) werden. Dieses Kapitel soll dazu dienen, ein Grundverständnis über die heutzutage genutzten Verfahren der visuellen Objekterkennung zu bekommen. Auch soll es als Einleitung für die in den nächsten beiden Kapiteln beschriebenen und für die Verkehrszeichenerkennung genutzten Objektdetektionssysteme dienen. Dafür wird zuerst der Begriff der Klasse definiert. Darauf aufbauend wird die Arbeitsweise eines typischen Bildklassifikators beschrieben, wie auch die Probleme welche eine Klassifikation eines Bildes erschweren. Anschließend wird auf die Objektlokalisation in Bildern eingegangen. Es folgt das darauf aufbauende Gebiet der Objektdetektion. Der letzte Abschnitt behandelt den Stand der Technik der Objekterkennung in Bildern.

4.1 Definition der Klasse in der Objekterkennung

Der Begriff Klasse ist relativ abstrakt. Es gibt mehrere Ebenen der Klassenspezifikation. Allgemein wird eine Unterscheidung auf zwei Ebenen getroffen:

Instanzebene: Es erfolgt eine Unterscheidung zwischen spezifischen Objekten. (Bei der Verkehrszeichenerkennung müssen verschiedene Geschwindigkeitsschilder auseinander gehalten werden)

Kategorieebene: Es erfolgt eine Unterscheidung zwischen allgmeinen Zugehörigkeiten von Objekten. (Bei der Verkehrszeichenerkennung müssen Geschwindigkeits- und Vorfahrtsschilder auseinandergehalten werden)

4.2 Aufbau eines Bildklassifikators

Zur Klassifikationssystem braucht es einen sogenannten Klassifikator, welcher ein gegebenes Bild oder einen Bildausschnitt auf die Beinhaltung bestimmter Objektklassen überprüfen kann. Es gibt verschiedene Arten von Klassifikatoren. Um die Dinge zu vereinfachen wird nur auf den binären Klassifikator eingegangen. Dieser kann angewandt auf ein Bild, eine Aussage darüber treffen, ob auf diesem eine einzige bestimmte Objektklasse enthalten ist. Standard Bildklassifikatoren arbeiten häufig nach der in Abbildung 4.1 gezeigten Verarbeitungskette.

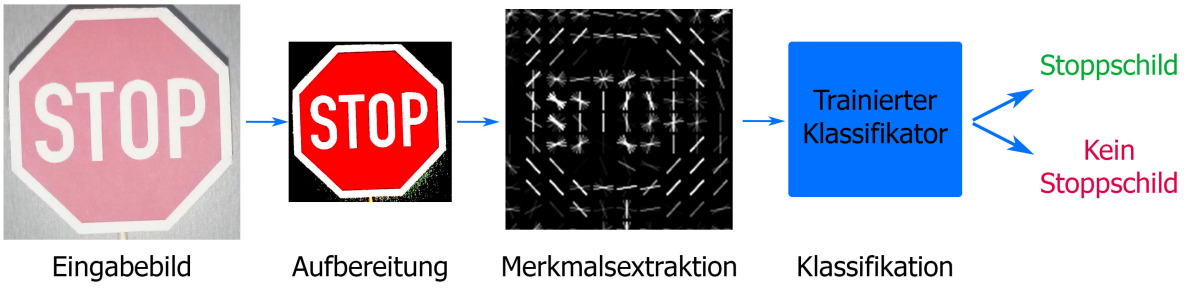


Abbildung 4.1. Bildklassifikation.

Im ersten Schritt werden die Bilder vorverarbeitet. Eine Anpassung der Kontrastwerte oder der Helligkeitseffekte ist möglich. Zudem werden häufig Methoden der Datenaufbereitung angewandt (Kapitel 3.4). Oft erfolgt auch eine Skalierung und ein Zuschnitt der Bilder, da die im nächsten Schritt angewandte Merkmalsextraktion meist mit einer festen Bildgröße arbeitet.

Um überflüssige Informationen aus dem Bild zu entfernen, werden nur die wichtigsten Merkmale aus dem Bild extrahiert. Solche Merkmale können sehr komplex sein. Einfache Merkmale sind Ecken oder Kanten, welche durch Farbgradienten im Bild gefunden werden können. Auch Bildregionen, welche sich durch ihre Helligkeit oder Farbe von anderen Regionen abheben, können Merkmale sein. Häufig genutzte Merkmale sind Haar-ähnliche Merkmale, auf die in Kapitel 5 näher eingegangen wird. Zudem werden oft Algorithmen wie HOG [9], SIFT [10] und SURF [11] genutzt. Diese extrahieren bestimmte Bildmerkmale und beschreiben diese in einem speziellen Format durch Merkmalsvektoren.

Im darauffolgenden Schritt werden die Merkmalsvektoren, durch einen Klassifikationsalgorithmus, einer Klasse zugeordnet. Damit dieser eine Klasse richtig einordnen kann, muss er mit maschinellen Lernalgorithmen trainiert werden. Durch ein überwachtes Lernen mit Trainingsbildern, kann der Klassifikationsalgorithmus seine Parameter auf die Klassifizierung eines bestimmten Objekts einstellen. Das allgemeine Prinzip des Lernens ist es, die Merkmalsvektoren als Punkte in einem hochdimensionalen Raum zu betrachten und Ebenen zu finden welche den Raum so aufteilen, dass nur Beispiele derselben Objektklasse von denselben Ebenen eingeschlossen werden. Ein sehr häufig in der Objekterkennung genutzter Klassifikator ist die Support Vector Machine (SVM) [12], welche nach diesem allgemeinen Prinzip arbeitet. Aber auch Künstliche Neuronale Netze können Bildklassifikatoren sein. Sie folgen einem anderem Ansatz der Bildklassifikation, indem sie die Merkmalsextraktion und Klassifikation in einem System vereinen, dies geschieht durch das Training. Beim Training werden die Gewichte des KNNs auf das zu klassifizierende Objekt angepasst.

4.3 Probleme der Bildklassifizierung

Eine Erkennung von Objekten in einem Bild durch trainierte Klassifikationsalgorithmen ist eine sehr schwierige Aufgabe. Die Genauigkeit der Klassifikation kann durch folgende Faktoren beeinflusst werden:

Verdeckung: Objekte können sich gegenseitig verdecken.

Hintergrund: Andere Objekte und Bildelemente erschweren die Erkennung.

Intraklassenabstand: Objekte können in unterschiedlichen Erscheinungen (Rotation, Skalierung, Perspektive, Farbe) dargestellt sein.

Interklassendistanz: Ähnliche Objekte können schwer voneinander zu unterscheiden sein.

Die Klassifikationsalgorithmen müssen mit einer sehr großen Anzahl von Trainingsbildern trainiert werden, welche all diese Probleme beinhalten, um korrekt klassifizieren zu können.

4.4 Objektlokalisation

Bei der Objektlokalisation soll zusätzlich zur Bildklassifikation, ein Begrenzungsrahmen um ein gesuchtes Objekt gezogen werden, welcher die Position und Größe dieses Objekts im Bild beschreibt. Die in dieser Arbeit zur Verkehrzeichenerkennung genutzten Objektdetektionssysteme, nutzen unterschiedliche Verfahren zur Objektlokalisation, welche im Folgendem vorgestellt werden.

4.4.1 Objektlokalisation durch Sliding-Window Klassifikation

Bei diesem Verfahren wird der Bildklassifikator auf Bildausschnitte, sogenannte Fenster, des Gesamtbildes angewandt. Für die aktuelle Position wird eine Klassifikationsentscheidung getroffen und dadurch geprüft, ob der Bildausschnitt das Objekt enthält. Das Fenster wird Pixel für Pixel über das Gesamtbild geschoben, bis alle Stellen überprüft wurden. Dies ermöglicht die Lokalisierung des gesuchten Objekts im Bild.

4.4.2 Objektlokalisation durch Künstliche Neuronale Netze

Mit bestimmten Architekturen von KNNs ist es möglich, Begrenzungsrahmen für ein im Bild enthaltenes Objekt zu schätzen. Das KNN muss dafür, anstatt eine Klasse zu schätzen, mehrere Zahlenwerte ausgeben, welche den Begrenzungsrahmen im Bild beschreiben. Um eine gute Schätzung des Begrenzungsrahmens zu ermöglichen, muss das KNN mit annotierten Trainingsbildern trainiert werden. Für das im Trainingsbild zu lokalisierende Objekt stehen Ground Truth Daten zu Verfügung, welche die Position und Größe des Objekts im Bild definieren. Beim Training werden Zahlen geschätzt und mit den Ground Truth Daten verglichen (Verlustfunktion könnte Euklidischer Abstand sein), die Gewichte des KNN werden auf die Ground Truth Daten der Bilder angepasst. Nach dem Training ist es dem KNN möglich, Begrenzungsrahmen für unbekannte Bilder zu schätzen. Abbildung 4.2 veranschaulicht die Schätzung eines Begrenzungsrahmens während des Trainings.

Abbildung 4.2. Schätzung eines Begrenzungsrahmens.

4.5 Objektdetektion

Die Objektdetektion ist eine Erweiterung der Objektlokalisation. Bei der Objektdetektion sollen verschiedene Objekte in einem Bild erkannt und mit Begrenzungsrahmen versehen werden. Da in dieser Arbeit mehrere Verkehrszeichen in einem Bild erkannt werden sollten, bestand die Lösung darin Objektdetektionssysteme zu verwenden. Die zwei genutzten Objektdetektionssysteme erweitern die in Kapitel 4.4 vorgestellten Verfahren. In den Kapiteln 5 und 6 wird näher auf die Objektdetektion eingegangen.

4.6 Stand der Technik

Ein Faltendes Neuronales Netz, im englischem Convolutional Neural Network (CNN), ist eine Form eines Künstlichen Neuronalen Netzes. CNNs gelten heutzutage als State of the Art Methode im Bereich der visuellen Objekterkennung. Beim jährlichen ILSVRC-Wettbewerb der Benchmark-Datenbank ImageNet wird die Objekterkennungsqualität von Algorithmen beurteilt. ImageNet wurde vom Stanford Vision Lab der Stanford Universität ins Leben gerufen. Das Stanford Vision Lab ist ein Vorreiter in der Erforschung des menschlichen und maschinellen Sehens und ist zurzeit sehr an der Erforschung von Deep Neural Networks interessiert.

2012 verbesserte ein CNN namens AlexNet [13] den vormaligen Rekord des ILSVRC-Wettbewerbs. Die Fehlerrate im Bereich der Bildklassifikation wurde von 25.8% auf 15.3% herabgesetzt. Das Testdatenset bestand aus 100000 Bildern, wobei zwischen 1000 verschiedenen Klassen unterschieden werden musste. Für eine korrekte Klassifizierung musste die richtige Klasse eine aus fünf vorhergesagten Klassen sein (top-5 error). Seit dieser Zeit nutzen alle vorne platzierten Algorithmen CNN-Architekturen.

Auch die großen Firmen wie Google und Microsoft nehmen an diesem Wettbewerb teil und bieten Frameworks wie Tensorflow (Google) und CNTK (Microsoft) an, welche unterstützend

bei der Programmierung und dem Training von KNNs wirken. Im Jahre 2014 gewann Google mit dem GoogLeNET [14], einem 22 Lagen tiefen CNN, den ILSVRC-Wettbewerb im Bereich der Bildklassifikation, mit einer Fehlerrate von 6.7%. Im Jahr darauf gewann Microsoft, mit einer aus 152 Lagen bestehenden CNN-Architektur namens ResNET [15] und einer Fehlerrate von 3.57%. Microsoft bewies, dass mit mehr Lagen eine noch geringere Fehlerrate erreicht werden kann, wobei mehr Lagen auch mehr Rechenleistung und Trainingszeit erfordern.

KNNs lassen sich sehr gut parallel berechnen, wodurch viel Trainingszeit eingespart werden kann. Deshalb bieten Firmen wie NVIDIA immer mehr neue und bessere Hardware- und Softwarelösungen an. NVIDIA ist auf die parallele Berechnungen durch grafische Recheneinheiten spezialisiert. Die NVIDIA CUDA Deep Neural Network Programmbibliothek cuDNN, ermöglicht eine parallele Berechnung von KNNs in Kombination mit Hochleistungs-Grafikkarten wie zum Beispiel der NVIDIA Titan X. Auch werden von NVIDIA Embedded-Lösungen wie der NVIDIA Jetson TX1 angeboten. Dieser ist auf die Bildverarbeitung ausgelegt und besitzt genug Rechenleistung, um Objekterkennungssysteme basierend auf CNNs in Echtzeit auszuführen.

Algorithmen-Benchmarks, die nur auf Verkehrszeichen ausgelegt sind, sind schwer zu finden. 2011 wurde der GTSRB-Wettbewerb von der Ruhr-Universität Bochum ausgetragen. In diesem Wettbewerb wurden Algorithmen, im Bereich der Bildklassifikation, bewertet. Bei diesem Wettbewerb mussten 12630 Testbilder klassifiziert werden, welche eine aus 42 verschiedene Klassen von Verkehrszeichen enthielten. Auch hier gewann eine CNN-Architektur mit 99.46% richtig klassifizierten Schildern, was sogar die menschliche Performance übertraf. Dies lag womöglich an der großen Anzahl von zu klassifizierenden Bildern.

Die Ruhr-Universität Bochum veranstaltete 2013 auch einen Objektdetektions-Wettbewerb. Bei dem GTSDB-Wettbewerb [16] mussten Verkehrszeichen, in 300 gegebenen Testbildern, lokalisiert und klassifiziert werden. Der beste Algorithmus mit dem Namen HIT501 [16] war keine CNN-Architektur. Kurz beschrieben basiert HIT501 auf der Sliding-Window Klassifikation. Für jeden Bildausschnitt werden Merkmale mit dem HOG-Algorithmus extrahiert. Die Merkmale werden anschließend durch binäre SVM-Klassifikatoren klassifiziert.

Der von ImageNet 2016 ausgetragene ILSVRC-Wettbewerb, brachte im Bereich der Objektdetektion, einen Gewinner mit einer CNN-Architektur hervor. Die top-platzierten Algorithmen sind Objektdetektionssysteme. Sie kombinieren häufig Region Proposal Methoden wie Selective Search mit CNN-Strukturen wie GoogLeNET, ResNET oder einer Kombination aus vieler dieser Architekturen. Zudem nutzen sie Klassifikatoren wie SVM, welche die von den CNNs extrahierten Merkmale klassifizieren. Die Begrenzungsrahmen werden auch durch die extrahierten Merkmalen des CNNs geschätzt. State of the Art Objektdetektionssysteme sind Faster-RCNN [17], der Single Shot MultiBox Detector [18] und das You Only Look Once Objektdetektionssystem [19], welche in den Papers der top-platzierten Teilnehmern des ILSVRC-Wettbewerbs häufig genannt werden.

5 Verkehrszeichenerkennung mittels des OpenCV Haar-Kaskaden-Klassifikators

Bei der Suche nach echtzeitfähigen Objektdetektionssalgorithmen stößt man schnell auf den Haar-Kaskaden-Klassifikator. Der Grund dafür ist, das die weit verbreitete Programmbibliothek OpenCV bereits Werkzeuge besitzt, um einen solchen Haar-Kaskaden-Klassifikator auf beliebige Objekte zu trainieren und eine Objektdetektion in Bildern zu ermöglichen. Das Verfahren wurde 2001 von Paul Viola und Michael Jones [20] erstmals vorgestellt. Dabei handelt es sich um einen auf maschinellem Lernen basierenden Objektdetektionssalgorithmus, bei dem eine Kaskade aus Klassifikatoren basierend auf Haar-ähnlichen Merkmalen durch eine große Anzahl von Trainingsbildern trainiert wird. Nach dem Training ist es dem Haar-Kaskaden-Klassifikator möglich, eintrainierte Objekte in unbekannten Bildern zu detektieren. Diese Methode der Objektdetektion wird auch in einigen anderen betrachteten Projekten genutzt, welche einem autonomen Fahrzeug eine Verkehrszeichenerkennung ermöglichen wollen [1, 2]. Die Wahl fiel auf diesen Objektdetektionssalgorithmus, da dieser einfach durch OpenCV in die Arbeit integriert werden konnte und in den ähnlichen Projekten gute Ergebnisse mit diesem Verfahren erzielt wurden. Im ersten Abschnitt dieses Kapitels, wird auf die Funktion dieses Objektdetektionssalgorithmus eingegangen. Anschließend folgt eine Anleitung zum Trainieren eines Haar-Kaskaden-Klassifikators auf die Detektion eines beliebigen Verkehrszeichens. Abschließend wird beschrieben, wie eine Verkehrszeichenerkennung mittels Haar-Kaskaden-Klassifikatoren in ROS realisiert wurde und wie der Programmcode erweitert werden musste, um eine Reaktion des Fahrzeugs auf Verkehrszeichen zu ermöglichen.

5.1 Architektur und Funktion des Haar-Kaskaden-Klassifikators

Der in OpenCV integrierte Haar-Kaskaden-Klassifikator kombiniert verschiedene Algorithmen und Erkenntnisse zu einem Objektdetektionssystem und stellt eine verbesserte Version des von Viola und Jones vorgestellten Verfahrens dar. [20] ermöglicht eine schnelle Verarbeitung von Bildern, mit einer hohen Detektionsrate. Die Objektdetektion wird durch die Auswertung von formbasierten Merkmalen realisiert. Die Extraktion dieser Merkmale wird mit sogenannten Haar-ähnlichen Merkmalen ermöglicht. Dabei führt dieses Verfahren seine Berechnungen auf Bilder im Graustufenformat aus. Das System besteht aus drei Hauptmethoden, welche eine Echtzeitanwendung der Objektdetektion mittels Haar-ähnlichen Merkmalen ermöglichen. Zum einen ist dies eine neue Bilddarstellung namens Integralbild. Diese ermöglicht es, die in diesem Verfahren genutzten Haar-ähnlichen-Merkmale sehr schnell auszuwerten. Mit der zweiten auf maschinellem Lernen basierenden Methode genannt Adaboost, werden durch Training effiziente Haar-ähnliche Merkmale ausgewählt und zu einem starken Klassifikator zusammengeführt. Die dritte ist eine Methode, um diese Klassifikatoren in einer Kaskadenstruktur anzureihen und die Geschwindigkeit der Objektdetektion noch einmal zu erhöhen. Das Verfahren arbeitet nach der in Kapitel 4.2 vorgestellten Verarbeitungskette. Auf die einzelnen Komponenten und Methoden des Haar-Kaskaden-Klassifikators wird nun näher eingegangen.

5.1.1 Merkmalsextraktion durch Haar-ähnliche Merkmale

Mit Haar-ähnlichen Merkmalen können bestimmte formbasierte Merkmale aus einem Bild extrahiert werden. Erstmals wurde diese Methode 1998 von Papageorgiou et al. [21] beschrieben, um Personen, Gesichter und andere Objekte in statischen Bildern zu erkennen. Ein Haar-ähnliches Merkmal stellt den durchschnittlichen Grauwert-Unterschied von Teilregionen eines Bilds dar. Haar-ähnliche Merkmale sind skalierbare rechteckige Kernel mit schwarzen und weißen Bereichen. In der Abbildung 5.1 sind einige dieser in OpenCV enthaltenen Merkmale dargestellt.

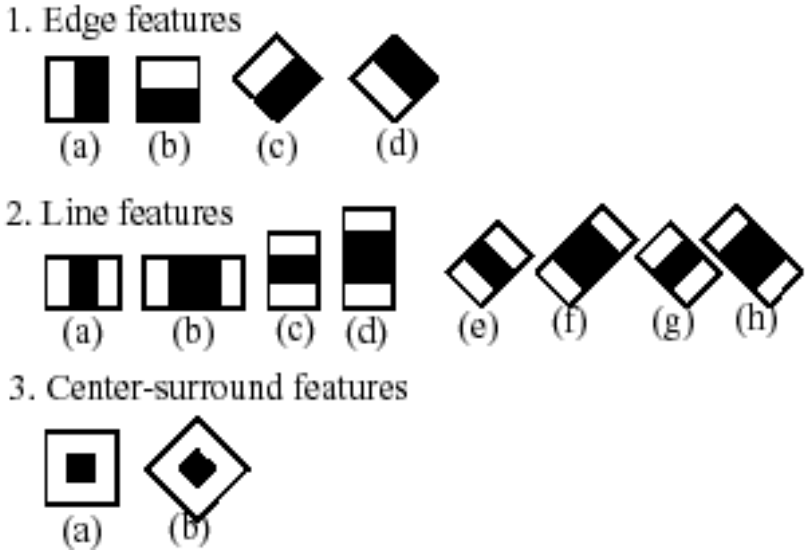


Abbildung 5.1. Haar-ähnliche Merkmale aus OpenCV. [22]

Um Merkmale aus Bildern mit diesen Kerneln zu extrahieren, wird an jeder Stelle eines zu überprüfenden Bildes die Summe der Pixel unter dem weißen Bereich von der Summe der Pixel unter dem Schwarzen Bereich subtrahiert. Überschreitet diese Differenz einen durch maschinellem Lernen ermittelten Schwellwert, lässt sich sagen, dass das Haar-ähnliche Merkmal in diesem Bereich des Bildes enthalten ist.

Als Beispiel kann das Erkennen von Gesichtsmerkmalen durch Haar-ähnliche Merkmale in Abbildung 5.2 dienen. Die Auswertung des linken Haar-ähnlichen Merkmals, wird wahrscheinlich eine hohe Differenz zwischen schwarz und weiß ergeben und somit den Schwellwert überschreiten. Dies ist darauf zurückzuführen, dass die Augenpartien meist dunkler sind als die Partien der Nase oder Backen. Das rechte Merkmal passt auch auf seine aktuelle Position, da der Nasenrücken in Bildern meist heller ist als die Regionen der Augen.

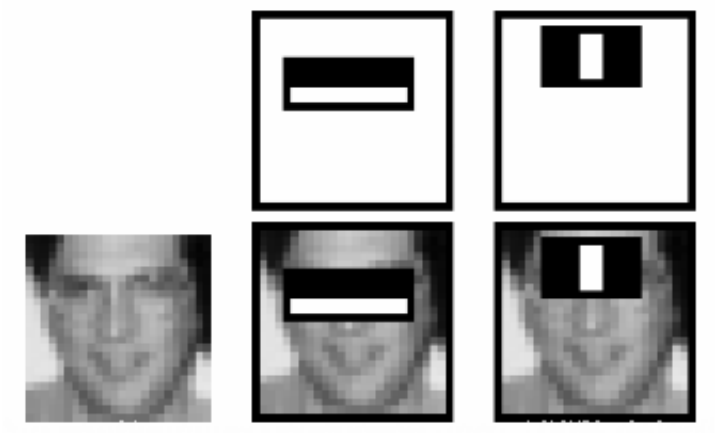


Abbildung 5.2. Merkmalsextraktion durch Haar-ähnliche Merkmale. [20]

Es sei erwähnt, dass vielzählige Haar-ähnlichen Merkmale in dem OpenCV Haar-Kaskaden-Klassifikator enthalten sind, welche auch komplexere Auswertungen erfordern wodurch aber bessere Detektionsergebnisse geliefert werden können.

Der OpenCV Haar-Kaskaden-Klassifikator verwendet das in Kapitel 4.4.1 vorgestellte Sliding-Window Verfahren zur Objektdetektion. Das Gesamtbild wird durch ein sich verschiebendes Fenster in Bildausschnitte aufgeteilt, welche einzeln auf die Haar-ähnlichen Merkmale getestet werden. Die Haar-ähnlichen Merkmale werden in verschiedenen Skalierungen und auf alle Positionen der Bildausschnitte angewandt. Allein in einem 24×24 Pixel großen Bildausschnitt müssen dadurch über 180000 Haar-ähnliche Merkmale berechnet werden. Für jedes dieser Haar-ähnlichen Merkmale, muss die Summe der Pixel unter dem weißen und schwarzen Bereich berechnet werden um deren Differenz bilden zu können. Dies führt zu einem sehr hohen Rechenaufwand. Um den Rechenaufwand bei der Summenbildung zu reduzieren, wird ein sogenanntes Integralbild verwendet, welches auf dem Konzept der Summed Area Tables [23] basiert. In jedem Punkt (x, y) des Integralbilds steht dabei die Summe $I_{\sum}$ der Pixel innerhalb des Rechtecks eingeschlossen durch die Punkte (0,0), (x,0), $(0, y)$, (x, y).

$$I_{\sum}(x, y) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j) \tag{5.1}$$

Zur Veranschaulichung ist in Abbildung 5.3 ein Originalbild (links) und das durch Anwendung der Formel (5.2) auf jede Position (x, y) des Originalbildes entstehende Integralbild (rechts) dargestellt.

1	3	6	1	1	3	1
1	4	4	1	3	5	2
0	4	5	1	1	3	2
0	7	2	1	0	1	2
0	2	1	1	0	1	1
0	1	0	1	1	1	1
0	9	0	0	0	0	1

→

1	4	10	11	12	15	16
2	9	19	21	25	33	36
2	13	28	31	36	47	52
2	20	37	41	46	58	65
2	22	40	45	50	63	71
2	23	41	47	53	67	76
2	32	50	56	62	76	86

Abbildung 5.3. Originalbild und Integralbild. [24]

Die einzelnen Positionen im Integralbild repräsentieren die Summe aller Pixel über und links von dem Pixel des Orginalsbildes. Durch diese geänderte Darstellung ist es möglich, die Pixelsumme einer beliebigen rechteckigen Fläche des Ursprungsbildes, aus nur vier Werten des Integralbildes, mit folgender Formel zu berechnen:

$$I_{\sum}(Fläche_D) = I_{\sum}(P_1) + I_{\sum}(P_4) - I_{\sum}(P_2) - I_{\sum}(P_3) \tag{5.2}$$

Die durch die vier Punkte $P1$, $P2$, $P3$ und $P4$ eingeschlossene Fläche D in Abbildung 5.4 berechnet sich gemäß (5.2).

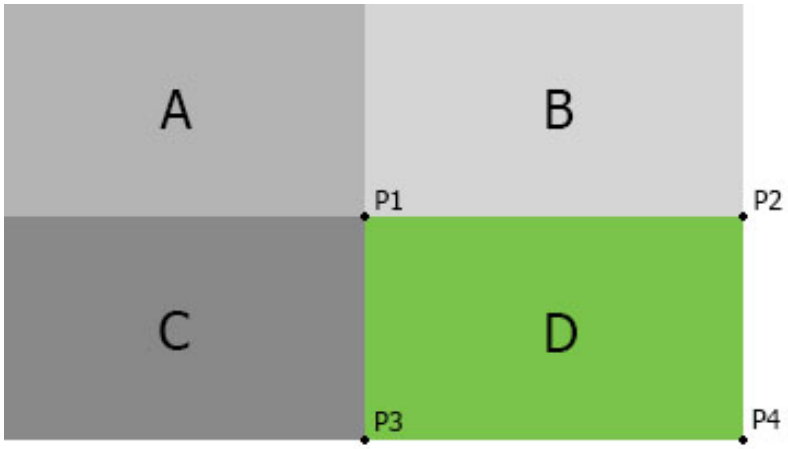


Abbildung 5.4. Summenberechnung im Integralbild.

Die Haar-ähnlichen Merkmale können somit schnell aus einem Bild extrahiert und ausgewertet werden. Verschiedene Objekte besitzen unterschiedliche Haar-ähnliche Merkmale. Mit der Übereinstimmung vieler objektspezifischer Merkmale lässt sich ein Objekt klassifizieren. In der Verarbeitungskette aus Kapitel 4.2 entspricht die Auswertung eines Haar-ähnlichen Merkmals dem Schritt der Merkmalsextraktion.

5.1.2 Erzeugen eines starken Klassifikators aus Haar-ähnlichen Merkmalen mittels Adaboost

Adaboost (Adaptive Boosting) [25] ist eine spezielle Variante des Boosting Algorithmus, welcher durch maschinelles Lernen viele schwache Klassifikatoren, zu einem starken Klassifikator kombiniert. Wie in Abschnitt 5.2 beschrieben, können durch die Merkmalsextraktion mit Haar-ähnlichen Merkmalen, mehr als 180000 Merkmale in einem 24×24 Pixel großen Bildausschnitt berechnet werden. Dabei sind jedoch sehr viele dieser Haar-ähnlichen Merkmale bei der Klassifikation eines bestimmten Objekts irrelevant, wodurch die Berechnungszeit unnötig erhöht wird. Der Adaboost Algorithmus wird in OpenCV zum Erzeugen eines effizienten Klassifikators genutzt, welcher Objekte durch Auswertung Haar-ähnlicher Merkmale klassifiziert. Er übernimmt die Aufgabe, die nützlichsten objektspezifischen Haar-ähnlichen Merkmale für das Detektieren eines Objekts auszuwählen und mit diesen einen starken Klassifikator zu formen. Dies wird durch überwachtes Lernen realisiert. Der Algorithmus arbeitet mit positiven und negativen Trainingsbildern. Ein positives Bild enthält immer das Objekt. Ein negatives Bild kann alles enthalten außer dieses Objekt. Die einzelnen Haar-ähnlichen Merkmale werden beim Training auf die Bilder angewandt. Adaboost weiß dabei schon zuvor, ob das Objekt auf einem Bild enthalten ist oder nicht. Durch das Training werden die Schwellwerte der einzelnen Haar-ähnlichen Merkmale auf das zu klassifizierende Objekt angepasst. Klassifiziert ein Haar-ähnliches Merkmal beim Training mehr als 50% der Trainingsdaten richtig, wird es als schwacher Klassifikator eingestuft. Jeder schwache Klassifikatoren wird zusätzlich mit einem Gewicht multipliziert. Je mehr Objekte der schwache Klassifikator richtig klassifizieren konnte, desto höher fällt er ins Gewicht. Adaboost formt anschließend aus all diesen gewichteten schwachen Klassifikatoren durch Linearkombination einen starken Klassifikator. Durch Auswertung dieses starken Klassifikators auf einen Bildausschnitt und dem Vergleich mit einem durch Training ermittelten Schwellwert, kann dieser Bildausschnitt auf das Enthalten eines Objekts geprüft werden.

5.1.3 Kaskadierung von starken Klassifikatoren

Das Basisprinzp des bisher beschriebenen Objektdetektionssystems ist es, mit dem aus Abschnitt 5.1.2 beschriebenen starken Klassifikator das gesamte Bild mit dem Sliding-Window Verfahren abzusuchen. Doch das Gesamtbild wird meist aus einem großen Bereich bestehen, welcher das gesuchte Objekt gar nicht enthält. Um die Geschwindigkeit des Verfahren zu erhöhen und gleichzeitig die korrekte Klassifizierung zu steigern, wird ein Konzept namens Kaskaden Klassifikation genutzt. Hierbei werden kleinere starke Klassifikatoren in mehreren Stufen effizient hintereinandergeschaltet. In jeder dieser Stufen befindet sich eine gewisse Anzahl an schwachen Klassifikatoren. Die Aufgabe dieser Stufen ist es zu ermitteln, ob in einem Bildausschnitt das gesuchte Objekt enthalten ist. Es wird dabei sofort zum nächsten Bildausschnitt übergegangen, wenn eine dieser Stufen keinen positiven Output liefert. Dies bedeutet gleichzeitig, dass das Objekt im abgesuchten Bildausschnitt nicht vorhanden ist. Die ersten Stufen des Haar-Kaskaden-Klassifikators haben meist eine sehr geringe Anzahl von schwachen Klassifikatoren, um Rechenzeit durch schnelles Auswerfen von negativen Bildausschnitten zu sparen. Nur wenn alle Stufen des Haar-Kaskaden-Klassifikators positiven Output liefern, ist das gesuchte Objekt auch sehr wahrscheinlich im Bildausschnitt enthalten.

5.2 Trainieren eines Haar-Kaskaden-Klassifikators mit OpenCV

Um auf verschiedene Verkehrszeichen bei der autonomen Fahrt reagieren zu können, wurden zur Objektdetektion eigens trainierte Haar-Kaskaden-Klassifikatoren verwendet. Das Training wurde mit einem separaten Programm innerhalb von OpenCV realisiert. Dabei musste für jedes zu erkennende Verkehrszeichen ein eigener Haar-Kaskaden-Klassifikator trainiert werden. Es wurden Haar-Kaskaden-Klassifikatoren für drei verschiedene Verkehrszeichen („Stopp“-, „Vorfahrt gewähren“-, „Tempo-30-Zone“-Schild) trainiert. Da die Methode des Haar-Kaskaden-Klassifikator-Trainings für jedes Verkehrszeichen gleich zu realisieren ist, dient diese Beschreibung auch als Anleitung zum Eintrainieren weiterer Verkehrszeichen. Im Folgenden werden die durchgeführten Schritte von der Vorbereitung, bis zu einem fertig trainierten Haar-Kaskaden-Klassifikator, zur Detektion der Klasse der „Vorfahrt gewähren“-Schilder, erläutert.

5.2.1 Sammeln positiver Bilder

Ein positives Bild enthält immer das zu detektierende Objekt. Um gute Trainingsergebnisse zu erzielen, sollten die positiven Bilder alle möglichen Eigenschaften aufweisen, welche auch bei der anschließenden Objektdetektion mit dem trainierten Haar-Kaskaden-Klassifikator auftreten können. Der „German Traffic Sign Recognition Benchmark“-Datensatz [26] wurde für das Training verwendet. In ihm sind 43 verschiedene Klassen von Verkehrszeichen enthalten. Jede dieser Klassen besitzt zwischen 200 und 2500 Farbbilder (RGB-Farbraum) desselben Verkehrszeichens. Diese sind unterschiedlich beleuchtet, verschieden groß und in verschiedenen Perspektiven dargestellt. Alle Bilder sind im Portable-Pixmap-Format (ppm) gespeichert. Es ist außerdem für jede Verkehrszeichenklasse eine CSV-Datei beigefügt. Diese enthält Metainformationen über jedes einzelne Verkehrszeichenbild. Diese sind der Dateiname, die Höhe und Breite des Bildes, eine Pixelkoordinatenbeschreibung der Position des Verkehrszeichens in dem Bild und eine Klassenindentifikationsnummer.

Für das Verkehrszeichen „Vorfahrt gewähren“standen in diesem Datenset 2161 Bilder zur Verfügung. Die Größe der positiven Bilder variierte zwischen 25×25 und 224×201 Pixeln. Um einen Haar-Kaskaden-Klassifikator durch OpenCV mit diesen positiven Bildern trainieren zu können, wurden die einzelnen Bilder mit dem Python-Skript *convert_sign_imgs.py* in ein Graustufenformat konvertiert und im JPEG-Format abgespeichert. In Abbildung 5.5 sind einige bearbeitete „Vorfahrt gewähren“-Schilder dargestellt, welche für das Training eines Haar-Kaskaden-Klassifikators genutzt wurden.

Abbildung 5.5. Positive Trainingsbilder. [26]

5.2.2 Sammeln negativer Bilder

Negative Bilder sind beliebige Bilder, welche alles enthalten können außer dem zu detektierenden Objekt. Dabei können diese Bilder verschiedene Größen besitzen, sollten aber größer als die positiven Bilder sein. Für das Training wird eine Vielzahl an negativen Bildern benötigt. Es gibt verschiedene Möglichkeiten diese Bilder zu sammeln.

Um schnell an eine große Menge dieser Bilder zu kommen, bietet sich die Bilddatenbank Imagenet an [27]. Imagenet stellt URLs zur Verfügung, welche aus Listen von Links zu Bildern gleicher Kategorie bestehen. Für das Haar-Kaskaden-Klassifikator-Training wurden Kategorien ausgewählt, welche sicher keine Verkehrsschilder enthalten. Mit der Funktion *store_raw_images()* des Python-Skripts *collect_train_images.py* konnten die in der URL enthaltenen Links zu den kategorisierten Bildern einzeln besucht und das enthaltene Bild heruntergeladen werden. Jedes Einzelne Bild wurde zusätzlich in ein 250×250 Pixel großes Graustufenbild konvertiert und im JPEG-Format in einem Ordner gespeichert.

5.2.3 Erzeugen von Beschreibungsdateien

OpenCV benötigt eine Beschreibung jedes einzelnen Bildes, mit welchem der Haar-Kaskaden-Klassifikator trainiert werden soll. Diese Bildbeschreibungen müssen für die negativen und positiven Bilder in zwei verschiedenen Dateien hinterlegt sein. In der negativen Datei muss der Dateipfad jedes einzelnen negativen Bilds hinterlegt werden. Der Dateiinhalt hat wie folgt auszusehen:

<Speicherpfad>/<Bild1.jpg>
<Speicherpfad>/<Bild2.jpg>
...

Erzeugt werden kann solch eine Beschreibungsdatei durch Eingabe dieses Befehls in der Kommandozeile:

find ./<Speicherpfad> -iname „*.jpg“> <Name der Beschreibungsdatei>.txt

Die Beschreibungsdatei der positiven Bilder muss auch den Dateipfad der einzelnen Bilder enthalten. Zudem muss die Anzahl der enthaltenen Objekte und deren Position und Größe im Bild definiert sein. Dies hat wie folgt auszusehen:

<Speicherpfad>/<Bild1.jpg> <Objektanzahl> <x> <y> <w> <h>
<Speicherpfad>/<Bild2.jpg> <Objektanzahl> <x> <y> <w> <h>
...

Die Parameter <x> und <y> stehen für die Koordinate der oberen linken Ecke eines Begrenzungsrahmens, welcher das Objekt im Bild umrahmt. Der Parameter <w> steht für die Begrenzungsrahmenbreite und <h> für die Höhe. Um für die „Vorfahrt gewähren“-Schilder des “German Traffic Sign Recognition Benchmark“-Datensatzes eine Beschreibungsdatei in dem benötigten Format zu erzeugen, wurden mit dem Python-Skript *gen_image_training_DAT.py* die Metainformationen aus der gegebenen CSV-Datei ausgelesen, in das benötigte OpenCV-Format konvertiert und in einer neuen DAT-Datei gespeichert.

5.2.4 Erzeugen einer Vektor-Datei

Das OpenCV Trainingsprogramm benötigt sogenannte positive Samples von den zu detektierenden Objekten und nimmt diese nur in einer speziellen Vektor-Datei an. Diese Vektor-Datei besteht aus einer Aneinanderreihung der ausgeschnittenen positiven Bildobjekte. Mit dem von OpenCV angebotenen separaten Programm *opencv_createsamples* kann eine solche Vektor-Datei erzeugt werden. Dieses Programm schneidet die Objekte aus den positiven Bildern aus, skalierte alle auf eine einheitliche Größe und speichert diese in der Vektor-Datei. Um dies zu ermöglichen, wurden dem Programm folgende Parameter übergeben:

-info	<Speicherort>	Speicherort der DAT-Datei
-num	2161	Anzahl der auszuschneidenden positiven Bilder
-vec	<Name>.vec	Name der zu erzeugenden Vektor-Datei
-w	20	Pixelbreite auf welche die Bilder skaliert werden sollen
-h	20	Pixelhöhe auf welche die Bilder skaliert werden sollen

Für das Erzeugen einer Vektor-Datei aus den „Vorfahrt gewähren“-Schildern, wurde dafür die Beschreibungsdatei (DAT-Datei) der positiven Bilder als Parameter übergeben.

Für den Parameter -num wurde die maximale Anzahl an zur Verfügung stehenden positiven Bildern verwendet. Die Parameter -w und -h legen die Größe der aus den positiven Bildern zu erzeugenden positiven Samples fest. Außerdem ist dies gleichzeitig die Fenstergröße, welche bei der Anwendung des Sliding-Window Verfahren genutzt wird.

5.2.5 Training des Haar-Kaskaden-Klassifikators

Durch das OpenCV Trainingsprogramm *opencv_traincascade* kann ein Haar-Kaskaden-Klassifikator trainiert werden.

Dabei ist es möglich eine Vielzahl an Parametern zu definieren, welche die Qualität des Trainings beeinflussen. In einer empirischen Analyse [28], welche sich mit dem Training von Haar-Kaskaden-Klassifikatoren befasst, wurde eine 20-stufige Kaskade mit einer minimalen Trefferquote von 0.999, sowie einer maximalen Falschen-Alarm-Rate von 0.5 pro Stufe verwendet.

Trefferquote (Richtig-Positiv-Rate): Verhältnis von richtig klassifizierten positiven Bildern zur Gesamtanzahl der positiven Bilder im Trainingsdatenset.

Falsch-Alarm-Rate (Falsch-Positiv-Rate): Verhältnis von falsch klassifizierten negativen Bildern zur Gesamtzahl der negativen Bilder im Trainingsdatenset.

Durch diese Parameter ist nach dem Training eine Trefferquote von $0.999^{20} \approx 0.98$ und eine Falsche-Alarm-Rate von $0.5^{20} \approx 9.6 \cdot 10^{-7}$ zu erwarten. Außerdem wurde eine Fensterbreite von 20×20 Pixeln genutzt und der Gentle Adaboost Algorithmus als effektivster Algorithmus aufgeführt. Diese Parameter wurden für das Training verwendet.

Dem Programm musste zudem der Speicherort der durch *opencv_createsamples* erzeugten Vektor-Datei und die Beschreibungsdatei der negativen Bilder übergeben werden. Da der „German Traffic Sign Recognition Benchmark“-Datensatz 2161 Bilder von „Vorfahrt gewähren “-Schildern enthielt, wurde diese Zahl für die Anzahl der positiven Bilder verwendet. Für den

Parameter -numNeg wurden 1050 Bilder verwendet, da in verschiedenen Quellen ein Verhältnis von 2:1 (POS:NEG) empfohlen wurde. Das Training des Haar-Kaskaden-Klassifikators zur Detektion eines „Vorfahrt gewähren“-Schildes wurde mit folgenden Parametern gestartet:

-vec	<Speicherort>	Speicherort der Vektor-Datei
-bg	<Speicherort>	Speicherort der negativen Beschreibungsdateien
-numPos	2161	Anzahl der positiven Bilder
-numNeg	1050	Anzahl der negativen Bilder
-numStages	20	Anzahl der Kaskaden-Stufen
-w	20	Sliding-Window Fensterbreite
-h	20	Sliding-Window Fensterhöhe
-bt	GAB	Typ des Boosted Algorithmus
-minHitRate	0.999	Minimale Trefferquote für jede Stufe
-maxFalseAlarmRate	0.5	Höchste Falsch-Alarm-Rate für jede Stufe
-mode	ALL	Genutzte Haar-ähnliche Merkmale
-data	<Speicherort>	Speicherort der erzeugten XML-Datei

Nach Fertigstellung des Trainings (Dauer $\approx$ 3h auf i7-4700MQ CPU) wird eine XML-Datei, in dem mit dem Parameter -data definierten Dateipfad, abgespeichert. In dieser XML-Datei sind die durch Adaboost trainierten Kaskaden-Stufen enthalten. Für jede Stufe sind die ausgewählten schwachen Klassifikatoren mit ihren trainierten Gewichten und dem zur richtigen Klassifizierung benötigten Schwellwert definiert.

5.3 Einbinden der trainierten Haar-Kaskaden-Klassifikatoren in ROS

Um verschiedene Verkehrszeichen bei der autonomen Fahrt mittels eigens trainierten Haar-Kaskaden-Klassifikatoren erkennen zu können, wurde die ROS-Node *trafficsign_finder* erzeugt. Zudem wurde die *motor_controller* Node erweitert, um eine Reaktion des Fahrzeugs auf erkannte Verkehrszeichen zu ermöglichen. Die Funktion und Kommunikation dieser beiden Nodes wird in diesem Abschnitt erklärt.

5.3.1 Die *trafficsign_finder* Node

Die *trafficsign_finder* Node realisiert die Verkehrszeichenerkennung bei der autonomen Fahrt des Fahrzeugs. Sie lädt dazu die durch *opencv_traincascade* erzeugten XML-Dateien, welche die Parameter der Haar-Kaskaden-Klassifikatoren enthalten in das Programm. Jede dieser Dateien wird einem Objekt der OpenCV-Klasse *CascadeClassifier* übergeben. Um Verkehrszeichen im Kamerabild des Fahrzeugs erkennen zu können, liest die Node die Topic */camera/image/compressed* (3.1.3) aus. Für jedes erzeugte Objekt und jeden Frame der Kamera, wird die Klassen-Funktion *detectMultiScale* aufgerufen. Die Funktion wendet für jeden Haar-Kaskaden-Klassifikator das Sliding-Window Verfahren auf den gegebenen Frame an. Wurde ein Verkehrszeichen detektiert, gibt die Funktion die Größe und Position des aktuell abgesuchten rechteckigen Fensters im Frame zurück und die Werte werden in einem Array gespeichert. Um auch Objekte die größer als das Suchfenster sind, erkennen zu können, wird das Bild erneut

abgesucht und dabei um einen definierten Faktor verkleinert. Dies wird solange wiederholt, bis das Suchfenster die Bildgröße überschreitet. Der pro Durchlauf zu reduzierende Faktor des Bildes wird der Funktion *detectMultiscale* als Parameter übergeben, welcher *scaleFactor* genannt wird. Oftmals treten bei der Anwendung des Haar-Kaskaden-Klassifikators viele Detektionen in unmittelbarer Nähe zum eigentlichen Objekt auf. Dadurch entstehen viele erkannte Fenster. Diese Fenster können durch *detectMultiscale* zusammengefasst werden, indem der Parameter *minNeighbours* angeben wird. Werden nun mehrere Fenster in einem bestimmten Bereich detektiert, werden diese durch das Aufrufen der Klassen-Funktion *groupRectangles* zu einem mittleren Fenster zusammengeführt. Einzelne falsch-positve Detektionen können so eliminiert werden, da bei diesen nicht genug Detektionen in einem bestimmten Bereich vorkommen. Die analysierten Frames werden nach der beendeten Suche in einem von OpenCV erzeugten Fenster auf dem Laptop dargestellt, wobei in diesem die erkannten Verkehrszeichen durch einen Begrenzungsrahmen farblich markiert sind. In Abbildung 5.6 ist die Detektion der drei eintrainierten Verkehrszeichen dargestellt.

Abbildung 5.6. Verkehrszeichendetektion durch Haar-Kaskaden-Klassifikatoren.

Die Node sendet zudem eine ROS-Message auf die Topic */traffic_sign_finder*. Diese Message enthält ein Array aus Bool-Variablen. Für jedes detektierbare Verkehrszeichen steht eine Bool-Variable zur Verfügung. Solange ein bestimmtes Verkehrszeichen detektiert wird, setzt die Node die Bool-Variable dieses Verkehrszeichen auf der Topic */traffic_sign_finder* auf True.

5.3.2 Erweiterung der *motor_controller* Node

Um auf die erkannten Verkehrszeichen mit einer eindeutigen Reaktion des Fahrzeugs reagieren zu können, musste eine Kommunikation zwischen der *motor_controller* Node und der *trafficsign_finder* Node hergestellt werden. Die *motor_controller* Node greift dafür auf die Topic */traffic_sign_finder* zu. Wird eine Bool-Variable eines Verkehrszeichen auf der Topic */traffic_sign_finder* auf True gesetzt, erkennt dies die *motor_contoller* Node. Je nach erkanntem Verkehrszeichen kann eine Reaktion des Fahrzeugs durch entsprechendes Ändern der Steuerungssignale ermöglicht werden.

6 Verkehrszeichenerkennung mit dem YOLO: Echtzeit Objektdetektionssystem

Da CNN-Architekturen die besten Ergebnisse in den heutigen Objektdetektions-Benchmark-Tests liefern, fiel die Entscheidung darauf, ein CNN-Objektdetektionssystem zur Verkehrszeichenerkennung zu verwenden. Durch die hohe Anzahl an kleinen Berechnungen, die bei der Propagation eines Bildes durch ein CNN erforderlich sind, erfordern auf CNNs basierende Objektdetektionssysteme das Nutzen von GPUs um in Echtzeit Bilder verarbeiten können. Die neusten CNN-Objektdetektionssysteme benötigen sehr viel GPU-Speicherressourcen. Da in dieser Arbeit eine Verkehrszeichendetektion durch einen Laptop ermöglicht werden sollte, musste ein effizientes System mit niedrigen Speicheranforderungen gefunden werden. Das Arbeiten mit YOLO wurde zum Sammeln erster Erfahrungen bei der Objektdetektion mit CNNs, in einer Online-Stanford-Vorlesung [29] über Objekterkennung mit CNNs empfohlen. YOLO steht für „You Only Look Once“und ist ein State of the Art Objektdetektionssystem, basierend auf einer CNN-Architektur mit einem neuen Ansatz der Objektdetektion. Das Verfahren wurde 2015 [19] erstmals vorgestellt. YOLO war das erste auf CNNs basierende Objektdetektionssystem welches Bilder in Echtzeit (bis zu 45 FPS) verarbeiten konnte und ist zudem das bis heute schnellste Verfahren. Die YOLO-Version *YOLOv1* mit der komprimierten Architektur *YOLO-Tiny* kann auf dem Laptop (2000 MB Grafikkartenspeicher) dieser Arbeit genutzt werden, da diese nur ungefähr 611 MB Grafikkartenspeicher benötigt (siehe Kapitel 7.2). Diese kleinere CNN-Architektur macht die Bildverarbeitung schneller (bis zu 155 FPS), jedoch müssen auch Abstriche bei der Detektionsgenauigkeit gemacht werden. Zudem ermöglicht die YOLO-Architektur ein Eintrainieren von mehreren Verkehrszeichen auf einmal, da die gesamte Detektion (Klassifizierung und Lokalisierung mehrerer verschiedener Objekte) durch ein spezielles Training einer einzigen CNN-Architektur erreicht wird, was für CNN-Objektdetektionssysteme nicht üblich ist. Außerdem standen schon Projekte zur Verfügung, welche sich mit dem eintranieren von Verkehrszeichen in YOLO [30] und der Integration von YOLO in ROS [31] befassten. Aus diesen Gründen fand das YOLO-Objektdetektionssystem in dieser Arbeit Verwendung. Der erste Teil dieses Kapitels geht auf die Bildklassifikation mit CNNs ein. Darauf aufbauend wird sich mit der Objektdetektion durch CNNs auseinandergesetzt. Anschließend wird die Funktion des YOLO-Objektdetektionssystems näher erläutert. Nach der Klärung der Funktion wird beschrieben wie die YOLO-Tiny-Architektur auf die Detektion von Verkehrszeichen trainiert werden kann. Abschließend wird die Integration des YOLO-Objektdetektionssystems in das Robot Operating System beschrieben und wie es bei der autonomen Fahrt zur Verkehrszeichendetektion genutzt wird.

6.1 Bildklassifikation durch Convolutional Neural Networks

CNNs sind ähnlich aufgebaut wie Künstliche Neuronale Netze. Sie sind jedoch viel besser dafür geeignet, Bilder zu klassifizieren. Inspiriert wurden CNNs durch den visuellen Kortex des Gehirns. Dieser besitzt Zell-Regionen die auf unterschiedliche Bereiche des Blickfelds reagieren. Durch das Einsetzen einer Mikroelektrode in den primären visuellen Kortex einer Katze, konnten Hubel und Wiesel 1961 beweisen, dass einzelne neuronale Zellen im Gehirn nur auf die Präsenz von Linien und Kanten in spezieller Orientierung reagieren [32]. Manche reagierten zum Beispiel nur auf horizontale, andere auf vertikale Linen oder Kanten. Außerdem fanden sie heraus, dass diese Nervenzellen in einer Säulen-Architektur aufgebaut sind. Untersuchungen deuteten darauf hin, dass diese Informationsverarbeitung in den höheren Schichten fortgeführt wird, so dass Nervenzellen auf komplexe Formen wie etwa Buchstaben und Gesichter reagieren. CNNs ahmen diese Struktur im mathematischen Sinne nach. Sie versuchen bestimmte Charakteristiken, von einem Objekt in einem Bild, durch Faltungen mit speziellen Filtern zu extrahieren und daraus die Objektklasse abzuleiten. Sie sind aus verschiedenen Lagen aufgebaut. Um eine CNN-Architektur zu erschaffen, wird eine Kombination aus drei Hauptlagen verwendet. Dies sind das Convolutional Layer, das Pooling Layer und das Fully-Connected Layer. In den nächsten Unterkapiteln wird näher auf diese speziellen Lagen eingegangen und es wird aufgezeigt, wie durch Nutzen dieser Lagen eine Bildklassifikation ermöglicht werden kann.

6.1.1 Convolutional Layer

Convolutional Layer werden bei der Bildklassifikation zur Merkmalsextraktion (Kapitel 4.2) verwendet. Sie bestehen aus kleinen lernfähigen Filtern mit einem zugehörigen Bias. Die Filter eines Convolutional Layers besitzten alle die gleiche Dimension, enthalten jedoch meist andere Werte. Die Tiefe der Filter ist dabei immer gleich der Tiefe der Eingangsmatrix (Bei einem Eingangsbild im RGB-Farbraum wäre die Tiefe der Filter somit drei). Zur Extraktion von Merkmalen, wird jeder einzelne Filter mit einer bestimmten Schrittweite (stride) über alle Bereiche der Eingangsmatrix geschoben. An jeder Position wird das Skalarprodukt zwischen den Filtern und der aktuellen Position auf der Eingangsmatrix berechnet. Dies entspricht einer Faltungsoperation. Zusätzlich wird noch ein Bias addiert. Durch diese Operation entsteht für jeden Filter eine zweidimensionale Aktivierungskarte, bestehend aus der Reaktion des Filters auf alle räumlichen Positionen der Eingangsmatrix. Die zweidimensionalen Aktivierungskarten werden nach der Berechnung hintereinander gelegt, wodurch eine Ausgangsmatrix entsteht deren Tiefe der Anzahl der Filter im Convolutional Layer entspricht. In Abbildung 6.1 ist solch eine Operation auf eine dreidimensionale Eingangsmatrix mit zwei verschiedenen Filtern dargestellt.

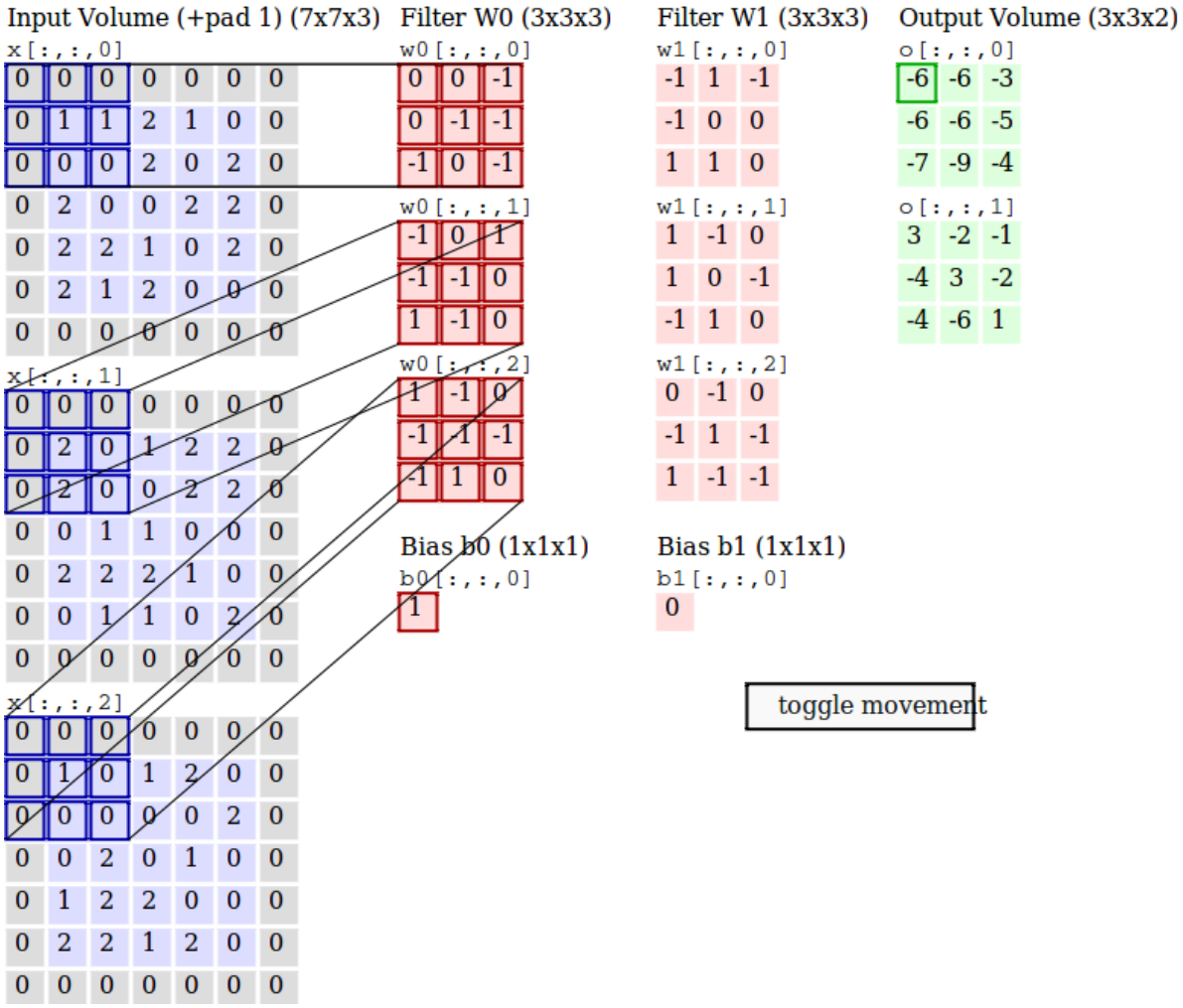


Abbildung 6.1. Arbeitweise des Convolutional Layers. [33]

Die Filter der Convolutional Layer stellen die Gewichte des CNNs dar. Diese können durch Training (Kapitel 3.2) auf die Extraktion von bestimmten Objektmerkmalen angepasst werden. Die erste Lage eines CNN besteht immer aus einem Convolutional Layer und lernt dabei intuitiv sehr reaktionsfreudig auf visuelle Charakteristiken, wie speziell orientierte Ecken oder Farbflecken, zu sein. Zum Beispiel könnte ein Filter durch Training die Eigenschaften eines Sobel-Operators annehmen. In den folgenden Convolutional Layern des CNNs werden diese Charakteristiken immer weiter aufgebaut.

6.1.2 Pooling Layer

Pooling Layer werden häufig in einer CNN-Architektur verwendet, um überflüssige Informationen auszuwerfen. Diese Pooling Layer folgen dabei meist auf ein Convolutional Layer. Durch Pooling kann die Fläche der Ausgangsmatrix eines Convolutional Layers auf wesentliche Merkmale reduziert werden, wodurch Rechenaufwand minimiert werden kann und ein Overfitting der CNN-Architektur verhindert werden soll. Dies ist möglich, da bei der Auswertung von

CNNs weniger die exakte Position eines Merkmals relevant ist, sondern vielmehr die grobe Lage der Merkmale relativ zueinander. MAX-Pooling ist die am häufigsten verwendete Art des Poolings. Das Pooling Layer wendet Max-Pooling auf jede Aktivierungskarte der Eingangsmatrix an. Dabei wird ein Filter, mit einer bestimmten Dimension und einer gewissen Schrittweite (stride), über alle Aktivierungskarten geschoben. An jeder Position zwischen dem Filter und einer Aktivierungskarte wird Max-Pooling angewandt, wodurch der höchste Wert aus diesem Bereich extrahiert wird. Es entsteht eine neue Matrix mit reduzierter Höhe und Breite aber gleicher Tiefendimension als Ausgangsmatrix. Abbildung 6.2 veranschaulicht das MAX-Pooling mit einem 2×2 großen Filter und einer Schrittweite von zwei.

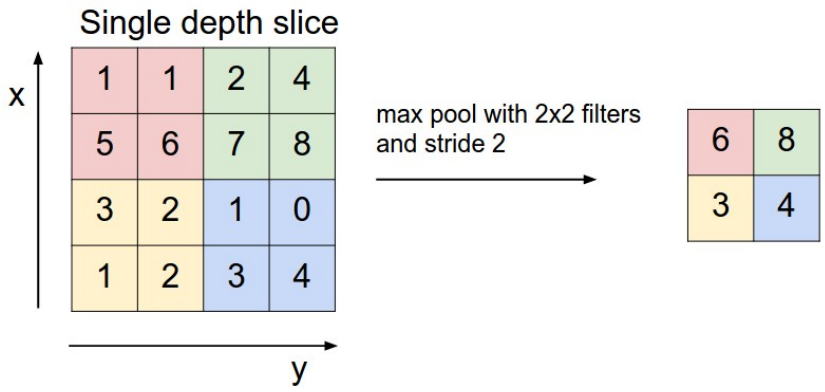


Abbildung 6.2. MAX-Pooling Operation. [34]

6.1.3 Fully-Connected Layer

Die letzten Lagen eines CNNs bestehen aus Fully-Connected Layers. Die Neuronen in einem Fully-Connected Layer sind mit allen Aktivierungen des vorherigen Layers verbunden. Sie schließen das CNN ab und erzeugen die Ausgabe des Netzes. Fully-Connected Layer übernehmen in der Verarbeitungskette der Bildklassifikation (Kapitel 4.2) die Aufgabe des Klassifikators. Dabei enthält das letzte Fully-Connected Layer meist die selbe Anzahl an Neuronen wie es zu Klassifizierende Objekte gibt. Das Neuron mit der höchsten Aktivierung steht dann für das erkannte Objekt.

6.1.4 CNN-Architektur

Die am häufigsten zur Bildklassifikation verwendete CNN-Architektur ist aus einer gewissen Anzahl von Convolutional Layern, verbunden mit einer Aktivierungsfunktion, auf die ein Pooling Layer folgt, aufgebaut. Dies wird wiederholt bis das Bild, durch Pooling, in einer sehr kleinen räumlichen Form mündet. Daraufhin folgt ein Fully-Connected Layer, welches die Klassifikation der Eingangsbilder übernimmt. Um eine korrekte Bildklassifikation zu ermöglichen, werden die Filter der Convolutional Layer und die Gewichte der Fully-Connected Layer durch überwachtes Lernen mit annotieren Trainingsbildern angepasst. In Abbildung 6.3 ist eine Klassifizierung eines Bilds durch solch eine CNN-Architektur, bestehend aus den einzelnen Hauptlagen und der Aktivierungsfunktion ReLU (Rectified linear unit), dargestellt.

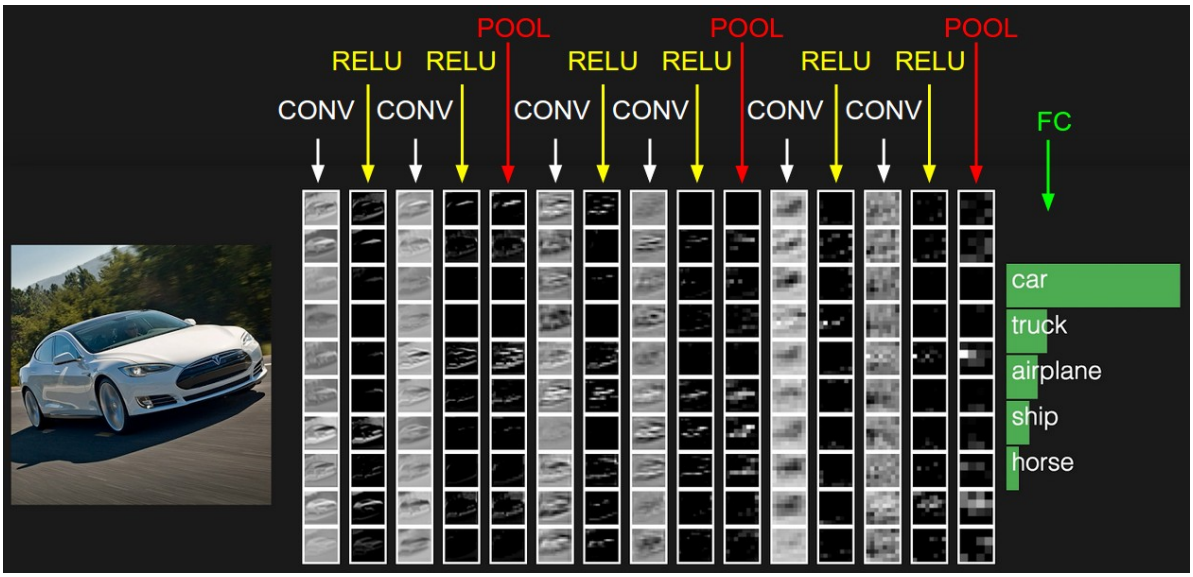


Abbildung 6.3. Eine typische CNN-Architektur. [35]

Es gibt jedoch weit aus komplexere CNN-Architekturen, wie das in Kapitel 4.6 erwähnte GoogLeNET oder das ResNET, welche eine viel höhere Genauigkeit bei der Klassifizierung ermöglichen.

6.2 Objektdetektion durch Convolutional Neural Networks

Dieser Abschnitt des Kapitel soll die typische Vorgehensweise eines auf CNNs basierenden Objektdetektionssystems verdeutlichen, um im nächsten Kapitel den besonderen Ansatz der Objektdetektion durch YOLO besser verstehen zu können. Dafür wird zuerst auf eine Möglichkeit der Objektlokalisation (Kapitel 4.4) durch CNNs eingegangen. Darauf wird aufgezeigt wie dieser Ansatz der Objektlokalisation auf die Objektdetektion erweitert werden kann.

6.2.1 Ein Ansatz der Objektlokalisation

Um eine Objektlokalisation durch CNNs zu ermöglichen, wird meist die Schätzung von Begrenzungsrahmen und das Klassifizieren eines Objekts getrennt betrachtet. Nach der Merkmalsextraktion durch die Convolutional Layer und Pooling Layer wird dafür das CNN in zwei Fully-Connected Layer Bereiche aufgespalten, welche getrennt trainiert werden. Der Klassifkationskopf wird auf das Klassifizieren der verschiedenen Objekte trainiert. Zum Schätzen der Begrenzungsrahmen wird Regression (Kapitel 3.2.4) mit der in Kapitel 4.4.2 vorgestellten Methode angewandt.

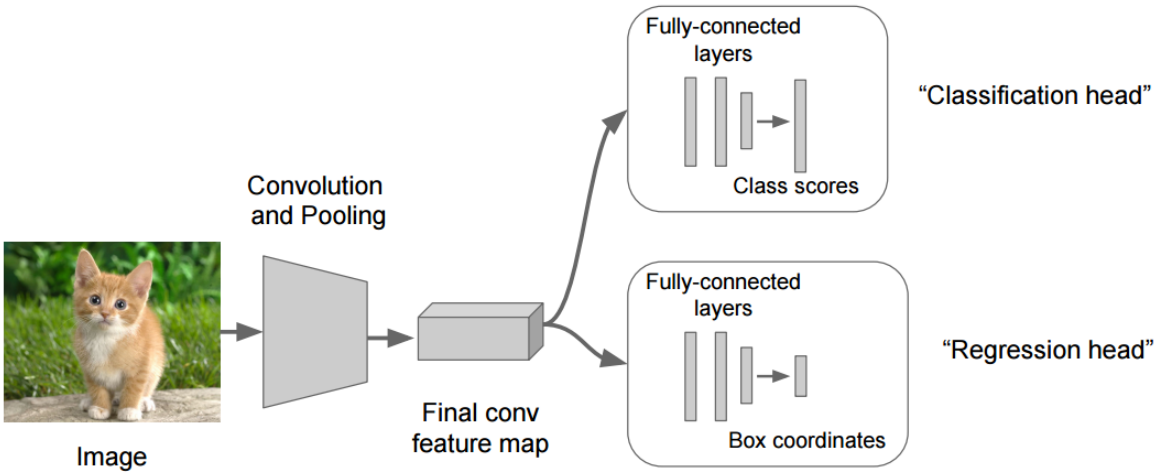


Abbildung 6.4. Eine Lösung der Objektlokalisation durch eine CNN-Architektur. [36]

6.2.2 Ein Ansatz der Objektdetektion

Bei der Verkehrszeichenerkennung im Straßenverkehr müssen meist mehrere Objekte auf einmal detektiert werden. Für das Lokalisieren und Klassifizieren von mehrere Objekten (Objektdetektion) in einem Bild, wird somit für jedes zu detektierende Objekt ein spezifisch trainierter Klassifikation- und Regressionskopf benötigt. Dies führt einen sehr hohen Trainingsaufwand des Objektdetektionssystem mit sich. Das Objektdetektionssystem R-CNN arbeitet auf diese Weise. R-CNN wendet eine ausgewählte CNN-Architekur (z.B. GoogLeNET oder ResNET) auf durch Algorithmen (Region Proposal Methoden) vorgeschlagenen Regionen eines Bildes an. Diese CNNs extrahieren Merkmale aus dem Bildbereich. Anschließend werden diese Merkmale durch binäre SVM-Klassifikatoren ausgewertet, welche zuvor auf die zu detektierenden

Objekte trainiert wurden. Zusätzlich werden Begrenzungsrahmen für jedes Objekt geschätzt. In Abbildung 6.5 ist dieses Verfahren grafisch dargestellt.

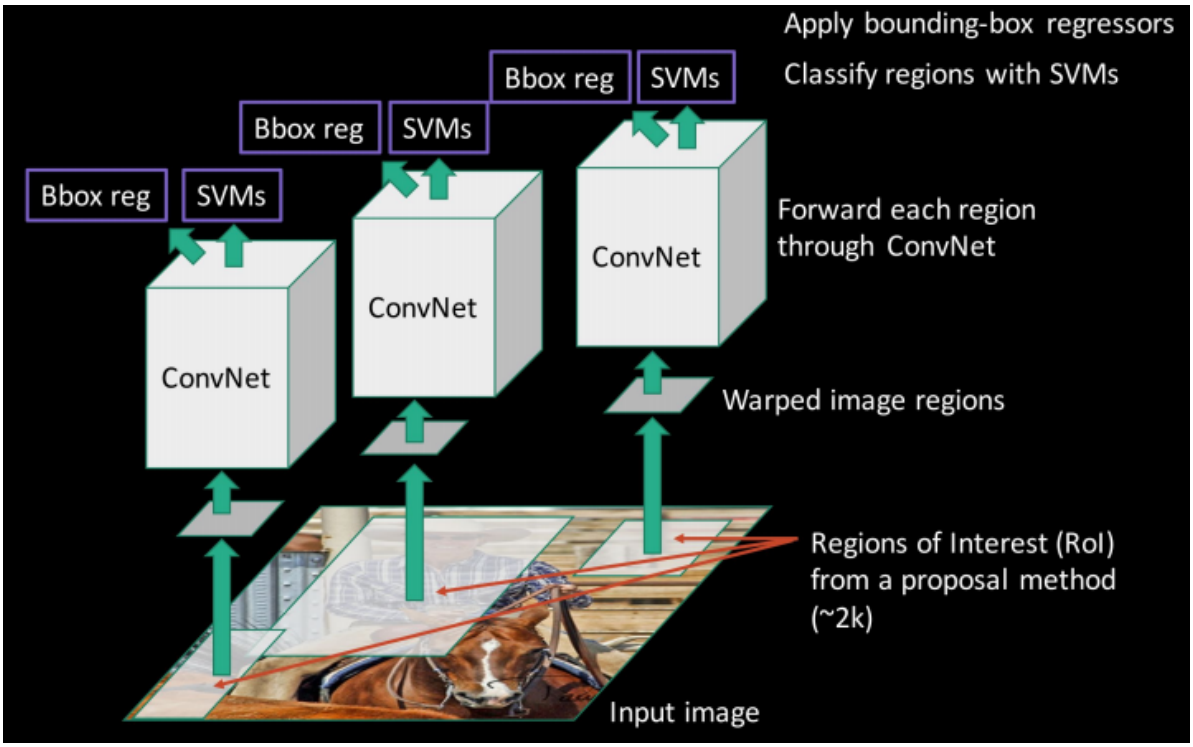


Abbildung 6.5. Eine Lösung der Objektdetektion durch eine CNN-Architektur. [37]

Dadurch, dass das Verfahren jeden seiner binären Klassifikatoren auf alle vorgeschlagenen Bereiche anwenden muss, um Objekte klassifizieren zu können, verarbeitetet es Bilder nur sehr langsam. Zudem erfordert ein solches Verfahren einen sehr großen Trainingsaufwand da jeder SVM-Klassifikator getrennt trainiert werden muss.

6.3 Das YOLO: Echtzeit Objektdetektionssystem

Typische Objektdetektionssysteme betrachten die Detektion als ein Klassifikationsproblem. YOLO verwendet einen anderen Ansatz der Objektdetektion. Es betrachtet die gesamte Detektion als ein Regressionsproblem von räumlich getrennten Begrenzungsrahmen und damit verbundenen Klassenauftrittswahrscheinlichkeiten. Im folgenden Abschnitt wird zuerst näher auf die Arbeitsweise von YOLO eingegangen. Anschließend wird die Architektur und der Ablauf des Trainings näher betrachtet.

6.3.1 Arbeitsweise

Bei der Anwendung der trainierten YOLO-Architektur auf ein Bild wird dieses zuerst auf 448×448 Pixel skaliert. Anschließend wird dieses Bild durch die CNN-Architektur propagiert.

Dabei wird das Eingangsbild in ein Raster, mit einer definierten Anzahl an Zellen (**S**), aufgeteilt. Für jede Zelle wird eine definierte Anzahl an Begrenzungsrahmen (**B**) geschätzt. Jeder Begrenzungsrahmen besitzt fünf charakteristische Werte: x, y, w ,h und einen Zuversichtlichkeitswert. Die (x,y) Koordinaten repräsentieren den Mittelpunkt des Begrenzungsrahmens relativ zur zugehörigen Zelle. Die Höhe (h) und die Breite (w) sind relativ zum Gesamtbild definiert. Der Zuversichtlichkeitswert gibt an, wie sicher es ist, dass dieser Rahmen ein gesuchtes Objekt enthält und gleichzeitig wie gut er das Objekt einrahmt (Kapitel 4.4.2). Außerdem wird für jede Zelle die Auftrittswahrscheinlichkeit der eintrainierten Klassen (**C**) geschätzt. Für jede Zelle werden die Klassenauftrittswahrscheinlichkeiten mit den Zuversichtlichkeitswerten der Begrenzungsrahmen multipliziert. Dies führt zu klassenspezifischen Zuversichtlichkeitswerten für jeden Begrenzungsrahmen einer jeden Zelle. Durch Festlegung eines Schwellwerts werden nur die Klassen mit ihren zugehörigen Begrenzungsrahmen im Bild markiert, welche mit ihrem klassenspezifischen Zuverlässigkeitswert diesen Schwellwert überschreiten. In Abbildung 6.6 ist diese Arbeitsweise noch einmal veranschaulicht.

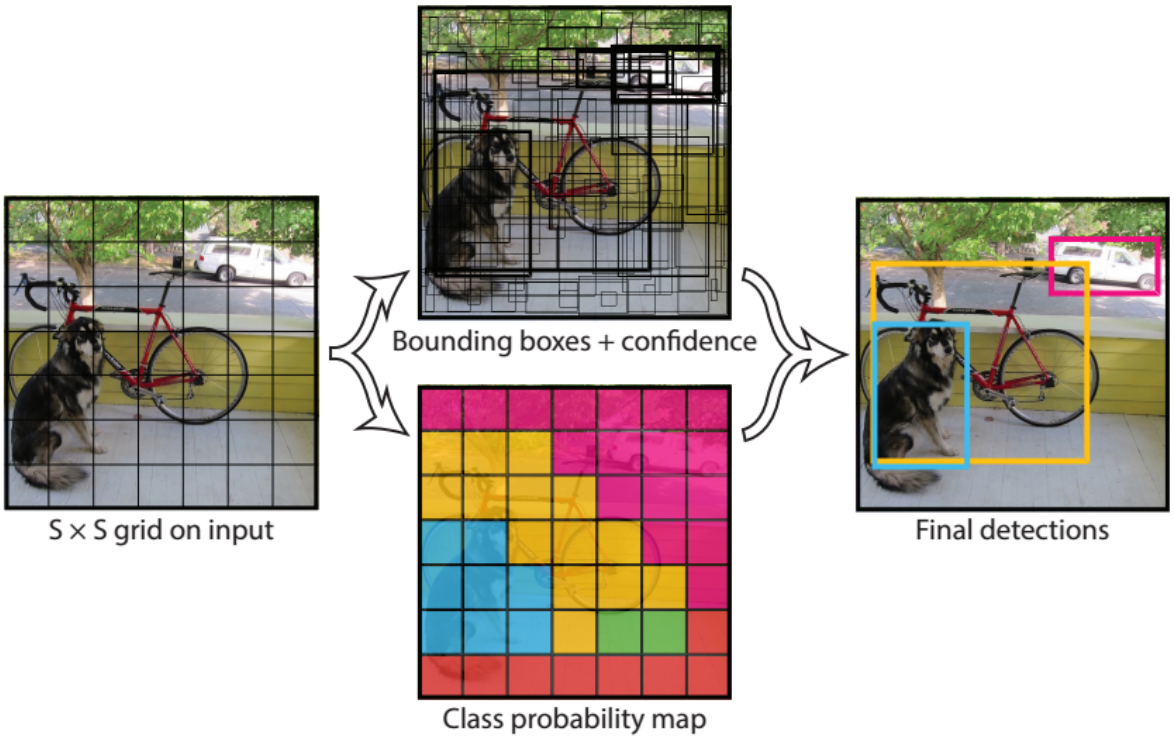


Abbildung 6.6. Arbeitsweise des YOLO-Objektdetektionssystems. [38]

6.3.2 Architektur

YOLO ist durch die GoogLeNET-Architektur inspiriert worden und besteht aus einer Kombination von 24 Convolutional Layern und Pooling Layern gefolgt von zwei Fully-Connected Layern (26 Lagen). In Abbildung 6.7 ist diese Architektur dargestellt.

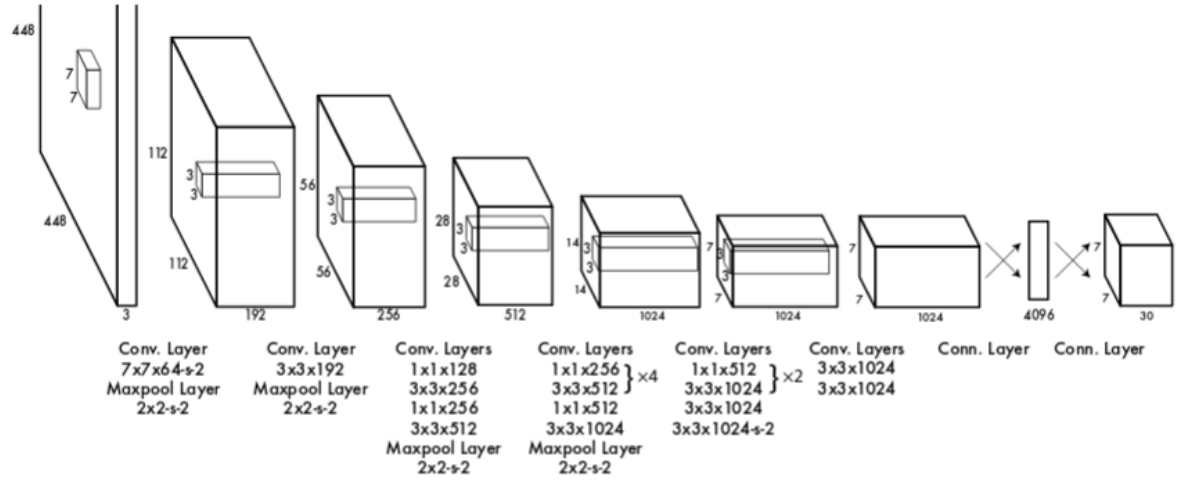


Abbildung 6.7. Architektur des YOLO-Objektdetektionssystems. [19]

Durch den speziellen Ansatz der Objektdetektion, gestaltet sich die Ausgabe von YOLO anders als bei herkömmlichen Verfahren. Der letzte Fully-Connected Layer des YOLO kann als ein Tensor betrachtete werden, welcher abhängig von den in Kaptitel 6.3.1 genannten Parametern ist und folgende Dimension besitzt:

$$\begin{array}{ccccc} \text{Breite} & \times & \text{Höhe} & \times & \text{Tiefe} \\ \text{S} & \times & \text{S} & \times & (5 \cdot \text{B} + \text{C}) \end{array}$$

6.3.3 Training

Das Training dieser CNN-Architektur wird mit annotierten Trainingsbildern realisiert, bei denen die Ground Truth Daten der enthaltenen Klassen bekannt sind. Durch Optimierung einer speziellen Verlustfunktion werden die Gewichte des CNNs so angepasst, dass die im beschriebenen Tensor codierten Voraussagungen auf die Position der Begrenzungsrahmen und Klassen der Trainingsbilder passen. Während der Testphase können dann Objekte in unbekannten Bildern detektiert werden.

Der Vorteil dieser Architektur und des Trainings ist, dass nicht für jedes zu detektierende Objekt ein Klassifikator trainiert werden muss, sondern nur der Tensor auf die Eingangsbilder angepasst werden muss (End to End Training). Dies ermöglicht ein einfaches Eintrainieren beliebig vieler Objekte.

6.4 Training des YOLO-Objektdetektionssystems mit dem Darknet-Framework

Um mit dem YOLO-Objektdetektionssystem eine Verkehrszeichenerkennung zu ermöglichen wurde das Darknet-Framework genutzt. Dieses Framework ist eine Programmierumgebung, basierend auf C und CUDA, ausgelegt um KNN-Architekturen zu erzeugen, zu trainieren und zu nutzen. Für das Training von YOLO in Kombination mit dem Darknet-Framework wurde ein Softarwarepaket [30] genutzt, welches dafür ausgelegt ist, ein Training von YOLO zur Detektion eigener Objekte zu ermöglichen. Mit Hilfe dieses Softwarepakets wurde eine YOLO-Tiny-Architektur auf die Detektion eines „Vorfahrt gewähren“- und eines „Stopp “-Schilds trainiert. Da diese Methode des Trainings für alle Verkehrszeichen gleich zu realisieren ist, wird im Folgenden eine Anleitung von der Vorbereitung, bis zu der fertig trainierten YOLO-Tiny-Architektur gegeben.

6.4.1 Sammeln von Trainingsbildern

Wie beim Training des Haar-Kaskaden-Klassifikators (Kapitel 5.2) auch, werden für das Training von YOLO Trainingsbilder benötigt. Im Gegensatz zu diesem arbeitet YOLO aber nur mit positiven Trainingsbildern. Die Trainingsbilder müssen zudem die gleichen Proportionen haben wie die Bilder in denen YOLO nach dem Training Klassen detektieren soll. Dies liegt daran, dass YOLO alle Eingangsbilder auf 448×448 Pixel skaliert. Die Trainingsbilder müssen entweder von Hand gesammelt werden oder es muss ein Datensatz gefunden werden der passend für das Training des YOLO ist.

In dem angebotenen Softwarepaket waren bereits 484 Trainingsbilder von „Stopp“-Schildern und 284 von „Vorfahrt gewähren“-Schildern enthalten. In Abbildung 6.8 ist ein Traininsbild aus jeder Verkehrszeichen-Klasse aufgeführt.

Abbildung 6.8. Eintrainierte Verkehrszeichen-Klassen. [30]

Dieser Trainingsdatensatz wurde im Speicherort des Darknet-Frameworks platziert.

6.4.2 Erzeugen einer Trainingsliste

Es wird eine Trainingsliste in Form einer Textdatei benötigt, welche Verzeichnispfade zu allen Trainingsbildern enthält. Durch das in dem Softwarepaket enthaltene Python-Skript *convert.py* konnte solch eine Liste erzeugt werden.

6.4.3 Erzeugen von Beschreibungsdateien

Für jedes Trainingsbild wird eine eigene Textdatei mit Metainformationen benötigt. Diese Textdateien enthalten die Information über die im Trainingsbild enthaltene Klasse und beschreiben die Position eines Begrenzungsrahmens, der die Klasse im Bild umrahmt. Dies sind die Ground Truth Informationen, auf welche das Darknet-Framework die Gewichte der YOLO-Tiny-Architektur während des Trainings anpasst.

Auch diese Beschreibungsdateien wurden bereits in dem Softwarepaket zur Verfügung gestellt und auch genutzt. Um selbst solche Beschreibungsdateien zu erzeugen, kann ein Software-Tool namens *Bbox-Label-Tool* verwendet werden. Mit diesem können Trainingsbilder annotiert werden, indem von Hand Begrenzungsrahmen um die im Bild enthaltenen Klasse gezogen werden. Die Parameter des Begrenzungsrahmens, mit zusätzlicher Identifikationsnummer der im Bild enthaltenen Klasse, werden anschließend in einer Textdatei gespeichert. Mit dem Python-Skript *convert.py* müssen die Parameter der Textdatei dann noch in das Format konvertiert werden, welches das Darknet-Framework auslesen kann.

6.4.4 Auswahl einer Konfigurationsdatei

Für das Training und die spätere Detektion wird eine Konfigurationsdatei benötigt. Diese Datei enthält Parameter, welche den Aufbau der YOLO-Tiny-Architektur beschreiben. Zudem können auch Trainingsparameter festgelegt werden. Diese Konfigurationsdatei muss dem Darknet-Framework beim Einleiten des Trainings übergeben werden. Für die YOLO-Architektur stehen bereits vorgefertigte Konfigurationsdateien zur Verfügung [39].

Da in dieser Arbeit die YOLO-Tiny-Architektur genutzt werden sollte, wurde die *yolo-tiny.cfg*-Konfigurationsdatei gewählt.

6.4.5 Anpassen der Konfigurationsdatei

Die Detektionsqualität von YOLO wird durch viele sogenannte Hyperparameter beeinflusst, die in der Konfigurationsdatei eingestellt werden können. Durch Feintuning dieser Hyperparamter können bessere Detektionsergebnisse erzielt werden. Das Einstellen dieser Hyperparamter erfodert tiefgreifende Kenntnisse im Bereich des maschinellen Lernens und viel Erfahrung im Training von KNNs.

Ein wissenschaftlicher Bericht der Stanford Universität [40] beschäftigt sich mit der Verkehrszeichendetektion durch YOLO. Folgende Parameter wurden dabei als effizient ermessen:

batch	64	Trainings-Batch-Größe
learning_rate	0.001	Lernrate
drop_out	0.5	Neuronen Auswurfrate
jitter	0	Zufälliges Zuschneiden von Trainingsbildern
momentum	0.9	Optimierungsmethode

Diese Parameter wurden für das Training genutzt. Außerdem mussten noch weitere Parameter angepasst werden, da wie in Kapitel 6.3.2 beschrieben die Größe des Ausgabe-Tensors von den Trainingsparametern abhängig ist.

Die Anzahl der Klassen (**C**) wurde auf zwei gesetzt, da zwei Verkehrszeichen eintrainiert werden sollten. Das Raster (**S**×**S**), welches das Bild aufteilt, wurde auf 11×11 gesetzt, da Verkehrszeichen meist nur einen kleinen Bereich des Gesamtbildes einnehmen. Mit einem feinerem Raster sollte eine präzisere Vorhersage der Begrenzungsrahmen erreicht werden. Die Anzahl der pro Zelle zu schätzenden Begrenzungsrahmen (**B**) wurde auf zwei gesetzt. Durch die Parameter ergibt sich ein Tensor folgender Dimension:

Breite	×	Höhe	×	Tiefe
S	×	S	×	$(5 \cdot \mathrm{B} + \mathrm{C})$
11	×	11	×	$(5 \cdot 2 + 2)$
11	×	11	×	12

Dieser Tensor besteht aus 11· 11 · 12 Werten. Das Ausgangslayer der YOLO-Tiny-Konfigurationsdatei musste somit noch auf den Wert 1452 geändert werden.

6.4.6 Anpassen des Quellcodes

Bevor das Training gestartet werden kann, muss der Quellcode angepasst werden.

In der *yolo.c*-Datei musste die Anzahl der Klassen auf zwei gesetzt werden. Zusätzlich mussten die Dateipfade zur Trainingsliste und den Beschreibungsdateien angepasst werden. Außerdem musste die Anzahl der Klassen auch in der *yolo_kernels.cu*-Datei geändert werden.

6.4.7 Training der YOLO-Tiny-Architektur

Das Training von YOLO-Tiny fand wieder auf dem hochschuleigenen PC, ausgerüstet mit einer NVIDIA Titan X GPU statt. Gestartet werden konnte das Training durch den Befehl:

./darknet yolo train cfg/yolo-tiny.cfg

Mit dem durch die GPU parallelisierten Training, dauerte das Anpassen der YOLO-Tiny-Architektur ungefähr 6 Stunden. Nach dem Training speicherte das Darknet-Framework die trainierten Gewichte in einer Datei namens *yolo-tiny.weights*. Diese Datei enthält die auf die beiden Verkehrszeichen trainierten Gewichte und kann in Kombination mit der zugehörigen Konfigurationsdatei zur Verkehrszeichendetektion genutzt werden.

6.5 Verkehrszeichendetektion bei der autonomen Fahrt mit YOLO und ROS

Um eine Verkehrszeichendetektion durch YOLO bei der autonomen Fahrt zu ermöglichen, wurde das ROS-Package *darknet_ros* [31] in das Robot Operating System eingebunden. Dieses Package stellt ein ROS-Interface für das Darknet-Framework zur Verfügung und enthält eine ROS-Node namens *yolo_object_detector* auf deren Funktion in diesem Abschnitt eingangen wird. Außerdem wird auf die Erweiterung der *motor_controller*-Node eingegangen, durch die dem Fahrzeug eine Reaktion auf die durch YOLO detektierten Verkehrszeichen ermöglicht werden kann.

6.5.1 Die *yolo_object_detector* Node

Die *yolo_object_detector* Node liest die YOLO-Tiny-Konfigurationsdatei aus und erzeugt diese CNN-Architektur. Zusätzlich liest sie die *yolo-tiny.weights*-Datei aus und initialisiert die CNN-Architektur mit den trainierten Parametern. Die Node liest zudem die Kameradaten des Fahrzeugs aus der Topic *camera/image/compressed* aus und propagiert die einzelnen Frames durch das CNN. Wird ein Verkehrszeichen detektiert, sendet diese Node eine ROS-Message auf die Topic */YOLO_bboxes*. Diese Message enthält Informationen über das detektierte Verkehrszeichen und über Parameter eines Begrenzungsrahmens der das Verkehrszeichen im Bild einrahmt. Desweiteren werden die analysierten Frames in einem Fenster auf dem Laptop dargestellt, wobei die detektierten Verkehrszeichen durch Begrenzungsrahmen farblich umrahmt werden. Abbildung 6.9 zeigt die Detektion der zwei eintrainierten Verkehrszeichen aus den Kameradaten des Fahrzeugs.

Abbildung 6.9. Verkehrszeichendetektion mit YOLO.

6.5.2 Erweiterung der *motor_controller* Node

Um auf die durch YOLO erkannten Verkehrszeichen reagieren zu können liest die *motor_controller* Node die Topic */YOLO_bboxes* aus. Je nach detektiertem Verkehrszeichen kann eine Reaktion des Fahrzeugs, durch entsprechendes Ändern der Steuerungssignale ermöglicht werden.

7 Testen der eingesetzten Methoden

Da die verwendeten Algorithmen zur Realisierung einer autonomen Steuerung (MLP) und der Verkehrszeichenerkennenung (Objektdetektionssysteme) alle auf maschinellem Lernen beruhen und die Qualität der Methoden abhängig von den Hyperparametern und den zur Verfügung gestellten Trainingsdaten sind, ist ein Qualitätstest dieser schwierig zu gestalten. Um auf Verkehrszeichen und Änderungen der Fahrstrecke bei der autonomen Fahrt schnell reagieren zu können, ist jedoch vor allem die Bildverarbeitungszeit der Algorithmen ausschlaggebend. Zudem muss bei autonomen Fahrzeugen die Bildverarbeitung und Steuerung intern stattfinden, wodurch die Auswahl der Algorithmen von den Speicherressourcen der auf dem Fahrzeug befindlichen Recheneinheit abhängig ist. Die beiden trainierten Objektdetektionssysteme und das MLP, welches die Steuerung des Fahrzeugs übernimmt, wurden getrennt auf ihre Laufzeit und Speicheranforderungen getestet. Die Tests wurden realisiert, indem die Kameradaten des Fahrzeugs ausgelesen wurden und die Algorithmen einzeln, ausgeführt auf dem Laptop (Technische Daten in Kapitel 2.2), auf diese angewandt wurden. Das MLP und YOLO-Tiny führten die Berechnungen auf der Grafikkarte (NVIDIA GeForce GT 755M) aus. Der OpenCV Haar-Kaskaden-Klassifikator arbeitete allein mit der CPU (Intel i7-4700MQ).

7.1 Bildverarbeitungszeit

Um die Bildverarbeitungszeit der einzelnen Algorithmen testen zu können, wurde die Differenz zweier Timer berechnet. Diese wurden vor und nach der Ausführung der Algorithmen im Quellcode gesetzt. Das Ergebnis dieser Differenz wurde in der Kommandozeile ausgegeben.

Das trainierte MLP benötigte zwischen 0.0027 und 0.003 Sekunden um aus einem eingespeisten Kamerabild einen Fahrbefehl zu generieren.

Der Quellcode der beiden Objektdetektionssysteme wurde auf eine Detektion von einem „Vorfahrt gewähren“- und einem „Stopp“-Schild angepasst, um gleiche Ausgangsverhältnisse herzustellen. Deren Verarbeitungsgeschwindigkeit wurde auf zwei verschiedene Bildauflösungen getestet. Diese sind in folgender Tabelle dargestellt:

Tabelle 7.1. Bildverarbeitungszeit.

Bildauflösung	YOLO-Tiny	Haar-Kaskaden-Klassifikatoren
320×240	0.08s - 0.11s	0.016s
640×480	0.09s	0.065s - 0.125s

7.2 Speicheranforderungen

Um die Arbeitsspeicheranforderungen der Algorithmen zu messen, wurde der Prozessmanager *htop* verwendet. Dieser bietet eine Übersicht über laufende Prozesse sowie belegte Systemressourcen. Dabei wurde die Differenz zwischen den zuvor belegten Speicherressourcen und der während der Ausführung der Algorithmen berechnet. Genau auf gleiche Weise wurde das NVIDIA eigene Programm *nvidia-settings* verwendet, um die Grafikkartenspeicheranforderungen der Algorithmen zu berechnen. Auch hier wurden die Objektdetektionssysteme auf zwei verschiedene Bildauflösungen angewandt. In folgender Tabelle sind die Ergebnisse dargestellt:

Tabelle 7.2. Arbeitsspeicheranforderungen.

Bildauflösung	MLP	YOLO-Tiny	Haar-Kaskaden-Klassifikatoren
320×240	335 MB	325 MB	32 MB
640×480	-	336 MB	54 MB

Tabelle 7.3. Grafikkartenspeicheranforderungen.

Bildauflösung	MLP	YOLO-Tiny	Haar-Kaskaden-Klassifikatoren
320×240	1158 MB	530 MB	-
640×480	-	530 MB	-

8 Ergebnisse

In diesem Kapitel werden die Ergebnisse und Erkenntnisse der eingesetzten Methoden vorgestellt. Zuerst wird auf das genutzte Robot Operating System eingegangen. Anschließend wird sich mit dem allgemeinen Aufbau des Fahrzeugs und dessen manueller Steuerung auseinandergesetzt. Darauf folgt die autonome Steuerung durch Künstliche Neuronale Netze. In den letzten beiden Abschnitten werden die angewandten Objektdetektionssysteme behandelt.

8.1 Robot Operating System

Das Robot Operating System erleichterte den Aufbau des Steuerungssystems zwischen dem Laptop und Fahrzeug in vielen Bereichen. Zum einen wurde durch das *joy*-Package eine manuelle Steuerungsmöglichkeit durch einen Controller geliefert. Durch Multiple Machines konnte der Aufbau des Kommunikationsnetzwerks zwischen dem Fahrzeug und dem Laptop ermöglicht werden. Auch das Auslesen der Kameradaten konnte durch das *raspicam*-Package schnell ermöglicht werden. Vor allem das Beschreiben und Auslesen von Topics machte ROS sehr nützlich, da neue Softwareprogramme (Nodes) schnell in das ROS-Framework integriert werden konnten und ihnen die benötigten Daten (Kameradaten, Steuerungsbefehle) einfach zur Verfügung gestellt werden konnten. Abbildung 8.1 veranschaulicht noch einmal alle Datenwege, welche über ROS realisiert wurden.

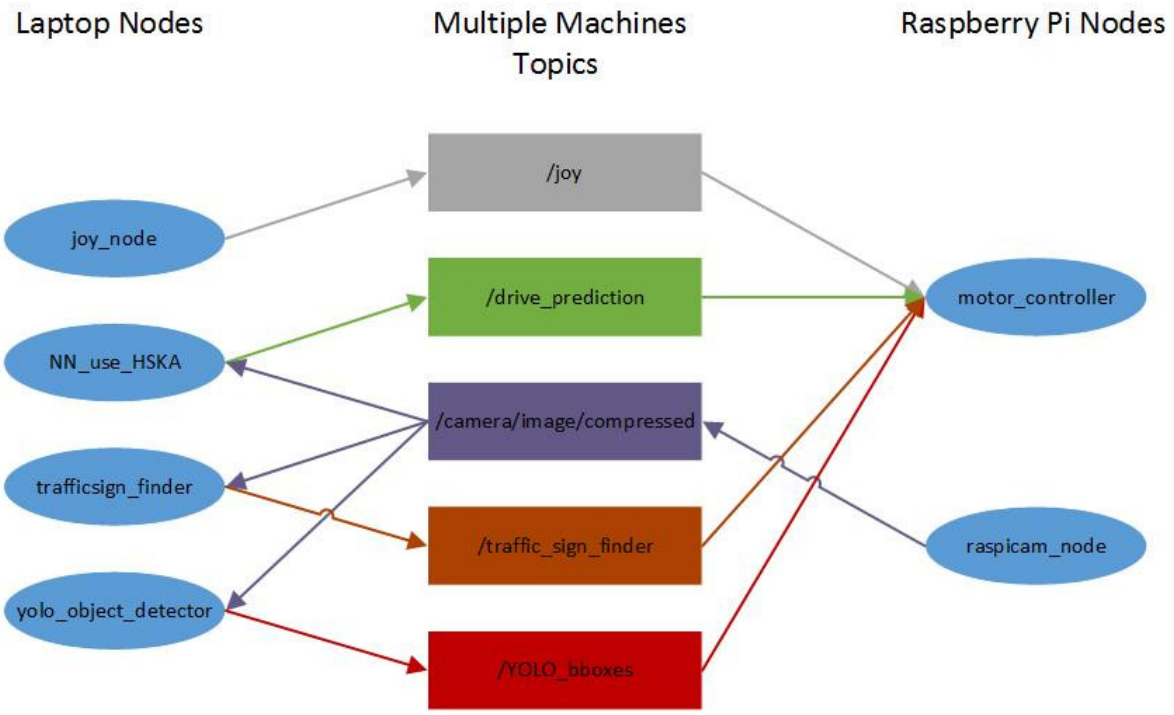


Abbildung 8.1. ROS Datenwege.

ROS ist jedoch kein Echtzeitbetriebssystem, da es ein auf Linux aufgesetztes Framework ist. Die Latenzzeiten von ROS sind deshalb von allen Prozessen, die auf dem Linux Betriebssystem ausgeführt werden, abhängig und können somit nicht vorhergesagt werden.

8.2 Fahrzeugaufbau und manuelle Steuerung

Das Fahrzeug lässt sich durch einen Controller über den Laptop steuern. Der in Abbildung 2.2 präsentierte Schaltungsaufbau bot eine einfache Lösung zur Steuerung des Fahrzeugs. Durch die hohe Kapazität der Powerbank (16750 mAh) ist eine lange Betriebszeit des Fahrzeugs möglich. Nach dem Sammeln der Trainingsdaten für das MLP (60 Fahrten), auf der in Abbildung 3.4 gezeigten Teststrecke, war die Powerbank immer noch zu drei Vierteln geladen (laut LED-Anzeige). Das manuelle Sammeln der Trainingsdaten erwies sich jedoch als schwierig. Zum einen lag dies an den eingeschränkten Lenkeigenschaften des Fahrzeugs (Kapitel 2.1), zum anderen daran, dass die Kommunikation zwischen dem Laptop und dem Fahrzeug des Öfteren abbrach, wenn zu viel Datenverkehr auf dem Hochschulnetzwerk vorhanden war. Dies ist vor allem auf die geringe Leistung des genutzten WLAN-Sticks zurückzuführen. Außerdem hatte das Fahrzeug Probleme, sich auf rauen Untergründen fortzubewegen. Deshalb stand an

der Hochschule nur eine kleine Teststrecke auf einem Tisch zur Verfügung, durch welchen die möglichen Teststrecken begrenzt wurden. Da die DC-Motoren zur Steuerung des Fahrzeugs im Niedrigpreissektor angesiedelt sind und zur Generierung von Trainingsdaten lange Fahrten nötig waren, könnte das Fahrzeug zudem bald auf neue Motoren angewiesen sein.

8.3 Autonome Steuerung des Fahrzeugs

Nach der Fertigstellung dieser Arbeit konnte das Fahrzeug autonom die in Abbildung 3.4 gezeigte Teststrecke, durch Auswertung der Kameradaten mit dem trainierten MLP, abfahren. Die Reaktion des Fahrzeugs auf verschiedene Verkehrszeichen wurde durch Nachrichtenaustausch der beteiligten Steuerungs- und Objektdetektions-Nodes (*motor_controller*-, *traffic-sign_finder*-, *yolo_object_detector*) ermöglicht. Als eindeutige Reaktion des Fahrzeugs wurde hierbei bereits ein farbliches Markieren der Verkehrszeichen in einem auf dem Laptop präsentierten Fenster und das Bestätigen der Detektion eines bestimmten Verkehrszeichens in der Kommandozeile der Steuerungs-Node akzeptiert.

Durch die schnelle Auswertung der Bilder im Millisekundenbereich (Kapitel 7.1) kann eine Reaktion des Fahrzeugs in Echtzeit ermöglicht werden.

Wie die Testergebnisse in Kapitel 7.2 aber zeigen, benötigt das MLP viel Speicherressourcen um ausgeführt zu werden. Dies liegt an der hohen Anzahl an Neuronen im Input-Layer des MLP und zudem daran, dass Tensorflow viele nicht unbedingt notwendige Speicherressourcen reserviert.

Das Training des MLP konnte auf dem in dieser Arbeit genutzten Laptop, mit CUDA und Tensorflow, ermöglicht werden. Es fand auf dessen Grafikkarte (NVIDIA GeForce GT 755M) statt. Die Trainingszeit des MLP betrug ungefähr 30 Minuten. Bei den Testfahrten wurde jedoch festgestellt, dass sich das MLP nur schlecht auf neue Fahrstrecken einstellt. Nach Umstellung der Pylonen oder einem Ändern der Lichtverhältnisse konnte das MLP keine korrekten Vorhersagen mehr treffen. Beim Training wurden dem MLP aber auch nur die monotonen Trainingsdaten (9300 Bilder mit zugehörigen Fahrbefehlen) dieser einen Teststrecke präsentiert. Dadurch lernte das MLP die Fahrstrecke einfach auswendig. Um eine Generalisierung des MLP auf verschiedene Fahrstrecken zu ermöglichen, muss eine Überanpassung des MLP auf den Trainingsdatensatz verhindert werden. Dies kann mit Hilfe eines Validierungsdatensatzes geschehen, welcher Daten enthält, die das MLP zuvor noch nicht gesehen hat. Das Training des MLP wird dabei solange fortgeführt, bis seine Leistung für den unabhängigen Datensatz optimal ist und dann sofort gestoppt. Zudem müsste der Trainingsdatensatz selbst eine größere Variation aufweisen. Wie zum Beispiel verschiedene Lichtverhältnisse oder andere Anordnungen der Pylonen und daraus resultierende neue Fahrmanöver.

8.4 Verkehrszeichenerkennung durch Haar-Kaskaden-Klassifikatoren

Durch trainierte Haar-Kaskaden-Klassifikatoren in Kombination mit der *trafficsign_finder*-Node ist es möglich, drei verschiedene Verkehrszeichen („Vorfahrt gewähren“-, „30er Zone“-, „Stopp“-Schild) in den Kamerabildern des Fahrzeugs zu detektieren (siehe Abbildung 5.6).

Die Merkmalsextraktion durch Haar-ähnliche Merkmale beschränkt sich nur auf die Form der Verkehrszeichen und berücksichtigt nicht deren Farbe, da der Algorithmus mit Bildern im Graustufenformat arbeitet. Verkehrszeichen müssen aber auch von farbenblinden Personen unterschieden werden können. Deshalb ist nicht klar, inwieweit Vorteile bei der Klassifikation durch das Einbeziehen der Farbe ermöglicht werden können.

Verkehrszeichen können durch das Verfahren gut detektiert werden, da durch die Sliding-Window Klassifikation (Kapitel 4.4.1) jeder Bildbereich in verschiedenen Skalierungen abgesucht wird. Dies ermöglicht eine präzise Schätzung von Begrenzungsrahmen. In Kombination mit der Kaskadierung der Haar-Klassifikatoren ermöglicht die Sliding-Window Klassifikation immer noch eine sehr schnelle Auswertung.

Das Verfahren verarbeitet Bilder sehr schnell (siehe Kapitel7.1). Die Verarbeitungsgeschwindigkeit ist jedoch von einigen Faktoren abhängig. Zum einen ist die Bildauflösung ein ausschlaggebender Punkt, da das Verfahren mit der Sliding-Window Klassifikation arbeitet. Bei einer höheren Auflösung muss somit eine größere Anzahl an Bildausschnitten klassifiziert werden. Außerdem wird die Geschwindigkeit des Verfahrens von der Anzahl der zu erkennenden Objekte beeinflusst, da jeder Haar-Kaskaden-Klassifikator einzeln auf das Bild angewandt werden muss (dies kann durch Multithreading umgangen werden). Auch die Anzahl der im Bild enthaltenen Objekte spielt eine Rolle, da für jede Detektion alle Stufen des Haar-Kaskaden-Klassifikators durchlaufen werden müssen.

Ein weiterer Vorteil des Haar-Kaskaden-Klassifikator-Verfahrens ist, das es ein sehr leichtgewichtiger Prozess ist. Zur Ausführung sind nur geringe Systemrressourcen erforderlich. Für die Anwendung des Verfahrens auf einem Bild mit der Auflösung 320×240 wurden nur 32 MB RAM benötigt (siehe Kapitel 7.2).

Sehr viel Zeit nimmt das Sammeln und Annotieren von eigenen Trainingsbildern in Anspruch. Zudem muss für dieses Verfahren für jedes zu detektierende Verkehrszeichen ein eigener binärer Klassifikator trainiert werden. Die Trainingszeit eines Klassifikators, auf die Detektion eines Verkehrszeichens, betrug ungefähr drei Stunden. Das Training konnte auf dem Laptop ermöglicht werden und fand auf dessen CPU (Intel i7-4700MQ) statt, der 16 GB RAM zur Verfügung standen. Um gute Detektionsergebnisse zu erzielen, verlangt das Verfahren tiefere Kenntnisse im Bereich des maschinellen Lernens, da Trainingsparameter eingestellt werden müssen.

Die trainierten Haar-Kaskaden-Klassifikatoren passen sich sehr gut auf verschiedene Lichtverhältnisse an. Sehr stark und sehr schwach beleuchtete Verkehrszeichen werden immer noch detektiert. Außerdem lassen sich auch Verkehrszeichen in verschiedenen Perspektiven detektieren. Dies ist auf den qualitativ sehr hochwertigen GTSRB-Trainingsdatensatz [26] zurückzuführen, der eine große Anzahl an verschiedenen Verkehrszeichen enthält. Weiter entfernte Verkehrszeichen lassen sich mit den derzeit trainierten Haar-Kaskaden-Klassifikatoren schlechter detektieren (Abbildung 5.6 verdeutlicht dies). Die Detektionsentfernung ist abhängig von der

Fenstergröße der Sliding-Window Klassifikation und könnte bei einem weiteren Training verkleinert werden. Die eigens eintrainierten Haar-Kaskaden-Klassifikatoren haben zudem Probleme, Verkehrszeichen mit einer niedrigen Interklassendistanz zu unterscheiden (z. B. zwischen einem „30er Zone“- und einem „50er Zone“-Schild). Dies ist darauf zurückzuführen, dass den binären Klassifikatoren beim Training keine negativen Bilder mit solch ähnlichen Schildern präsentiert wurden. Zudem macht das Verfahren des Öfteren falsch-positive Detektionen. Diese müssen bei dem Haar-Kaskaden-Klassifikator-Training aber in Kauf genommen werden (Kapitel 5.2.5). Mit dem Einstellen des Parameters *minNeighbours = 3* (minimale Anzahl von Detektionsnachbarn) in der *trafficsign_finder* Node (siehe Kapitel 5.3.1), konnten objektiv die besten Ergebnisse (Verhältnis von richtig-positiver zu falsch-positiver Detektionsrate) erreicht werden.

8.5 Verkehrszeichenerkennung durch YOLO-Tiny

Die trainierte CNN-Architektur YOLO-Tiny wird durch die *yolo_object_detector*-Node ausgeführt und erlaubt es, „Vorfahrt gewähren“- und „Stopp“-Schilder in den Kamerabildern des Fahrzeugs zu detektieren (siehe Abbildung 6.9).

Durch die Convolutional- und Pooling-Layer der CNN-Architektur können form- und farbbasierte Verkehrszeichenmerkmale aus Bildern extrahiert werden. Zudem ist YOLO durch seine spezielle Architektur dazu fähig, Aussagen über eine im Bild enthaltene Klasse auf Grundlage des Gesamtbilds zu machen. Da Verkehrszeichen aber an verschiedenen Orten vorkommen können, bietet dies keinerlei Vorteile.

Da YOLO bei der Detektion von Verkehrszeichen deren Form und Farbe berücksichtigt, macht das Verfahren wenige falsch-positive Detektionen. Jedoch ist die Schätzung der Begrenzungsrahmen abhängig von den Trainingsdaten, da die Schätzungen beim Training auf die Ground Truth Daten der Trainingsbilder angepasst werden.

YOLO besitzt eine relativ konstante und schnelle Verarbeitungsgeschwindigkeit von Bildern. Diese ist unabhängig von der Bildauflösung, da jedes Bild auf eine vorher festgelegte Größe skaliert wird (siehe Kapitel 7.1). Zudem ist diese Verarbeitungsgeschwindigkeit auch unabhängig von der Anzahl der zu detektierenden Objekte, da für jede Zelle in nur einer Anwendung auf das Bild Klassenauftrittswahrscheinlichkeit und Begrenzungsrahmen geschätzt werden können. Dies bringt bei der Detektion von vielen Verkehrszeichen Geschwindigkeitsvorteile.

Die Speicheranforderungen von YOLO-Tiny, sind trotz der kleineren Architektur zum Original YOLO und der damit verbundenen schlechteren Detektionsqualität, immer noch sehr hoch. YOLO-Tiny benötigte bei der Anwendung auf ein Bild (Auflösung: 320×240) 325 MB RAM und 530 MB Grafikkartenspeicher.

Auch für YOLO-Tiny müssen Trainingsbilder gesammelt werden, der Aufwand gleicht dem der Haar-Kaskaden-Klassifikatoren. Durch die spezielle Architektur von YOLO wird der Trainingsaufwand aber minimiert, da beliebig viele Verkehrszeichen auf einmal eintrainiert werden können. Das Training wurde auf der Grafikkarte (NVIDIA GeForce GTX TITAN X) eines hochschuleigenen PCs realisiert. Die Trainingszeit betrug ungefähr sechs Stunden. Eine CNN-Architektur stellt jedoch ein sehr komplexes System dar, welches beim Training durch viele Hyperparameter beeinflusst werden kann. Um eine gute Detektionsqualität zu ermöglichen, müssen tiefgreifende Kenntnisse im Bereich des Trainings von KNNs vorhanden sein.

Das trainierte YOLO-Tiny erkennt die beiden Verkehrszeichen in verschiedenen Perspektiven sehr gut. Das Unterscheiden von ähnlichen Verkehrszeichen (Interklassenabstand) stellt jedoch ein Problem dar. Bei verschiedenen Lichtverhältnissen oder gleichfarbigen Hintergründen werden zudem Begrenzungsrahmen auffällig falsch geschätzt. Dies liegt zum einem an dem sehr kleinen und monoten Trainingsdatensatz der für das Training zur Verfügung stand (siehe Kapitel 6.4.1). Zum anderen daran, dass YOLO die Klassifikation auf Grundlage des Gesamtbilds ausführt und Verkehrszeichen meist nur einen sehr kleinen Teil dessen ausmachen. Zudem hatte YOLO-Tiny Probleme nah beieinander liegende Verkehrszeichen richtig zu klassifizieren und passende Begrenzungsrahmen zu schätzen. Dieses Problem könnte darauf zurückzuführen sein, dass YOLO das Eingangsbild in Zellen aufteilt und pro Zelle nur eine Klasse richtig klassifizieren kann.

9 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, unterschiedliche Objekterkennungsalgorithmen zu betrachten und eine Verkehrszeichenerkennung mit diesen zu realisieren. Es sollte eine durch einen Laptop gesteuerte autonome Fahrt eines Modellfahrzeugs ermöglicht werden, welches eine beliebige Teststrecke abfahren und durch Objekterkennungsalgorithmen auf Verkehrszeichen reagieren kann. Die eingesetzten Methoden sollten getestet und bewertet werden.

Die Umsetzung dieser Aufgabe begann mit der Analyse des Stands der Technik und der Auseinandersetzung mit den neuesten Objektdetektionssystemen. Es fand eine Einarbeitung in die Programmierbibliothek OpenCV statt. Programmierkenntnisse in Python und C++ konnten erweitert werden. Kenntnisse im Bereich des maschinellen Lernens und der computergestützten Objekterkennung in Bildern wurden sich angeeignet. Da sich dazu entschieden wurde, eine autonome Fahrt durch ein KNN zu realisieren, wurde die Theorie dahinter verstanden und der Umgang mit der Programmierbibliothek Tensorflow erlernt. Außerdem mussten die maschinellen Lernalgorithmen der beiden genutzten Objektdetektionssysteme verstanden werden, um ein Training auf die Erkennung von Verkehrszeichen zu ermöglichen.

Das Ergebnis dieser Arbeit ist ein Fahrzeug, welches sich über einen Controller steuern lässt. Zudem ist das Fahrzeug in der Lage autonom eine Teststrecke abzufahren. Realisiert wurde dies durch Training eines MLP, welches aus Kameradaten des Fahrzeugs entsprechende Fahrmanöver erzeugen kann. Eine Generalisierung auf verschiedene Strecken ist dem trainierten MLP derzeit nicht möglich. Die Gründe dafür und ein Weg zum Erreichen einer Generalisierung wurden in Kapitel 8.3 aufgezeigt. Zur Verkehrszeichendetektion wurden zwei auf maschinellem Lernen basierende Objektdetektionssysteme verwendet und bewertet, welche unterschiedlich gute Ergebnisse lieferten. Diese Ergebnisse sind zum einen auf die unterschiedlichen Trainingsdatensätze zurückzuführen, aber auch auf die Ansätze der genutzten Verfahren. Das erste zur Verkehrszeichendetektion verwendete Haar-Kaskaden-Klassifikator-Verfahren ermöglicht es drei verschiedene Verkehrszeichen („Vorfahrt gewähren“-, „30er Zonen“-,„Stopp“-Schild) zu detektieren. Es liefert schnelle Bildverarbeitungsraten und eine gute Detektionsqualität von Verkehrszeichen mit wenig Anspruch auf Speicheressourcen. Jedoch hat das Verfahren bei der Unterscheidung ähnlicher Verkehrszeichen Probleme. Zudem kann es weiter entfernte Verkehrszeichen schlechter detektieren. Durch ein angepasstes Training können diese Probleme beseitigt werden (Kapitel 8.4). Das Verfahren hat Potential eine gute Verkehrszeichendetektion zu ermöglichen. Das zweite betrachtete Objektdetektionssystem namens YOLO-Tiny ermöglicht ein Detektieren von „Vorfahrt gewähren“- und „Stopp“-Schildern. Es liefert eine etwas langsamere Bildverarbeitungsrate eine schlechtere Detektionsqualität und benötigt zusätzlich viele Speicheressourcen (Kapitel 8.5). Die langsamere Bildverarbeitungsgeschwindigkeit von YOLO-Tiny im Vergleich zu den Haar-Kaskaden-Klassifikatoren ist hier hervorzuheben, da YOLO-Tiny als eines der schnellsten CNN-Objektdetektionssysteme beschrieben wird. Es wurde zudem festgestellt, dass sich das Verfahren weniger für eine Verkehrszeichendetektion eignet, da es durch seinen anderen Ansatz der Objektdetektion schlecht kleine Objekte lokalisieren und klassifizieren kann.

Im Laufe der Erstellung dieser Arbeit haben sich durch die gesammelten Erkenntnisse einige Erweiterungsmöglichkeiten aufgetan:

- Der aktuelle Aufbau des Fahrzeugs ermöglicht es nur drei verschiedene Fahrtrichtungen einzuschlagen, da die Lenkung des Fahrzeugs durch einen DC-Motor realisiert ist. Dieser könnte durch einen Servo-Motor ersetzt werden um mit dem Fahrzeug leichter Trainingsdaten Sammeln zu können und auch das Befahren komplexerer Strecken zu ermöglichen.
- Für die Generierung von Fahrmanövern aus Kameradaten könnte anstatt eines MLP eine CNN-Architektur verwendet werden, da diese durch ihre Faltungsoperationen mit trainierten Filtern bessere Rückschlüsse aus zweidimensionalen Daten ziehen können.
- Um die Detektionsqualität eines Haar-Kaskaden-Klassifikators zu verbessern und auf die eigenen Bedürfnisse anzupassen, sollten eigene Trainingsbilder durch die auf dem Fahrzeug verbaute Kamera gesammelt werden. Diese Trainingsbilder sollten alle in Kapitel 4.3 genannten Eigenschaften beinhalten.
- In den heutigen Benchmark-Wettbewerben erreichen nur noch Zusammenschlüsse von verschiedenen Objektdetektionssystemen eine Top-Platzierung, deshalb sollte auch ein Blick auf andere Verfahren gelegt werden. Das HOG-Verfahren in Kombination mit einem SVM-Klassifikator sollte näher betrachtet werden. Außerdem liefert das auf einer CNN-Architektur basierende Faster-RCNN sehr gute Ergebnisse bei der Verkehrszeichendetektion [41].
- Zur Realisierung einer Reaktion des Fahrzeugs auf Verkehrszeichen, sollte ein Algorithmus entwickelt werden, der robust gegen falsch-positive Detektionen ist. Zum Beispiel könnte dies durch Einbeziehung der Position des detektierten Verkehrszeichens ermöglicht werden.

Literaturverzeichnis

[1] Zheng Wang: *Self Driving RC Car*. https://zhengludwig.wordpress.com/projects/self-driving-rc-car/. – Besucht am 26.02.2017

[2] Ryan Zotti: *Self-Driving-Car*. https://github.com/RyanZotti/Self-Driving-Car. – Besucht am 26.02.2017

[3] James Bowman, Patrick Mihelich: *How to Calibrate a Monocular Camera*. http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration. – Besucht am 27.02.2017

[4] Vieri Failli: *Biologisches Neuron*. http://image.redbull.com/rbx00516/0001/0/795/9999/451/fileadmin/user_upload/images/3_-_Structure_of_a_neuron_DE_01.jpg. – Besucht am 27.02.2017

[5] Paul Balzer: *Kuenstliches Neuron*. http://www.cbcity.de/wp-content/uploads/2016/03/Neuronales_Netz_Layer-770x578.png. – Besucht am 27.02.2017

[6] Andrej Karpathy, Justin Johnson: *CS231n: Convolutional Neural Networks for Visual Recognition*. http://cs231n.github.io/. – Besucht am 27.02.2017

[7] John Langford: *Loss functions*. https://github.com/JohnLangford/vowpal_wabbit/wiki/Loss-functions. – Besucht am 27.02.2017

[8] Sebastian Ruder: *An overview of gradient descent optimization algorithms*. http://sebastianruder.com/optimizing-gradient-descent/. – Besucht am 27.02.2017

[9] Navneet Dalal, Bill T.: *Histograms of Oriented Gradients for Human Detection*. 2005

[10] Lowe, David G.: *Distinctive Image Features from Scale-Invariant Keypoints*. 2004

[11] Herbert Bay, Luc Van G. Tinne Tuytelaars T. Tinne Tuytelaars: *SURF: Speeded Up Robust Features*. 2006

[12] Corinna Cortes, Vladimir V.: *Support-Vector Networks*. 1995

[13] Alex Krizhevsky, Geoffrey E. H. Ilya Sutskever S. Ilya Sutskever: *ImageNet Classification with Deep Convolutional Neural Networks*. 2012

[14] Christian Szeged, Yangqing Jia Pierre Sermanet Scott Reed Dragomir Anguelov Dumitru Erhan Vincent Vanhoucke Andrew R. Wei Liu L. Wei Liu: *Going Deeper with Convolutions*. 2015

[15] Kaiming He, Shaoqing Ren Jian S. Xiangyu Zhang Z. Xiangyu Zhang: *Deep Residual Learning for Image Recognition*. 2015

[16] Houben, Sebastian ; Stallkamp, Johannes ; Salmen, Jan ; Schlipsing, Marc ; Igel, Christian: Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark. In: *International Joint Conference on Neural Networks*, 2013

[17] Shaoqing Ren, Ross G. Kaiming He H. Kaiming He ; Sun, Jian: *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2015

[18] Wei Liu, Dumitru Erhan Christian Szegedy Scott Reed Cheng-Yang Fu Alexander C. B. Dragomir Anguelov A. Dragomir Anguelov: *SSD: Single Shot MultiBox Detector*. 2015

[19] Joseph Redmon, Ross Girshick Ali F. Santosh Divvala D. Santosh Divvala: *You Only Look Once: Unified, Real-Time Object Detection*. 2015

[20] Paul Viola, Michael J.: *Rapid Object Detection using a Boosted Cascade of Simple Features*. 2001

[21] Constantine Papageorgiou, Tomasu P.: *A Trainable System for Object Detection*. 2000

[22] opencv dev team: *haarfeatures*. http://docs.opencv.org/2.4/_images/haarfeatures.png. – Besucht am 3.03.2017

[23] Crow, Franklin C.: *Summed-Area Tables for Texture Mapping*. 1984

[24] Evans, Alex: *Four Tricks for Fast Blurring in Software and Hardware*. http://www.gamasutra.com/features/20010209/figure3.gif. – Besucht am 3.02.2017

[25] Yoav Freund, Robert E. S.: *A Short Introduction to Boosting*. 1999

[26] Stallkamp, Johannes ; Schlipsing, Marc ; Salmen, Jan ; Igel, Christian: The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In: *IEEE International Joint Conference on Neural Networks*, 2011, S. 1453–1460

[27] *Imagenet*. http://image-net.org/index. – Besucht am 11.03.2017

[28] Rainer Lienhart, Vadim P. Alexander Kuranov K. Alexander Kuranov: *Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection*. 2002

[29] CS231n, Stanford U.: *Spatial Localization and Detection*. http://cs231n.stanford.edu/slides/winter1516_lecture8.pdf. – Besucht am 6.03.2017

[30] Guanghan Ning: *Start Training YOLO with Our Own Data*. http://guanghan.info/blog/en/my-works/train-yolo/. – Besucht am 6.03.2017

[31] pgigioli: *Darknet ROS*. https://github.com/pgigioli/darknet_ros. – Besucht am 6.03.2017

[32] D. H. Hubel, T. N. W.: *RECEPTIVE FIELDS, BINOCULAR INTERACTION AND FUNCTIONAL ARCHITECTURE IN THE CAT'S VISUAL CORTEX*. 1961

[33] *Convolution Demonstration*. http://cs231n.github.io/assets/conv-demo/index.html. – Besucht am 6.03.2017

[34] *MAX Pooling*. http://cs231n.github.io/assets/cnn/maxpool.jpeg. – Besucht am 6.03.2017

[35] *CNN Architetecture*. http://cs231n.github.io/assets/cnn/convnet.jpeg. – Besucht am 6.03.2017

[36] *Localization with Regression.* https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/more_images/LocalizationRegression2.png. – Besucht am 6.03.2017

[37] *Objectdetection with R-CNN.* https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/more_images/RCNNSimple.png. – Besucht am 6.03.2017

[38] *YOLO workflow.* https://4.bp.blogspot.com/-wpLrlSGc46I/V3g5g4jLZGI/AAAAAAAAF3M/8GmP8H6xioE32gpODHfxbI0lb-pS9jjQQCKgB/s1600/YOLO.png. – Besucht am 6.03.2017

[39] *Darknet configurations.* https://github.com/pjreddie/darknet/tree/master/cfg/yolov1. – Besucht am 6.03.2017

[40] Chung Yu Wang, Royce Cheng-Yue: *Traffic Sign Detection using You Only Look Once Framework.* 2016

[41] Zhe Zhu, Songhai Z. Dun Liang L. Dun Liang: *Traffic-Sign Detection and Classification in the Wild.* 2016