Fakultät für Elektrotechnik

Bachelorarbeit

Studienrichtung

Bachelor Mechatronik

Christian Scheglmann

Autonomes Fahren mit Deep Learning

Verfasser der Bachelorarbeit
Christian Scheglmann
Meraner Str. 32 b
86356 Neusäß
bachelorarbeit@scheglmann.me

Erstprüfer: Prof. Dr.-Ing. Friedrich Beckmann

Thema erhalten am: 10.02.2017

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences
An der Hochschule 1
D-86161 Augsburg
Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Abstract

Entwickelt wurde ein selbstfahrendes Modellauto, welches nur mit Hilfe einer Kamera autonom durch Gänge und auf Wegen fahren kann. Das auf ein Convolutional Neural Network basierende System bestimmt mit einem Kamerabild die Hauptrichtung des Ganges in Relation zum Blickfeld des Fahrzeuges. Es können drei Zustände erkannt werden, ob der Weg „nach links gerichtet“, „nach rechts gerichtet“ oder „geradeaus gerichtet“ ist.

Durch überwachtes maschinelles Lernen wurde ein neuronales Netzwerk trainiert. Der hierfür benötigte Datensatz wurde mit drei Kameras aufgenommen, während das Auto manuell gesteuert durch den Gang fuhr. Die Ausrichtungen der Kameras entsprachen den drei zu klassifizierenden Zuständen.

Das Hauptaugenmerk liegt auf der Integrierung der verschiedenen Komponenten zu einem autonomen Roboter. Diese umfassen den Aufbau des Fahrzeuges, die Erstellung eines geeigneten Datensatzes, das Konfigurieren und Lernen eines CNNs und das Testen des Autos.

Inhaltsverzeichnis

1. EINLEITUNG 1

2. HINTERGRUND UND VERWANDTE ARBEITEN 4

2.1 CONVOLUTIONAL NEURAL NETWORKS 4

2.2 GRUNDLEGENDE KONZEPTE 5

2.2.1 Convolution 5

2.2.2 Pooling 6

2.2.3 Lernen 7

2.2.4 ReLU 7

2.2.5 Dropout 8

2.2.6 Softmax 9

2.2.7 Architekturen 9

2.3 WEITERER EINSATZ FÜR DEEP LEARNING 10

3. EIGENE UNTERSUCHUNG UND FORSCHUNG 11

3.1 PLATTFORM 12

3.1.1 Arduino 13

3.1.2 Nvidia Jetson TX1 15

3.1.3 Fernsteuern des Autos 16

3.2 DATENSATZ 18

3.2.1 Datensatz Umgebungen 19

3.2.2 Erzeugung der Bilddaten 20

3.2.3 Kamera 21

3.2.4 Probleme und Besonderheiten bei der Erzeugung 22

3.2.5 Qualität der Daten 22

3.2.6 Nachbearbeitung 23

3.3 TRAINING UND TOOLS 25

3.3.1 GPU-Server 25

3.3.2 Caffe 25

3.3.3 Digits 26

3.3.4 Training 28

3.4 INFERENCE 30

3.4.1 Bibliotheken 30

3.4.2 Code 31

3.4.3 Regelung 31

3.4.4 Geschwindigkeit des Klassifizierens 33

4. EXPERIMENTE UND ERGEBNISSE 34
4.1 FAHREN IN BEKANNTER UMGEBUNG 34
4.2 FAHREN IN UNBEKANNTER UMGEBUNG 35
4.3 LERNEN NEUER UMGEBUNGEN 36
5. ZUSAMMENFASSUNG 38
LITERATURVERZEICHNIS 39

Abbildungsverzeichnis

Abbildung 1 Architektur des LeNet-5 Netzwerkes (Quelle: [8]) 4
Abbildung 2 Veranschaulichung der Convolution Ebene 5
Abbildung 3 Beispiel von gelernten Filter der ersten Ebene 6
Abbildung 4 Max-Pooling Beispiel 7
Abbildung 5 Vergleich von Aktivierungsfunktionen 8
Abbildung 6 Dropout im neuronalen Netzwerk... 9
Abbildung 7 Vier Hauptbereiche 11
Abbildung 8 Aufbau der Fahrzeugplattform.. 12
Abbildung 9 Zusammenhang der Fahrzeugkomponenten 13
Abbildung 10 I/O Platine in der Herstellung beim Fräsen und als Layout............................. 14
Abbildung 11 Codeausschnitt Arduino, read_Serial(void) 15
Abbildung 12 NVIDIA Jetson TX1 auf Developer Kit 16
Abbildung 13 App-Anbindung zum manuellen Fahren... 17
Abbildung 14 Klassen des Datensatzes 18
Abbildung 15 Draufsicht von Gang1 19
Abbildung 16 Draufsicht des Weges draußen 19
Abbildung 17 Kameraaufbau 20
Abbildung 18 Beispiele aus dem Datensatz... 21
Abbildung 19 Blickfeld der Kameras 21
Abbildung 20 Blickfeld in einer Kurve 22
Abbildung 21 Problematik beim Spiegeln der Bilder.. 23
Abbildung 22: Python Codeausschnitt - Videoframes extrahieren.. 24
Abbildung 23: Python Codeausschnitt - Frames spiegeln 24
Abbildung 24 Prinzip des Trainierens und Ausführen eines Netzwerkes 27
Abbildung 25 DIGITS – Visualisierung der Aktivierung eines Bildes in der ersten Ebene eines Netzes... 28
Abbildung 26 Architektur des AlexNet (Abbildung ähnelt einer in [3])................................. 29
Abbildung 27 Geschwindigkeitsvergleich von Caffe und TensorRT (Informationen von [27]) 30
Abbildung 28 Energieverbrauch Caffe vs. TensorRT (Informationen von [27]) 31
Abbildung 29 Problem des Abdriftens 32
Abbildung 30 Provoziertes Abdriften um Wände zu erkennen 32

Abbildung 31 Gegenüberstellung der Testumgebungen... 34
Abbildung 32 Fahren des Autos 35

Tabellenverzeichnis

Tabelle 1 Fahr- und Steuersignale 14

Tabelle 2 Jetson TX1 Technische Spezifikationen (Quelle: [24]) 16

Tabelle 3 Spezifikation des GPU-Servers 25

Tabelle 4 Regelung des Lenkwinkels 32

Abkürzungsverzeichnis

Ai	artificial intelligence
API	application programming interface
App	Applikation
C	Programmiersprache
C++	Programmiersprache
CNC	Computerized Numerical Control
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
FP16	floating point 16
fps	frames per second
Full HD	Full High Definition
GPU	graphics processing unit
I/O	Input/output
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
KI	künstliche Intelligenz
LMDB	Lightning Memory-Mapped Database
mAh	Milli Ampere Stunden
mp4	MPEG-4 – Video-Containerformat
png	Portable Network Graphics
PWM	Pulsweitenmodulation
SGD	stochastic gradient descent
TFLOP	Tera Floating Point Operations Per Second
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus

1. Einleitung

In den letzten Jahren haben künstliche neuronale Netzwerke zahlreiche Wettbewerbe in der Mustererkennung und dem maschinellen Lernen gewonnen [1]. Einer der größten Erfolge gelang dem Google DeepMind AlphaGo Programm mit dem Sieg in dem Brettspiel „Go" Anfang 2016 gegen den Südkoreaner Lee Se-dol, einer der besten Spieler auf der Welt [2]. Gerade weil das Spiel „Go" um ein Vielfaches mehr an spielbaren Varianten im Vergleich zu Schach besitzt und somit der Spielverlauf nicht vorhergesagt werden kann, sondern nach Erfahrung und Intuition gespielt werden muss, zeigt dieser Sieg die außerordentliche Leistung von künstlicher Intelligenz.

Doch nicht jede künstliche Intelligenz (KI) ist gleich. Der Begriff wurde 1956 auf der Dartmouth Konferenz geformt. Ein Team von Computerwissenschaftlern hatte den Traum, eine komplexe Maschine zu entwerfen - ermöglicht durch Computer - welche die gleichen Charakteristiken wie die menschliche Intelligenz besitzt.
Über die Jahre hinweg machte die künstliche Intelligenz eine Evolution durch, von vorprogrammierten Entscheidungsbäumen über *reinforcement learning* bis hin zum jetzigen Stand, dem *Deep Learning*. Eine Untergruppe der künstlichen Intelligenz sind maschinelle Lernalgorithmen, mit denen aus einer großen Menge an statistischen Daten Vorhersagen zu unbekannten Daten getroffen werden können. Zu dieser Untergruppe gehört das *Deep Learning*. *Convolutional Neural Network*, oft durch CNN abgekürzt, ist eine Variante von *Deep Learning* und wird für die Klassifizierung von Bildern eingesetzt.

Doch erst seit 2012 haben neuronale Netzwerke und *Deep Learning* eine überlegende Rolle in dem Feld der künstlichen Intelligenz und des maschinellen Lernens eingenommen. Dies ist auf die Forschung von Alex Krizhevsky und seinem Team der Universität von Toronto zurückzuführen. Sie entwarfen ein CNN zur Klassifizierung von Bildern und trainierten dieses anhand des ImageNet Datensatzes, welcher aus über 15 Millionen in 22.000 Klassen eingeteilten Bildern besteht. Erst durch den Einsatz von leistungsfähigen Graphikkarten mit einer effizienten Implementierung zur Berechnung von Faltungsmatrizen konnte die Trainingszeit des Netzwerkes erheblich reduziert werden. Bei der Teilnahme des „ImageNet Large-Scale Visual Recognition Challenge" (ILSVRC-2012) gewann das Team um Krizhevsky

mit einem erheblichen Vorsprung gegenüber traditionellen Methoden der Bildverarbeitung und Bildklassifizierung. [3]

Aus dem Fortschritt der KI resultierten viele technische Fortschritte wie E-Mail Spam Filter, Bild-, Text- oder Sprachverarbeitung. Firmen wie Pinterest nutzen neuronale Netzwerke zur Klassifizierung von Bildern, Facebook zur Gesichtserkennung und Netflix für Filmvorschläge [2]. Aber auch auf medizinischem Gebiet wird geforscht. Dank der Fähigkeit, Muster und Objekte zu erkennen, wird an der Universität von Hong Kong an der frühzeitigen Erkennung von Krebs gearbeitet. In Tumorbildern können betroffene Zellen durch ein neuronales Netzwerk segmentiert werden. Laut Hao Chen, Doktorand und Mitglied des Forschungsteams, sei das Trainieren eines neuronalen Netzwerks mit GPUs 100-mal schneller als auf CPUs. Gerade im Laufe der weiteren Forschung ist dieser Geschwindigkeitsvorsprung wichtig. [4]

Ende 2015 wurde ein wissenschaftlicher Bericht des „Dalle Molle Institute for Artificial Intelligence“ und der Universität Zürich über eine autonom fliegende Drohne veröffentlicht, die mit Hilfe eines CNNs Wald- und Wanderwege abfliegen kann. Das Team hat hierfür mit drei Kameras, eine nach vorne gerichtet, die anderen beiden um 30° nach links und rechts versetzt, Bilder von Wanderwegen gesammelt. Anhand dieser wurde das Netzwerk trainiert. Daraufhin konnte das aktuelle Kamerabild der Drohne klassifiziert werden, um die Richtung des Weges zu bestimmen. Diese drei Klassen ermöglichen der Drohne, autonom zu fliegen. Weitere Tests haben ergeben, dass das Netzwerk zufällige Testbilder genauer klassifizieren kann, als ein Mensch. [5]

Aufgabenstellung dieser Bachelorarbeit ist es, ein autonomes Modellauto zu entwickeln, welches eigenständig mit Hilfe einer Kamera durch einen Gang fahren kann. Das Prinzip aus [5] soll auf das Auto übertragen und untersucht werden, ob mit dieser Methode ein autonomes Fahren realisierbar ist. Zu den Aufgaben gehört das Erstellen eines Datensatzes, das Auslegen eines *Convolutional Neural Network* für die Klassifizierung des Kamerabildes, das Trainieren des Netzwerkes, das Ausführen und Klassifizieren auf der zuvor gebauten Fahrzeugplattform und das Evaluieren und Testen des Fahrverhalten. Ebenso wird auf aufgetretene Probleme und passende Lösungen eingegangen.

Im Folgenden wird die Struktur dieser Arbeit beschrieben.

Zuerst wird auf die Geschichte und Grundlagen von *Convolutional Neural Networks* eingegangen. Ebenso werden mathematische Berechnungen und Algorithmen erläutert, um ein Grundverständnis in die Thematik zu geben.
Danach erfolgt die Auseinandersetzung mit der eigentlichen Thematik. Schrittweise wird auf Komponenten eingegangen, welche für die Bearbeitung der Aufgabenstellung essentiell sind:

- Die Erzeugung des Datensatzes wird ausführlich beschrieben. Angewandte Methoden, Probleme und Optimierungen werden aufgeführt.
- Die für das Lernen des Netzwerkes verwendete Software, Tools und Hardware wird in dem nächsten Abschnitt behandelt.
- Die einzelnen Elemente, welche zusammen die Fahrzeugplattform ergeben, werden beschrieben.
- Das Ausführen des Netzwerkes auf dem Fahrzeug, der dazu benötigte Code und die Regelung werden erläutert.
- Für verschiedene Experimente, um die Performanz zu verbessern und neue Umgebungen zu lernen, werden die Ergebnisse zusammengefasst.

Zuletzt werden eine Zusammenfassung der gesamten Arbeit und ein Ausblick auf mögliche Verbesserungen gegeben.

2. Hintergrund und verwandte Arbeiten

In diesem Kapitel wird zuerst auf die Geschichte von *Convolutional Neural Networks* eingegangen, gefolgt von einem groben Überblick über die grundlegenden Konzepte und mathematischen Zusammenhänge in einem CNN.

2.1 Convolutional Neural Networks

Convolutional Neural Networks sind biologisch inspirierte Architekturen, welche verschiedene Eigenschaften lernen können. Pioniere auf dem Gebiet waren Hubel und Wiesel, welche 1962 die primäre Sehrinde von Katzen untersuchten und daraufhin ein tiefgehendes Model entwickelten [6]. Der Filterkern dieses Models ähnelt einfachen Zellen im lokalen rezeptivem Feld der Netzhaut und die Poolingschichten ähneln komplexen Zellen. Das erste auf einem Computer simulierte CNN wurde in Fukushima's Neocognitron vorgestellt [7]. Von Yann LeCun wurde 1990 LeNet entwickelt, ein Netzwerk welches durch die Einführung von Gradienten basierender Stochastik die Entwicklung voranbrachte. LeNet wurde zur Dokumenten- und Handschrifterkennung eingesetzt [8]. Die Architektur ist in Abbildung 1 zusehen. Der Eingang ist ein $32x32$ großes Bild und der Ausgang ist ein Vektor. Im Netzwerk sind verschiedene nichtlineare Ebenen vorhanden.

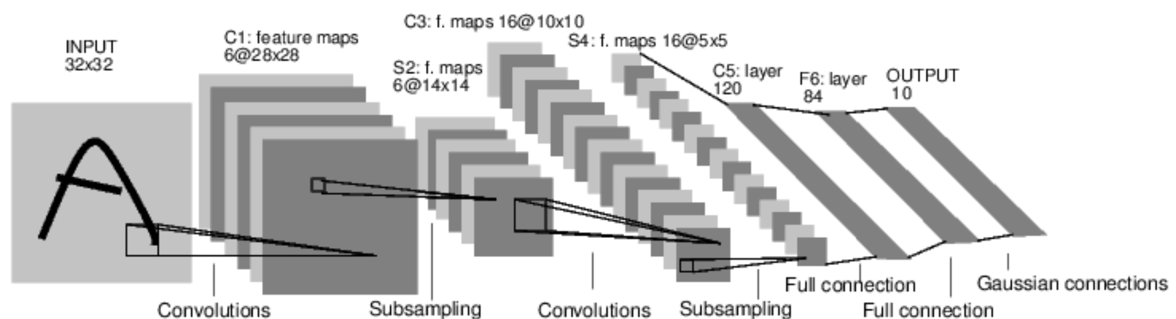

Abbildung 1 Architektur des LeNet-5 Netzwerkes (Quelle: [8])

Erst seit 2012 wurden CNNs durch das AlexNet populär. Drei wichtige Gründe für den Erfolg von *Convolutional Neural Networks* sind: 1) steigende Rechenleistung aufgrund des Moore's Law, insbesondere in heutigen Graphikkarten, 2) mehr vorhandene Trainingsdaten und 3) neuere, bessere und effizientere Trainingsalgorithmen.

In dem Bereich der Klassifizierung von Bildern erzielt ein neuronales Netzwerk in vielen Fällen ein besseres Ergebnis, als im Vergleich zum Menschen. Beschrieben wird dies im Falle der Klassifizierung von Wald- und Wanderwegen in [5].

2.2 Grundlegende Konzepte

Im Folgenden wird ein Überblick über grundlegende Konzepte gegeben, welche in *Convolutional Neural Networks* angewandt werden. Ein CNN besteht aus verschiedenen Ebenen, welche wiederum unterteilt werden können. Diese bestehen aus *Convolutions*, Aktivierungsfunktion und Max-Pooling. Abbildung 2 zeigt eine *Convolution* Ebene mit drei Stufen. Jede Operation ist in einem der nächsten Teilabschnitte beschrieben.

2.2.1 Convolution

Die *Convolutional* Ebene ist der Kernbestandteil eines *Convolutional Neural Network*, welche auch die meiste Rechenleistung benötigt. *Convolution* ist die Faltung zweier Matrizen. Es ist eine lineare Rechenoperation, welche als Eingang eine variable Größe verarbeiten kann.

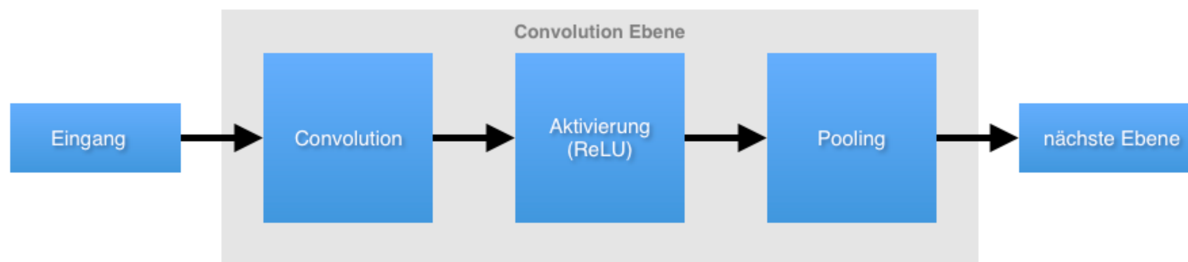

Abbildung 2 Veranschaulichung der Convolution Ebene

Die Abbildung 2 veranschaulicht den Aufbau einer Ebene in einem neuronalen Netzwerk, welche aus folgenden Stufen besteht: 1) *Convolution*, 2) *ReLU*, 3) *Max-Pooling*. Normalisierung wurde hier nicht angewandt. Diese Abbildung ähnelt einer, welche in Quelle [9] gezeigt ist.

Im Bereich des maschinellen Lernens ist der Eingang eine meist mehrdimensionale Matrix, ein Bild. Der Filterkern ist eine mehrdimensionale Matrix von Werten, welche durch den Lernalgorithmus angepasst werden. Mathematisch ist die Rechenoperation wie folgt ausgedrückt:

$$s[i,j] = (I\ x\ K)[i,j] = \sum_{m}\sum_{n} I[m,n]K[i-m,j-n] \qquad (1)$$

I ist der Eingang, also das Bild mit den Dimensionen m und n, K ist der Filterkern und s ist der resultierende Ausgang der Funktion [9].
Anders ausgedrückt enthält das Convolutional Layer lernbare Filter. Jeder Filter ist im Vergleich zu dem Eingangsbild relativ klein, so kann zum Beispiel ein Filter der ersten Ebene eine Größe von 5x5x3 besitzen (jeweils 5 Pixel in der Höhe und Breite, und 3 Pixel in der Tiefe, resultierend von den drei Farbkanälen). Während das Netzwerk vorwärts ausgeführt wird, also dem Klassifizieren, wird jeder Filter über die Höhe und Breite des Eingangsbildes geschoben, genauer gesagt gefaltet und das Skalarprodukt zwischen dem Bild und jeder Position des Filters gebildet. Als Ausgang wird eine zweidimensionale Aktivierungsmatrix, die *activation map*, gebildet. [10]

Abbildung 3 Beispiel von gelernten Filter der ersten Ebene

In Abbildung 3 sind Beispielfilter zu sehen. Diese sind durch ein trainiertes GoogLeNet entstanden. Auffällig ist die Ähnlichkeit zu den Gabor Filtern, welche in der Bildverarbeitung zur Kantenerkennung verwendet werden.

2.2.2 Pooling

Pooling-Schichten fassen benachbarte Ausgänge von Neuronen zu einem Wert der Faltungsmatrix zusammen. Der Autor von [11] zeigt, dass Pooling überlegen zu anderen Methoden wie zum Beispiel dem *subsampling* ist. Eine Pooling-Schicht ist ein Gitter, bestehend aus Pooling Elemente, welche s Pixel voneinander entfernt sind. In einem Bereich von z x z, mittig zum Pooling Element, werden benachbarte Pixel zusammengefasst. Traditionell überlappen sich die Pooling-Bereiche nicht, dies kann durch $s = z$ beschrieben werden, zu sehen in [12] [13] [14]. Doch die Arbeit um das Team von Alex Krizhevsky [3] zeigt, dass Netzwerke mit überlappenden Pooling-Bereichen weniger zu *overfitting* neigen, dies ist der Fall, wenn $s < z$ ist.

Zum Zusammenfassen der benachbarten Pixel stehen verschiedene statistische Möglichkeiten zur Verfügung, so kann zum Beispiel der Durchschnitt eines Pixelbereiches berechnet werden. Die am weitesten verbreitete Variante ist jedoch das Max-Pooling, bei der der höchste Wert des Pixelbereiches übernommen wird, zu sehen in Abbildung 4.

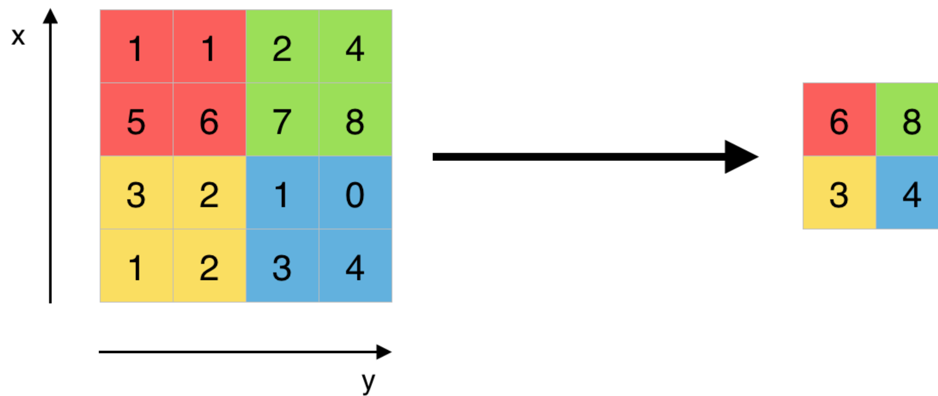

Abbildung 4 Max-Pooling Beispiel

2.2.3 Lernen

Um das Lernen eines Neurons zu ermöglichen, wird eine *cost function* benötigt. Diese berechnet den Unterschied der Gewichtung des Neurons zum Optimum. In neuronalen Netzwerken kann dies durch *stochastic gradient descent* (SGD) umgesetzt werden. Die Standard Form der SGD-Funktion schaut wie folgt aus:

$$\theta^{k+1} = \theta^{k} - \epsilon_k \frac{\partial L(\theta^k, z)}{\partial \theta^k} \quad (2)$$

Hierbei sind θ die Lernparameter und ϵ ist die Lernrate.

2.2.4 ReLU

In Abbildung 5 zu sehen sind die Aktivierungsfunktionen *sigmoid*, *tanh* und *ReLu*. In Blau ist die *sigmoid*-Funktion dargestellt:

$$f(x) = (1 + e^{x})^{-1} \quad (3)$$

Diese bringt die Aktivierung eines Neurons in den Bereich von 0 bis 1. Große negative Werte werden 0 und große positive Werte werden 1.
Die *tanh*-Funktion ist in Grün gezeichnet:

$$f(x) = tanh(x) \quad (4)$$

Diese formt Werte zu einem Bereich zwischen -1 und 1 um.
Eine bewährte Aktivierungsfunktion ist die *ReLU* (*Rectified Linear Unit*) [15], in Abbildung 5 in Rot dargestellt:

$$f(x) = max(0, x) \quad (5)$$

Diese ist im Vergleich zu den beiden anderen oben genannten Funktionen nicht sättigend, dies bedeutet, dass ein Neuron lernt, sobald ein positiver Wert anliegt. Wie in [3] beschrieben ist, wird die Trainingszeit eines Netzwerkes durch die *ReLU*-Funktion deutlich verringert.

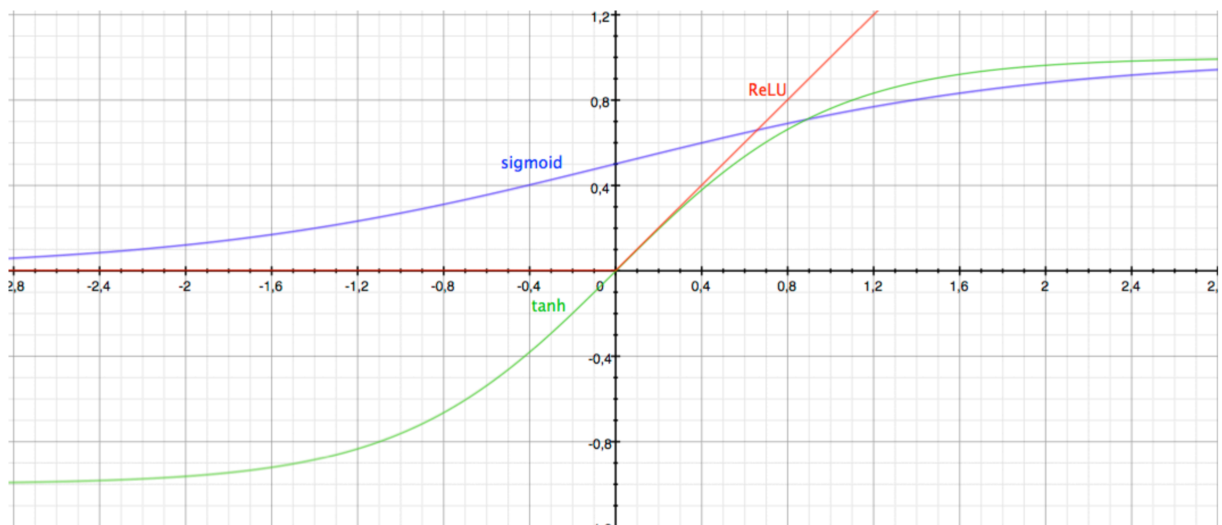

Abbildung 5 Vergleich von Aktivierungsfunktionen

2.2.5 Dropout

Unter *Dropout* bezeichnet man eine gängige Methode um *overfitting* zu verhindern. Insbesondere in Netzwerken mit vielen Parametern ist *overfitting* ein Problem. Deshalb werden in der Trainingsphase zufällig die Ausgänge von versteckten Neuronen mit einer festgelegten Wahrscheinlichkeit auf null gesetzt. Daraus resultierend werden in jeder Iteration des Lernens verschiedene Neuronen des nun verdünnten Netzwerkes trainiert, wodurch ein geringerer Verallgemeinerungsfehler entsteht und robustere Merkmale gefunden werden [16]. So wird auch im AlexNet die Regularisierungsmethode *Dropout* verwendet, da sie sich als effizient erwiesen hat. Die Wahrscheinlichkeit für einen *Dropout* wurde beim AlexNet auf 0.5 gesetzt. Eine *Dropout* Rate von 0.5 (50%) verdoppelt die Anzahl an Iterationen zum Lernen, reduziert

jedoch *overfitting* [3]. Das GoogLeNet verwendet im Vergleich eine Wahrscheinlichkeit von 70% für *Dropouts* [17].

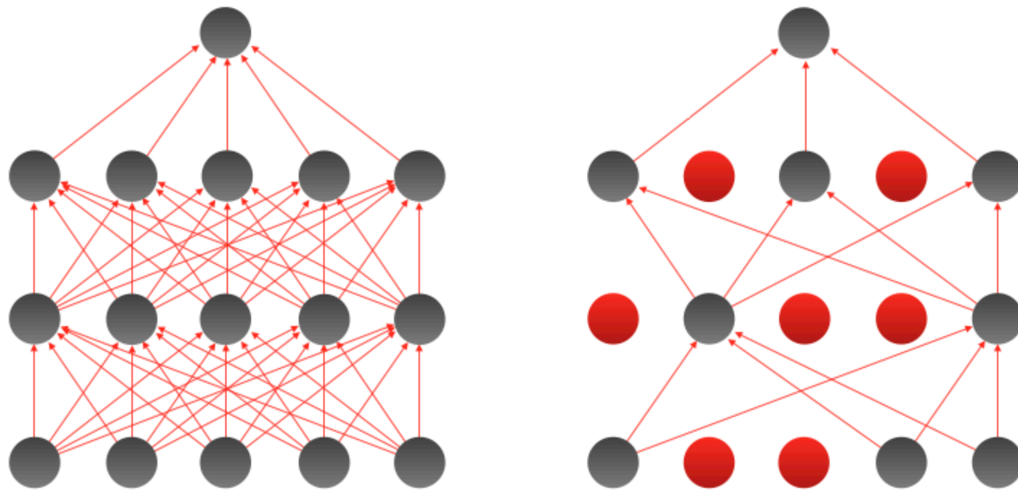

Abbildung 6 Dropout im neuronalen Netzwerk

In Abbildung 6 ist auf der linken Seite ein neuronales Netz mit zwei versteckten Ebenen zu sehen, auf der rechten Seite ein Beispiel zu einem durch *Dropout* verdünnten Netzwerk. Die Ausgänge der roten Neuronen wurden auf null gesetzt.

2.2.6 Softmax

Die letzte Ebene in einem Netzwerk zur Klassifizierung von Bildern ist typischerweise eine *Softmax* Ebene. Diese wird verwendet, um einen K-dimensionalen Vektor z zu einem K-dimensionalen Vektor σ(z) mit Werten zwischen 0 und 1 zu normalisieren. Das Ergebnis der Funktion beschreibt die Wahrscheinlichkeit eines Bildes, zu einer Klasse j zu gehören.

$$\sigma(z)_j = \frac{e_j^z}{\sum_{k=1}^{K} e_k^z} \tag{6}$$

mit $j = 1, \dots, K$. Wegen dem exponentiellen Anteil in der Gleichung neigt die Klasse mit der höchsten Wahrscheinlichkeit zu einem großen Wert, oft bei 0.9 oder 1.0.

2.2.7 Architekturen

Convolutional Networks bestehen für gewöhnlich aus drei Ebenen-Arten: *Convolution* (CONV), *Pooling* (POOL) und *Fully-connected Layer* (FC). Durch eine Wiederholung dieser Reihenfolge wird ein *Convolutional Neural Network* aufgebaut:

$$INPUT -> [[CONV -> RELU] * N -> POOL?] * M -> [FC -> RELU] * K -> FC$$

2.3 Weiterer Einsatz für Deep Learning

Gerade weil *Deep Learning* in Bild- und Sprachverarbeitung große Erfolge erzielt, werden *Deep Learning* Algorithmen in vielen Bereichen kommerzialisiert. Diese werden zum Beispiel zum Auswerten von Bank-Checks, Videoüberwachung, Gesichtserkennung [12], Personen und Schildererkennung im Straßenverkehr [18] oder off-road Roboter mit einer optischen Hinderniserkennung [19] verwendet. Motiviert von den zahlreichen Möglichkeiten wird an der Integrierung in vielen neuen Anwendungsbereichen geforscht.

3. Eigene Untersuchung und Forschung

Nach den Grundlagen wird in den folgenden Kapiteln auf die Aufgabenstellung der Einleitung eingegangen.
Ziel ist es, das Prinzip aus [5] auf ein Modellauto zu übertragen, so dass es autonom durch einen Gang oder auf einem Weg fahren kann. Es soll ein rohes Kamerabild verarbeiten werden und damit die Ausrichtung des Ganges zum Blickfeld des Fahrzeuges bestimmt werden.

Im Wesentlichen kann das Vorgehen in vier Schritte unterteilt werden, welche in Abbildung 7 dargestellt sind:

1) Erstellen einer Fahrzeugplattform
2) Erzeugen eines Datensatzes
3) Überwachtes maschinelles Lernen eines neuronalen Netzwerkes
4) Ausführen des Netzes auf dem Fahrzeug

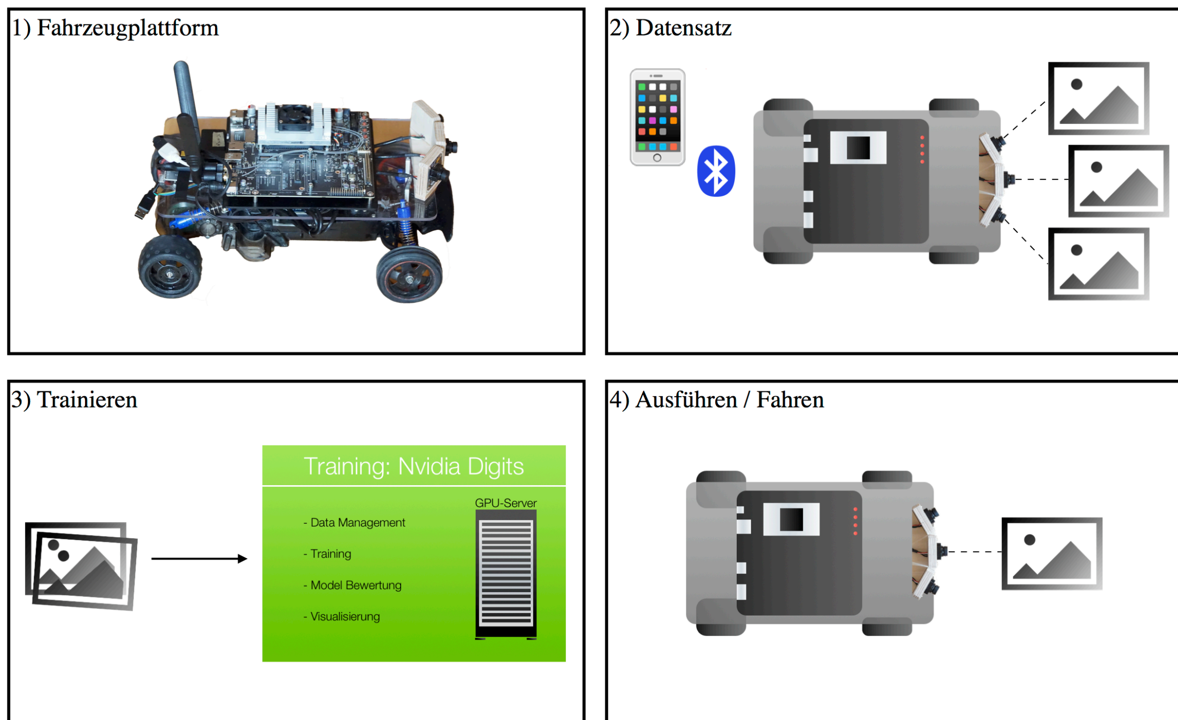

Abbildung 7 Vier Hauptbereiche

3.1 Plattform

Als fahrbare Basis dient ein Modellauto, an dem diverse Umbauten vorgenommen wurden, so dass es durch einen Computer gesteuert werden kann. In Abbildung 8 sind die wichtigsten Komponenten in der Seitenansicht des Fahrzeugs farbig markiert:

- Servo (Magenta)
- Arduino auf I/O Platine (Zyan)
- H-Brücke (Gelb)
- Akku (Blau)
- Elektromotor (Weiß)
- USB Hub (Braun)
- Nvidia Jetson TX1 (Rot)
- Kameramodule (Grün)

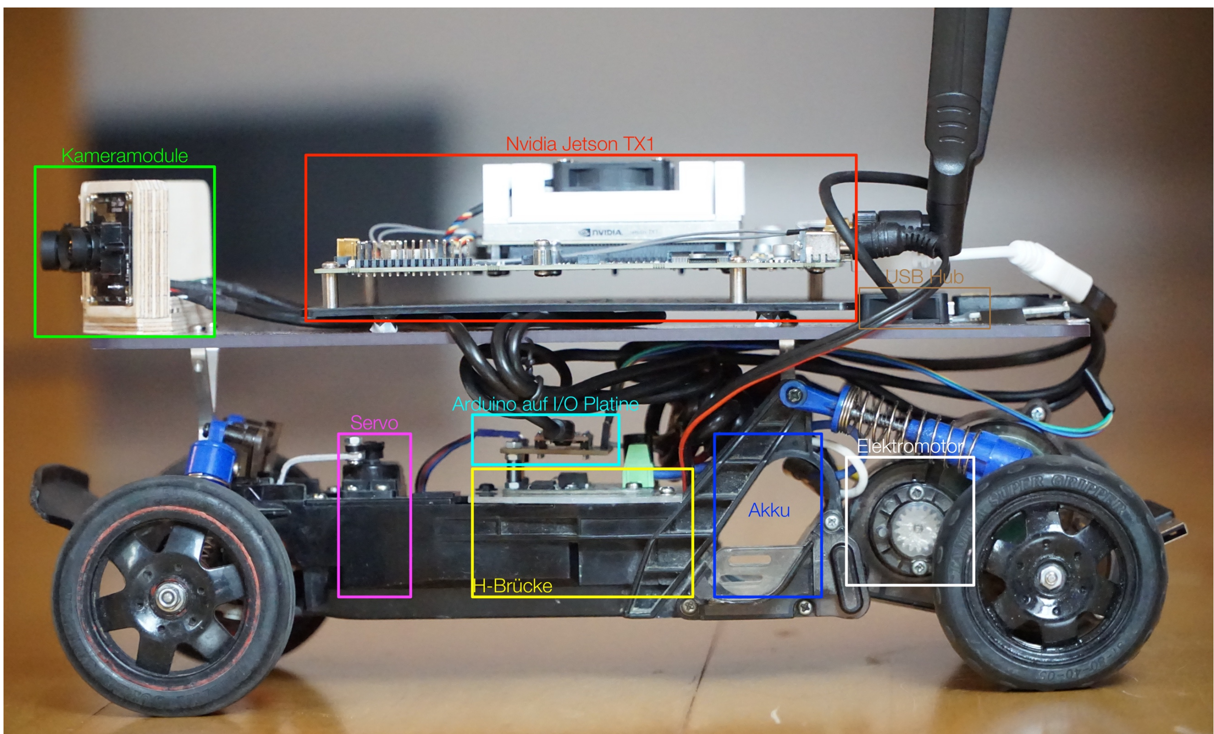

Abbildung 8 Aufbau der Fahrzeugplattform

Der Jetson TX1 Computer wertet ein Kamerabild aus, berechnet die Fahrsignale und sendet diese an den Arduino Mikrocomputer. Dieser steuert einen Servo für die Lenkung an und eine H-Brücke für den Gleichstrommotor. Eine Plattform aus Plexiglas wurde gefräst, auf der die Kameras im vorderen Teil, der Nvidia Jetson TX1 Computer mittig und ein USB-Hub im hinteren Teil des Fahrzeugs befestigt sind.

Alle elektronischen Komponenten werden von einem Lithium-Polymer-Akkumulator mit 11,1 Volt Spannung und 2200mAh Kapazität betrieben.

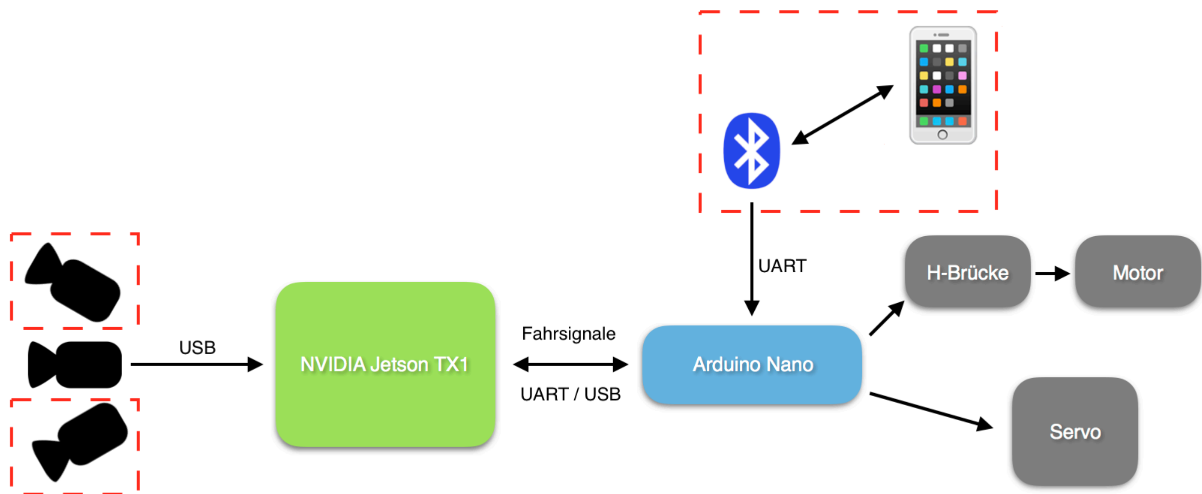

Abbildung 9 Zusammenhang der Fahrzeugkomponenten

Das Zusammenspiel der verschiedenen Fahrzeugkomponenten kann in Abbildung 9 nachvollzogen werden. Die in Rot umrandeten Elemente werden nur für die Erzeugung eines Datensatzes benötigt.

3.1.1 Arduino

„Arduino ist eine Open-Source Elektronik-Plattform, welche auf einer einfach zu benutzenden Hardware und Software basiert. Das Ziel ist es, jedem interaktive Projekte zu ermöglichen." [20] Programmiert werden Arduinos in den Sprachen C und C++. Die Arduino AG bietet eine Programmier- und Entwicklungsumgebung an, welche den einfachen Umgang mit den Mikroprozessoren ermöglicht. Es steht eine Vielzahl von verschiedenen Arduino Boards zur Verfügung, die sich durch ihre Größe, der Anzahl von Ein- und Ausgängen und ihrer Rechenleistung unterscheiden.
Für die hardwarenahe Ansteuerung der Plattform wurde ein Arduino Nano gewählt, da dieser klein, kostengünstig und ausreichend Ein- und Ausgänge bietet. Er basiert auf einem ATmega328 Mikroprozessor. Die Stromversorgung erfolgt über USB [21].

Der Arduino Nano wurde auf einer eigens entwickelten I/O Platine platziert. Diese bietet Anschluss für den Servo und die H-Brücke. Ebenfalls enthält die Platine eine UART-Schnittstelle, um einen Bluetooth Chip für das Fernsteuern des Fahrzeuges anschließen zu können. Diese Platine wurde auf einer CNC-Fräse gefertigt, zu sehen in Abbildung 10.

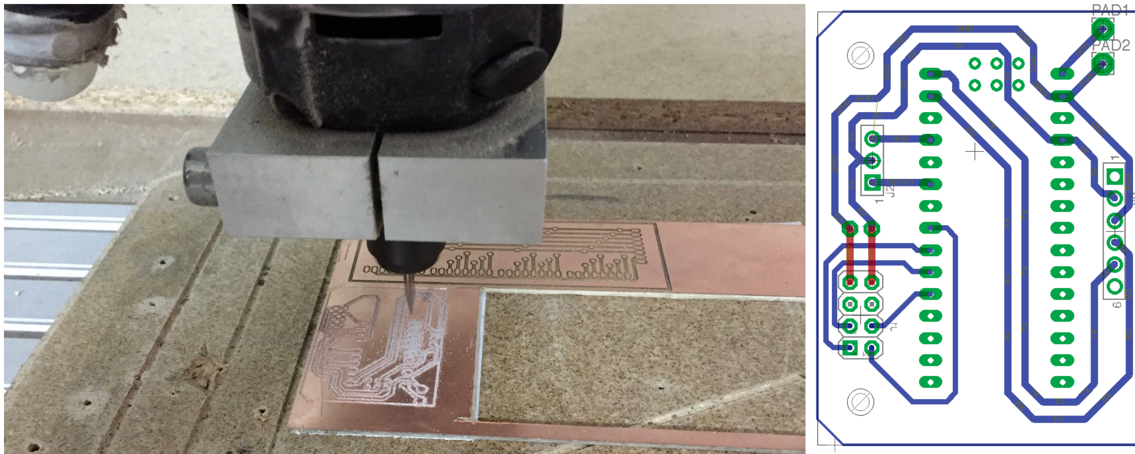

Abbildung 10 I/O Platine in der Herstellung beim Fräsen und als Layout

Der Servo, welcher für das Lenken zuständig ist und die H-Brücke zur Motoransteuerung, benötigen mehrere PWM-Signale. Ein Computer basierend auf einem Betriebssystem und Prozessen, wie der verwendete Jetson TX1, ist nicht in der Lage diese Signale in Echtzeit und ohne Unterbrechung zu generieren. Aus diesem Grund wurde auf einen Arduino Mikrocomputer zurückgegriffen, der mit dieser Aufgabe kein Problem hat.

Der Arduino kann Fahr- und Lenksignale über eine UART-Verbindung entgegennehmen und wandelt diese in Steuersignale beziehungsweise PWM-Signale für den Servo und die H-Brücke um.

Die seriellen Signale zum Fahren sind aus dem Buchstaben „F“ und einem Zahlenwert zwischen -255 und 255 aufgebaut. So werden zum Beispiel Fahrsignale in der Form von „F23“ übergeben, dies entspricht einem langsamen Fahren in Vorwärtsrichtung.

Zum Lenken wird der Buchstabe „L“ in Kombination mit einem Zahlenwert zwischen 45 und 135 an den Arduino übergeben. Geradeaus fahren entspricht „L90“, der Servo steht folglich bei 90 Grad. „L45“ wäre ein Vollausschlag nach links, also eine Servo Position von 45 Grad. Diese Signale sind als Übersicht in der Tabelle 1 angegeben.

Wenn der Arduino für mehr als 500 Millisekunden kein UART-Signal erhält, wird die Verbindung als unterbrochen angesehen und das Auto stoppt.

Tabelle 1 Fahr- und Steuersignale

	Art	Wert	Bemerkung
Lenken	L	45 - 135	„45“ ≙ links; „90“ ≙ gerade; „135“ ≙ rechts
Fahren	F	-255 - 255	„positiv“ ≙ vorwärts; „negativ“ ≙ rückwärts

Die Abbildung 11 zeigt die Funktion „void read_Serial(void)“, welche kontinuierlich in einer Schleife aufgerufen wird. Diese Funktion überprüft, ob ein neues serielles Signal vorhanden ist. Nachdem eine Zeitvariable gesetzt wurde, welche für die Überprüfung der Verbindung verwendet wird, wird die serielle Nachricht ausgelesen und die Funktionen „lenke“ oder „drive“ mit dem erhaltenden Parameter aufgerufen.

```
1  void read_Serial(void){
2    while(Serial.available() > 0){
3
4      lastTime = millis();  //Zeit zum berechnen des Time-Outs
5
6      char type;
7      String digit_read;
8      int digit;
9
10     type = Serial.read();
11
12     digit_read = Serial.readStringUntil('\n');
13     digit = digit_read.toInt();
14
15     switch (type){
16       //L -> Lenken
17       case 'L':
18         lenke(digit);
19         break;
20       //F -> Fahre
21       case 'F':
22         drive(digit);
23         break;
24     }
25   }
26 }
```

Abbildung 11 Codeausschnitt Arduino, read_Serial(void)

3.1.2 Nvidia Jetson TX1

Der Nvidia Jetson TX1 ist ein Embedded Supercomputer, der speziell für Computer Vision und *Deep Learning* entwickelt worden ist. Als Betriebssystem nutzt er Linux, liefert mit dem Tegra X1 Chip eine Rechenleistung von 1 TFLOP bei FP16 Genauigkeit und benötigt 10 Watt an Leistung.

Das scheckkartengroße Modul ist auf einem Entwicklerboard befestigt und ermöglicht hierdurch diverse Anschlussmöglichkeiten für Peripherien. In Tabelle 2 sind technische Details angegeben und in Abbildung 12 ist das Modul zu sehen.

Nvidia stellt zu dem Jetson TX1 passende Software bereit. Mit Hilfe des Entwicklertools „JetPack“ kann das Betriebssystem, Bibliotheken und APIs installiert werden. So wird auf dem Jetson „Linux4Tegra“ (L4T) installiert. Grundlegend ist dies Ubuntu 14.04 mit einem eigenen Kernel und vorkonfigurierten Treibern.

Tabelle 2 Jetson TX1 Technische Spezifikationen (Quelle: [24])

	Jetson TX1
GPU	NVIDIA Maxwell ™ , 256 CUDA cores
CPU	Quad ARM® A57/2 MB L2
Video	4K x 2K 30 Hz Encode (HEVC) 4K x 2K 60 Hz Decode (10-Bit Support)
Memory	4 GB 64 bit LPDDR4 25.6 GB/s
Display	2x DSI, 1x eDP 1.4 / DP 1.2 / HDMI
CSI	Up to 6 Cameras (2 Lane) CSI2 D-PHY 1.1 (1.5 Gbps/Lane)
PCIE	Gen 2 I 1x4 + 1x1
Data Storage	16 GB eMMC, SDIO, SATA
Other	UART, SPI, I2C, I2S, GPIOs
USB	USB 3.0 + USB 2.0
Conectivity	1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth
Mechanical	50 mm x 87 mm (400-Pin Compatible Board-to-Board Connector)

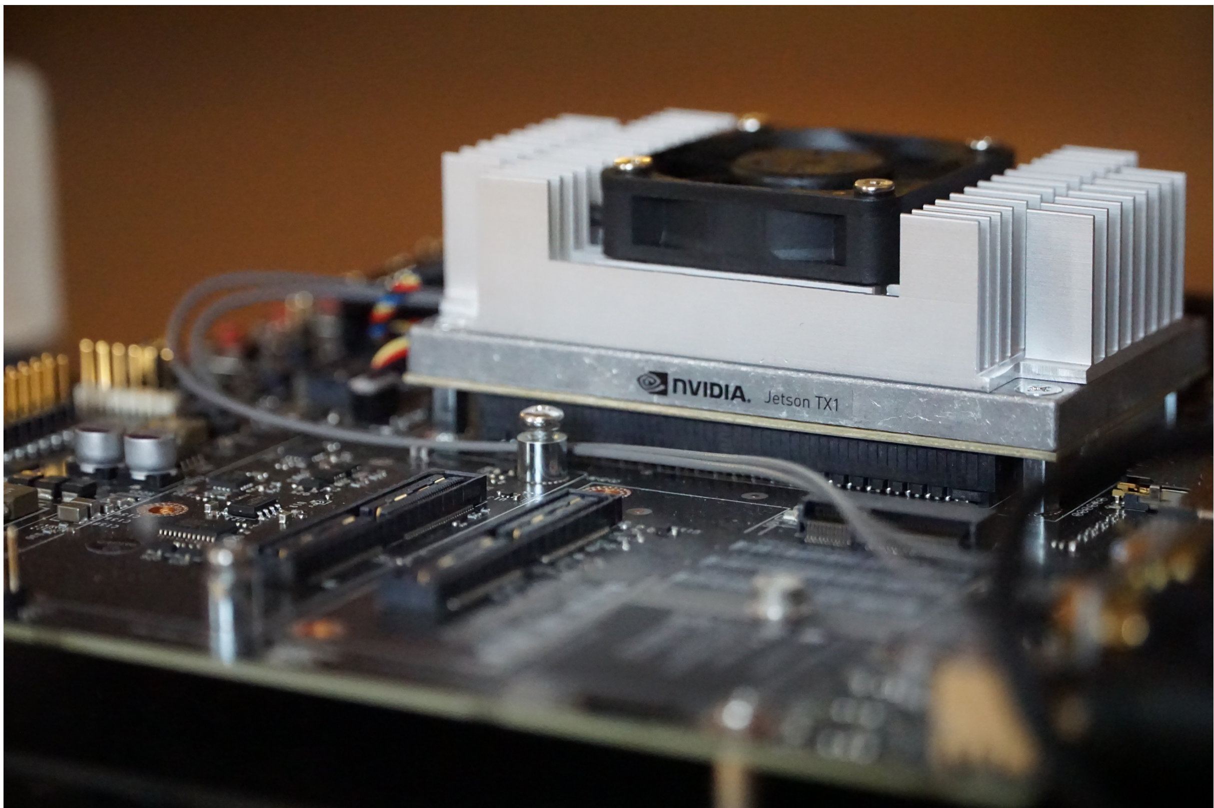

Abbildung 12 NVIDIA Jetson TX1 auf Developer Kit

3.1.3 Fernsteuern des Autos

Das Auto wird bei der Aufnahme der Trainingsbilder manuell gesteuert. An die Plattform können über eine serielle Verbindung Fahrsignale übertragen werden. Mit dem UART zu Bluetooth Adapter „HM-10“ der Firma „JNHuaMao Technology Company“ wurde eine

kostengünstige Lösung realisiert, um das Fahrzeug mittels einer iPhone App zu steuern. Um die Bluetooth Verbindung zwischen dem iPhone und dem HM-10 zu erstellen, wurden Teile des Beispielcodes aus [22] verwendet. Die fertige Applikation kann über einen Schieberegler die Geschwindigkeit des Fahrzeuges steuern und durch Neigen des iPhones kann gelenkt werden. Ein haptisches Feedback durch Vibration wird an den Nutzer der App übergeben, sobald das Smartphone einen Lenkwinkel von mehr als ±5 Grad überträgt und die Videoaufnahme gestoppt wird. Falls die Bluetooth Verbindung abbricht, hält das Fahrzeug aus Sicherheitsgründen an. In Abbildung 13 ist schematisch die Bluetooth Verbindung des iPhones zu dem HM-10 und die serielle Verbindung zur Fahrzeugplattform dargestellt.

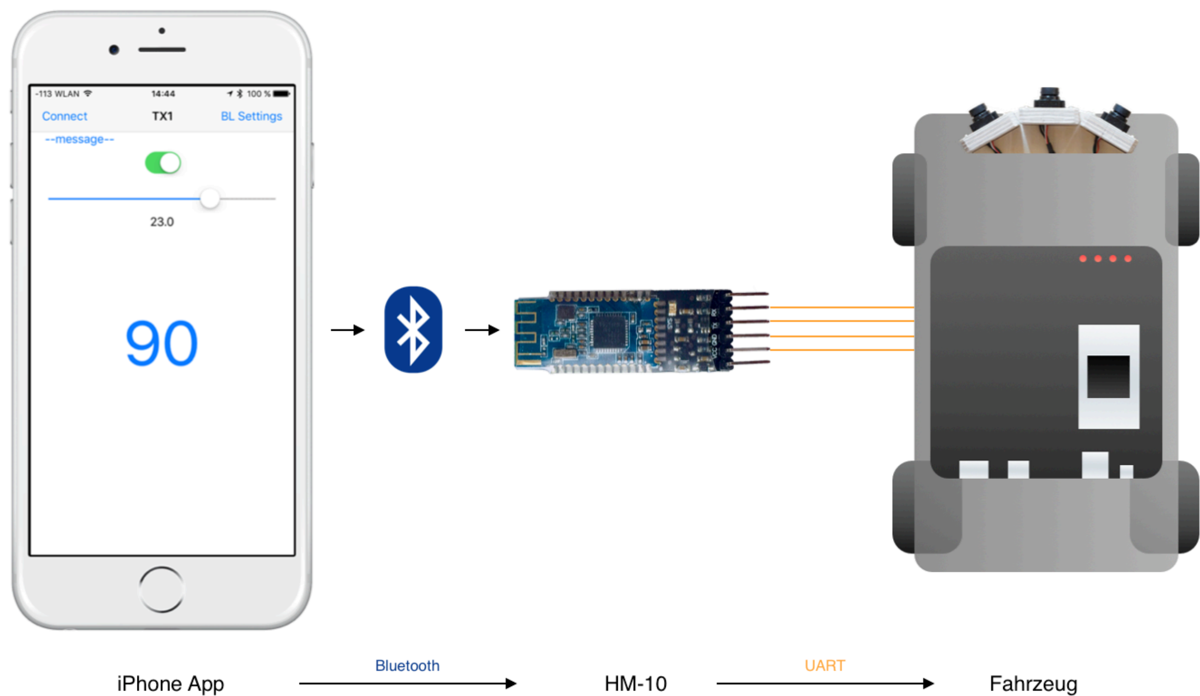

Abbildung 13 App-Anbindung zum manuellen Fahren

3.2 Datensatz

Wie Alex Krizhevsky in [3] beschreibt, benötigen *Convolutional Neural Networks* zum Trainieren eine große Anzahl an Daten, um beim Klassifizieren eine gute Performanz zu erzielen. Ebenso ist die Qualität des Datensatzes wichtig, so dass Algorithmen wichtige Merkmale lernen und diese auf unbekannte Daten angewandt werden können.

Das in dieser Bachelorarbeit untersuchte Konzept zur autonomen Steuerung eines Fahrzeuges beinhaltet im Kern die Zuordnung von Bildern einer Kamera in eine der drei folgenden Klassen:

- l: Das Fahrzeug ist relativ zur gewünschten Fahrtrichtung nach „links" gerichtet.
- r: Das Fahrzeug ist relativ zur gewünschten Fahrtrichtung nach „rechts" gerichtet.
- g: Das Fahrzeug ist zur gewünschten Fahrtrichtung „geradeaus" gerichtet.

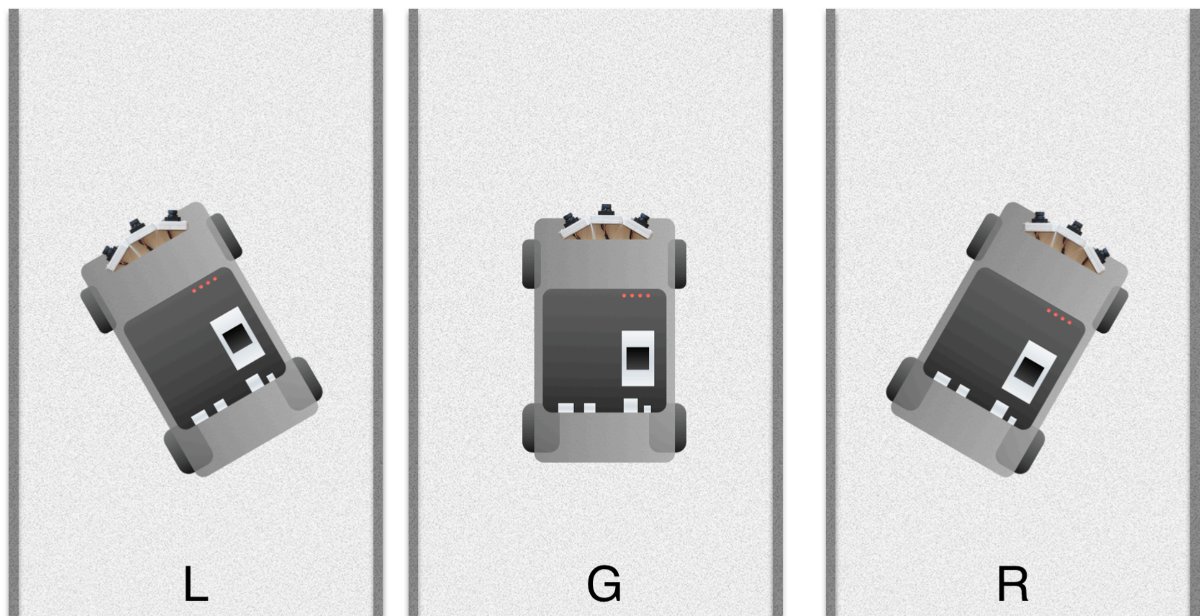

Abbildung 14 Klassen des Datensatzes

Das Lernen von CNNs wird zu dem Bereich des überwachten maschinellen Lernens eingeordnet. Für überwachtes Lernen werden repräsentative Trainingsdaten der drei möglichen Ausrichtungen des Autos benötigt, welche entsprechenden Labels (Klassennamen) zugeordnet sind. Der während dieser Arbeit aufgenommene Datensatz wird auf der Webseite http://bachelorarbeit.scheglmann.me/ zur Verfügung gestellt.

Im Folgenden wird das Erstellen und Sammeln der Bilddaten erklärt, welche Maßnahmen in der Nachbearbeitung vorgenommen wurden und wie die Qualität des Datensatzes gesteigert werden konnte.

3.2.1 Datensatz Umgebungen

Es wurden zwei Umgebungen ausgesucht, in denen Daten gesammelt wurden. Die erste Umgebung ist ein Gang, welcher in dieser Arbeit als „Gang1" referenziert wird. Dieser ist ein ca. zwei Meter breiter Gang, welcher wie ein Atrium aufgebaut ist. Jeder der vier Gangabschnitte ist 28 Meter lang. In Abbildung 15 ist die Draufsicht skizziert.

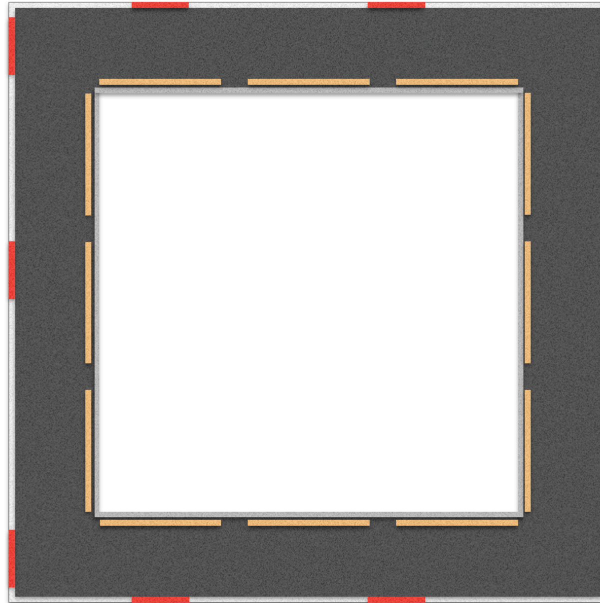

Abbildung 15 Draufsicht von Gang1

Die zweite Aufnahmeumgebung ist ein Weg außerhalb des Gebäudes, welcher „Weg draußen" genannt wird. Dieser unterscheidet sich wesentlich im Aussehen vom Gang1. Das Szenario besitzt auf einer Seite des Weges eine Wiese und auf der anderen Seite Bänke und eine Hecke. Der Weg draußen erstreckt sich über eine Länge von 80 Metern und ist zwei Meter breit, zu sehen in Abbildung 16. Bilder der beiden Aufnahmeumgebungen sind in Abbildung 18 gezeigt.

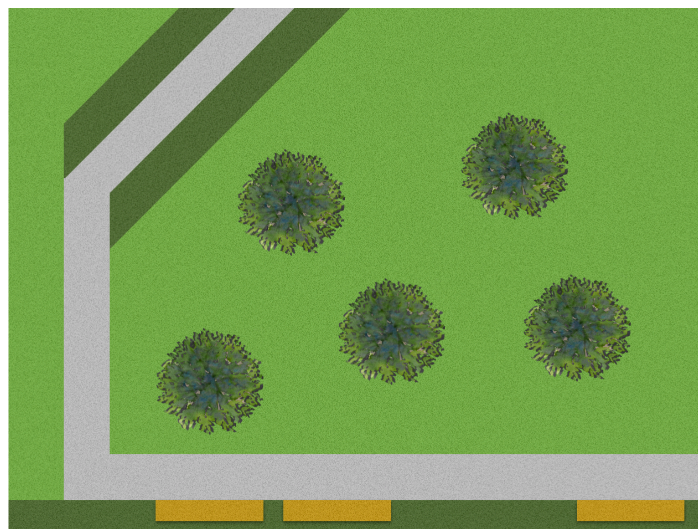

Abbildung 16 Draufsicht des Weges draußen

3.2.2 Erzeugung der Bilddaten

Um den Datensatz aufzunehmen, wurde das Fahrzeug manuell in den zwei zuvor beschriebenen Umgebungen gefahren, dem Gang1 und dem Weg draußen. Ausgestattet mit drei Kameras, nahm das Auto Trainingsbilder der drei Klassen auf.
Eine der Kameras ist in Fahrtrichtung ausgerichtet, die anderen beiden sind jeweils um 30° nach links und rechts versetzt. Dies ist in Abbildung 17 farblich eingezeichnet.

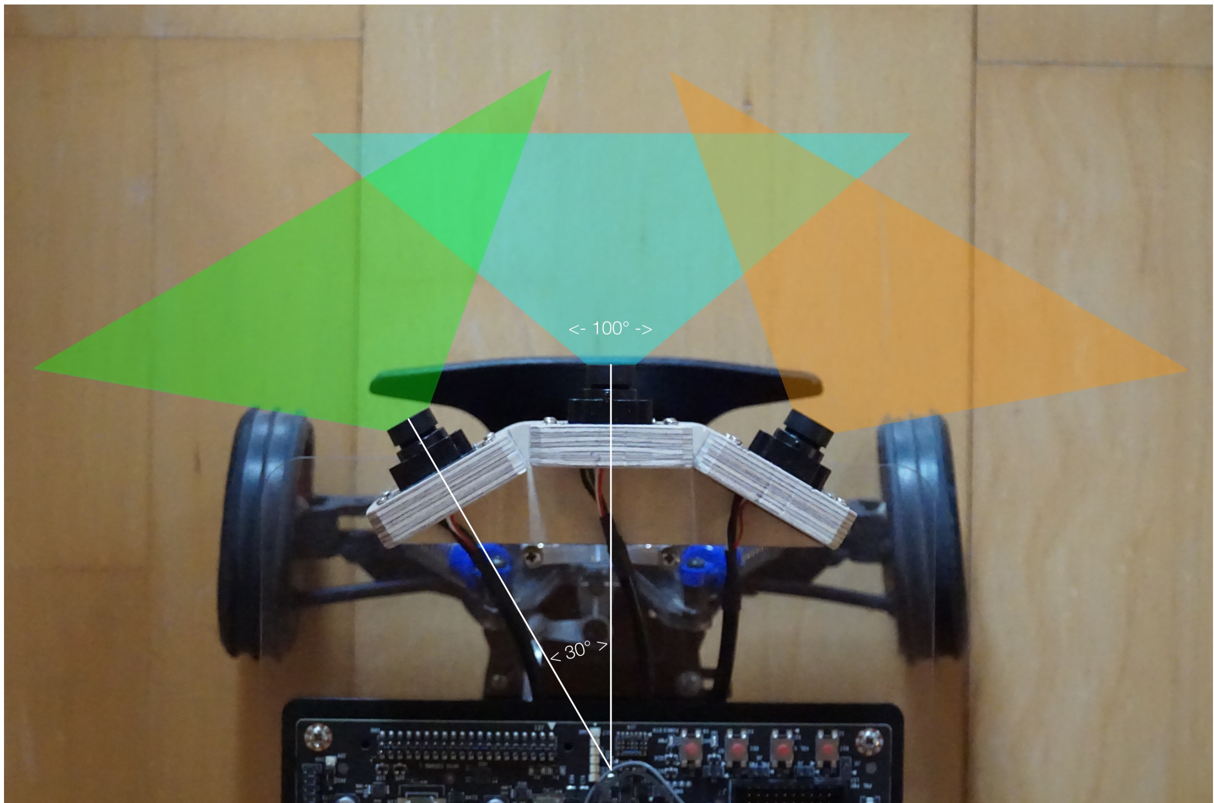

Abbildung 17 Kameraaufbau

Durch diesen Aufbau war es möglich, die Daten jeweils einer Kamera einer Klasse zuzuweisen. Insgesamt wurden 55.776 Bilder gesammelt, davon 33.787 Bilder im Gang1 und 21.989 Bilder auf dem Weg draußen. Im Gang1 wurden 17 Runden in dem Atrium gefahren, 1900 Meter, und auf dem Weg draußen wurden 550 Meter zurückgelegt.
Abbildung 18 zeigt einen Ausschnitt aus dem gesammelten Datensatz mit Bildern der drei Klassen.

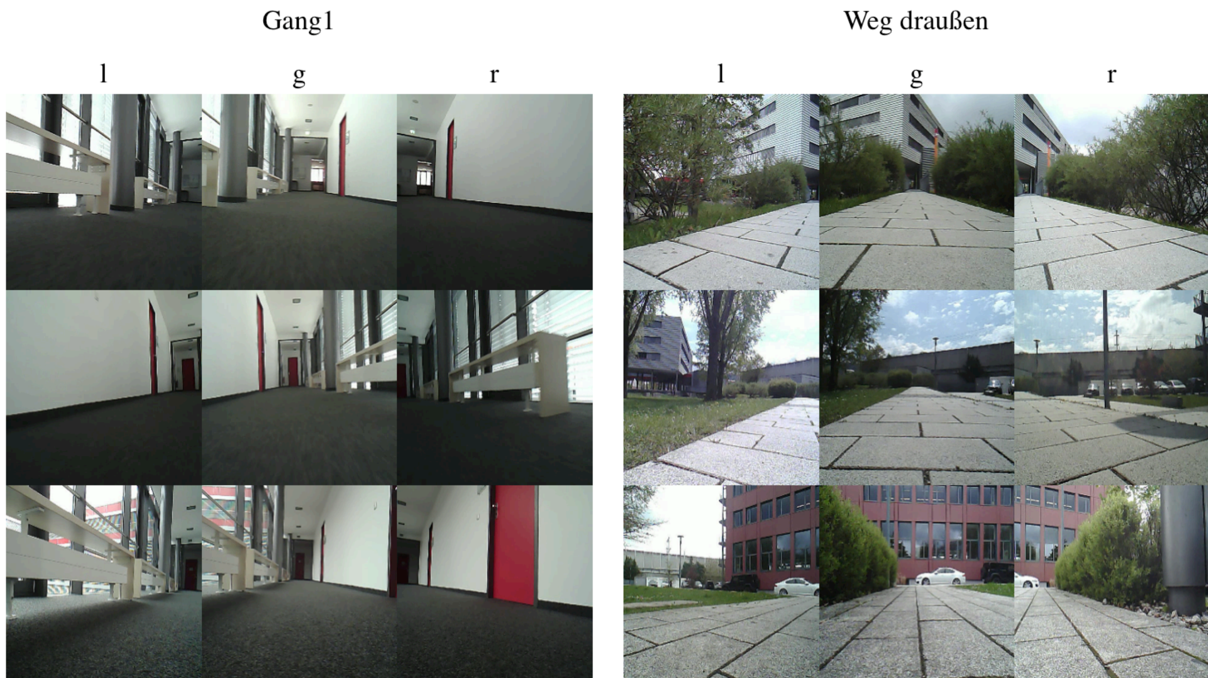

Abbildung 18 Beispiele aus dem Datensatz

In Abbildung 19 ist eine Überlagerung der Kamerabilder zu sehen. Durch einen farbigen Rahmen sind die einzelnen Kameras gekennzeichnet. Das Bild der „linken Kamera“ beinhaltet hauptsächlich die Wand, die „rechte Kamera“ die Fensterfront und Heizkörper und die „gerade Kamera“ den Gang.

Abbildung 19 Blickfeld der Kameras

3.2.3 Kamera

Zum Sammeln der Trainingsdaten wurden drei Webcams der Firma ELP mit einem CMOS OV2710 Sensor und 100° Objektiv ohne Verzerrung verwendet. Diese haben eine Auflösung von 2.0 Megapixel und können in Full HD aufnehmen. Bei den Aufnahmen wurde eine

Auflösung von 640x480 Pixeln gewählt, um die Speichergröße gering zu halten, zumal das Netzwerk nur Bilder in der Größe von 256x256 Pixeln verarbeiten kann. Synchron wurde mit allen drei Kameras .mp4 Videos aufgenommen.

3.2.4 Probleme und Besonderheiten bei der Erzeugung

Eine Problematik beim Sammeln der Daten entsteht, wenn das Auto um eine Kurve fährt. In diesem Szenario, in Abbildung 20 zu sehen, nehmen die „linke Kamera“ und die „gerade Kamera“ Bilder der linken Wand auf, die „rechte Kamera“ nimmt geradeaus-Bilder auf. Folglich werden Bilder einer falschen Klasse zugeordnet. Um dieser Problematik und somit dem Aufzeichnen eines fehlerhaften Datensatzes entgegen zu wirken, wird der aktuelle Lenkwinkel ausgelesen. Sobald das Auto mehr als ±5° lenkt, wird die Videoaufnahme gestoppt, bis das Auto wieder geradeaus fährt. Daraus resultierend werden keine Kurven aufgenommen.

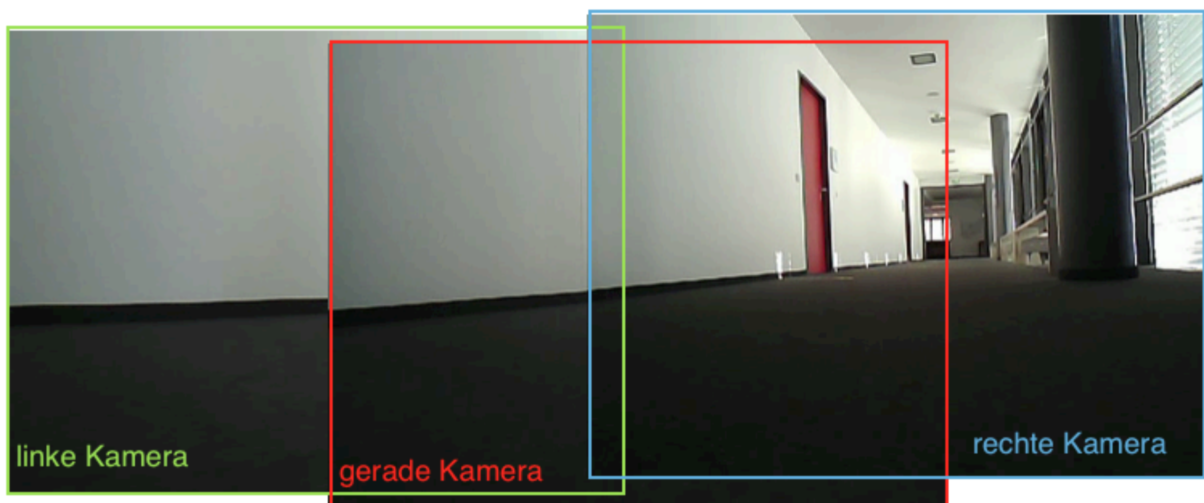

Abbildung 20 Blickfeld in einer Kurve

3.2.5 Qualität der Daten

Um einen qualitativ hochwertigen Datensatz zu erzeugen und somit ein gutes Lernergebnis zu erzielen, wurden verschiedene Maßnahmen getroffen.
So ist es wichtig, dass das Fahrzeug gerade durch den Gang gesteuert wird. Durch die Smartphone Applikation ist ein sehr genaues Steuern des Autos möglich, da kleine Korrekturen der Fahrtrichtung vorgenommen werden können.
Das zuvor beschriebene Problem, dass in Kurven falsche Daten gesammelt werden und deshalb die Aufnahme gestoppt wird, trägt ebenfalls zu einem besseren Datensatz bei.

Es wurde darauf geachtet, dass nicht nur mittig zum Gang gefahren wird, sondern auch nahe der Wand oder nahe der Fensterfront, um eine größere Variation der Bilder zu erstellen. Verschiedene Lichtverhältnisse, die in verschiedenen Teilen des Ganges vorhanden waren, tragen ebenfalls zu einem robusteren Netzwerk bei.

Es gibt auch andere Möglichkeiten, einen Datensatz zu erweitern, beziehungsweise während dem Trainieren die Daten programmatisch zu verändern. So werden oft die Bilder gespiegelt, rotiert, skaliert oder der Kontrast verändert.
Dies konnte bei den gesammelten Bildern nur bedingt angewandt werden. Problematisch ist zum Beispiel das Spiegeln eines Bildes während des Lernens. Dies ist bei der Klassifizierung von zum Beispiel einem Apfel kein Problem, aber in dem Szenario des Ganges verursacht dies eine Verfälschung des Datensatzes. Ein gespiegeltes Bild der Klasse „links“ kann nicht mehr seiner ursprünglichen Klasse zugeordnet werden, sondern gehört nun zur Klasse „rechts“, zu sehen in Abbildung 21. Deshalb wurden die Bilder des Datensatzes in der Nachbearbeitung gespiegelt und entsprechend der korrekten Klasse zugeordnet. Nach dem Erweitern des Datensatzes durch gespiegelte Bilder, wurde das Netzwerk trainiert.

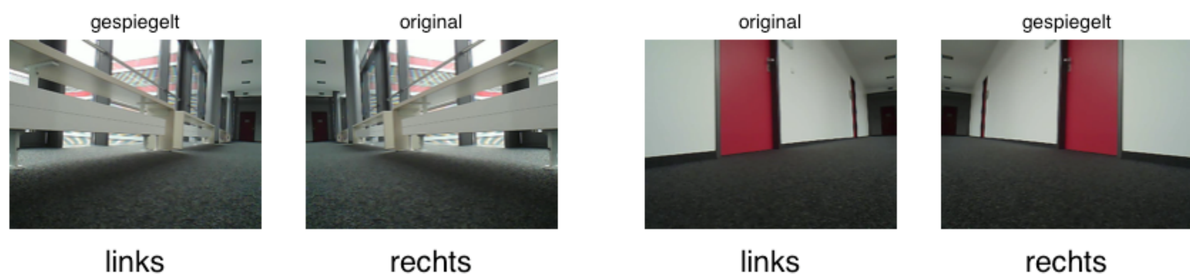

Abbildung 21 Problematik beim Spiegeln der Bilder

3.2.6 Nachbearbeitung

Der erste Schritt in der Nachbearbeitung ist das Extrahieren der einzelnen Frames aus den Videodateien. Hierfür wurde ein Pythonscript erstellt, welches die .mp4-Dateien öffnet und .png-Bilder erstellt. Mit Hilfe der OpenCV Library kann dies einfach realisiert werden. In Abbildung 22 ist der zuständige Codeausschnitt zu sehen.

```
30      while vidcap.isOpened():
31          success, image = vidcap.read()
32          if success:
33              #Prgressbar im Terminal
34              process = int((count/frameCount)*100)
35              progress(count, frameCount, suffix=count)
36              #Save frame
37              cv2.imwrite(os.path.join(path_output_dir, '%d.png') % count, image)
38              count += 1
39          else:
40              break
41      vidcap.release()
```

Abbildung 22: Python Codeausschnitt - Videoframes extrahieren

Das Script erstellt eine Ordnerstruktur, die Ordnernamen sind die jeweiligen Klassen beziehungsweise Labels. Die einzelnen Ordner beinhalten die Bilder der äquivalenten Klasse.

Nach dem Extrahieren werden die einzelnen Bilder durch ein weiteres Script gespiegelt, der zuständige Programmcode ist in Abbildung 23 abgebildet. Für die gespiegelten Bilddateien des Ordners „l" wird ein Ordner „l-flipped" angelegt und äquivalent wird für die anderen Klassen dasselbe gemacht. Die Dateien von „l-flipped" müssen anschließend in den Ordner „r" kopiert werden. Nach demselben Prinzip werden die verbleibenden Klassen erweitert.

```
20  def flip_png(input_dir, output_dir, totalx):
21      countnow = 0
22      os.makedirs(output_dir)
23      for subdir, dirs, files in os.walk(input_dir):
24          for file in files:
25              filepath = subdir + os.sep + file
26              if filepath.endswith(".png"):
27                  progress(countnow, totalx, suffix=file)
28                  img = cv2.imread(os.path.join(subdir, file))
29                  img = cv2.flip(img,1)
30                  cv2.imwrite(os.path.join(output_dir, '%s-flipped.png') % os.path.splitext(file)[0], img)
31                  countnow += 1
```

Abbildung 23: Python Codeausschnitt - Frames spiegeln

Durch das Spiegeln verdoppelt sich die Anzahl an Trainingsbildern auf 111.552 Bilder. Auf einem GPU-Server wird aus dem Ordner mit den Trainingsbildern eine Datenbank erstellt, so dass effizient und schnell mit diesen gearbeitet werden kann. Diese Datenbank hat eine Größe von 10.2 GB.

3.3 Training und Tools

In den folgenden Abschnitten werden verwendete Tools, Softwareumgebungen und Frameworks beschrieben und Hintergrundinformationen gegeben, welche zum Trainieren des Netzwerkes verwendet wurden.

3.3.1 GPU-Server

Das Trainieren eines CNNs ist sehr rechenintensiv, gerade die Berechnung der Faltung benötigt viel Leistung. Wie die Literatur [3] beschreibt, ist das Trainieren eines CNNs auf Grafikkarten im Vergleich zu CPUs um ein vielfaches schneller. Mit Hilfe aktueller Grafikkarten mit CUDA Unterstützung und cuDNN Libraries installiert, erhält man einen Geschwindigkeitsvorteil bei der Berechnungszeit von Faktor 17.
Deshalb wurde das Trainieren des Netzwerkes auf einem GPU-Server ausgeführt. Die wichtigsten Spezifikationen des Servers sind in Tabelle 3 zu finden.

Tabelle 3 Spezifikation des GPU-Servers

Prozessor	2 x Intel Xeon Broadwell E5-2690v4 (2.6 GHz, 14 Core)
Speicher	128 GB DDR4 RAM mit 2400 MHz
Graphik	4 x NVIDIA Geforce GTX 1080 mit GP104 Pascal, 2560 Cores, 8 GB RAM

3.3.2 Caffe

Caffe ist ein modifizierbares Framework für *Deep Learning Algorithmen*. Entwickelt wurde es von dem „Berkeley Vision and Learning Center“ (BVLC). Das Framework ist eine C++ Library mit Python und MATLAB Einbindung für das Trainieren und Ausführen von *Convolutional Neural Networks* und anderen *Deep Learning* Modellen. [23]
Durch die Benutzung von CUDA beziehungsweise cuDNN ist eine schnelle GPU-Berechnung möglich.
CUDA (Compute Unified Device Architecture) ist eine Programmier-Technik für parallele Berechnungen, entwickelt von NVIDIA. Es verwendet die Rechenleistung von Nvidia Grafikkarten. [24]
Die „NVIDIA CUDA® Deep Neural Network library“ (cuDNN) ist eine GPU-beschleunigte Bibliothek, speziell für neuronale Netzwerke. Verschiedene Standardroutinen und Berechnungen wie zum Beispiel *forward* und *backward convolution*, *pooling*, *normalization*, und *activation layers* sind effizient implementiert. [25]

Caffe wurde mit Hinblick auf Modularität entwickelt. So können neue Datenformate, Netzwerk Layer oder *loss functions* erweitert werden. Modellkonfigurationen sind im „Google Protocol buffer format“ definiert, ein menschenlesbares Textformat.

3.3.3 Digits

Das „NVIDIA Deep Learning GPU Training System“ (DIGITS) ist ein *Deep Learning* Entwicklungstool. Es kann zum Trainieren von *Deep Neural Networks* (DNNs) für Bildklassifizierung, Segmentation und Objektdetektierung verwendet werden. Es vereinfacht häufig angewandte Aufgaben, wie das Verwalten von Daten, Designen und Trainieren von neuronalen Netzwerken auf Multi-GPU-Systemen, visuelle Echtzeitüberwachung der Performanz und Evaluieren eines trainierten Netzwerkes. [26]
Über eine Weboberfläche wird auf DIGITS zugegriffen, dies ermöglicht ein plattformübergreifendes Arbeiten. Es ist auf cuDNN und Caffe aufgebaut und wurde im März 2015 veröffentlich. Zum Stand vom April 2017 gibt es DIGITS in der fünften Version, die aktiv von NVIDIA gepflegt wird. Für eine einfache Installation wurde DIGITS in einem Docker-Image auf dem GPU-Server installiert.
Die wichtigsten Eigenschaften zusammengefasst sind:

- Visualisierung von DNN Topologien und Aktivierungen von Testdaten im Netzwerk
- Verwalten des Trainierens mehrerer Netzwerke auf einem Multi-GPU System
- Einfaches konfigurieren und starten des Lernens
- Unterstützung einer großen Anzahl an Bildformaten und Quellen
- Echtzeitvisualisierung des Trainings
- Open source, erweitern und modifizieren der Software ist möglich

Zum Stand des Schreibens dieser Arbeit ist DIGITS an Bilddaten gebunden, in späteren Versionen sollen aber noch weitere Datentypen unterstützt werden.

Nvidia bietet mit DIGITS einen Workflow an, um ein neuronales Netzwerk auf einem Server zu trainieren und anschließend auf Plattformen wie dem Jetson TX1 oder dem Drive PX2 auszuführen.

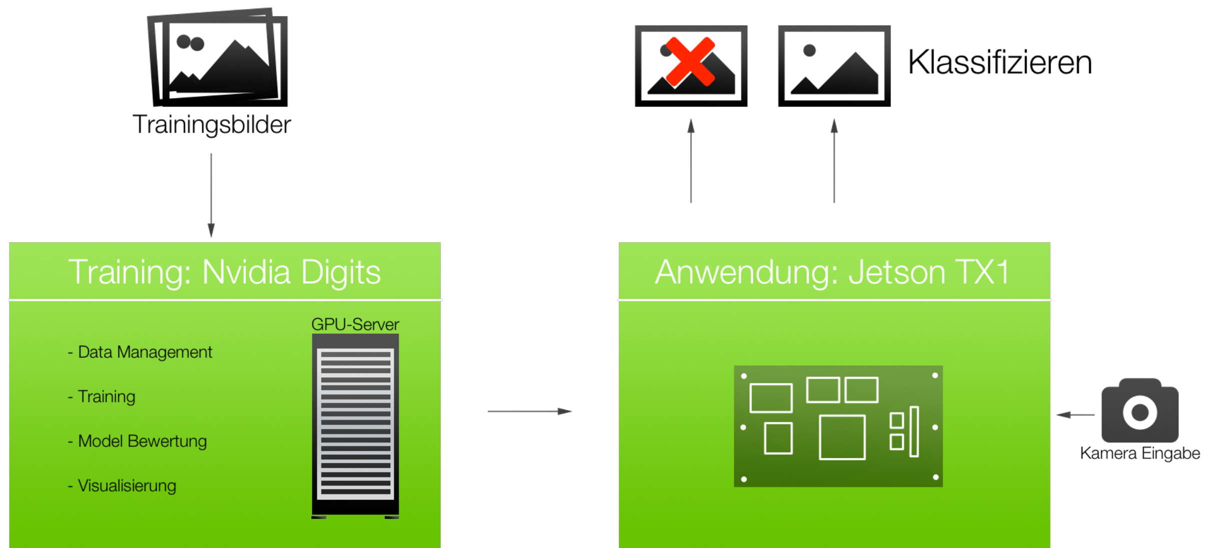

Abbildung 24 Prinzip des Trainierens und Ausführen eines Netzwerkes

DIGITS ermöglicht das Hochladen der gesammelten Trainingsbilder auf den Server und erstellt einer LMDB Datenbank (Lightning Memory-Mapped Database). Verschiedene Konfigurationen an den Bilddaten können gewählt werden, wie zum Beispiel die Auflösung oder die Art des quadratischen Zuschnittes. Zusätzlich lassen sich die Bilder in eine Validierungs- und Testdatenbank aufteilen.

Nach dem Erstellen einer Datenbank hat man die Möglichkeit, das Lernen eines neuen Netzwerkes anzulegen. Man kann zwischen Standardnetzwerken und vortrainierten Netzwerken wählen, oder eine eigene Netzwerkarchitektur erstellen.
Man kann die Anzahl an Trainings-Epochen, der *Solver* Typ, die *Batch* Größe, die Lern Rate, die Art der *Mean Subtraction* und die Anzahl der GPUs bestimmen.
Von den Trainingsdaten wird der Mittelwert der Pixel gebildet und dieser von dem zu klassifizierenden Bild abgezogen, um dieses zu normalisieren. Dies wird *Mean Subtraction* genannt.

Nach dem Starten des Trainingsvorganges in DIGITS werden auf der Weboberfläche Informationen zum Lernvorgang angezeigt. Die verbleibende Zeit zum Lernen, Hardwareinformationen und die aktuelle Genauigkeit des neuronalen Netzwerkes wird angegeben. Mit dem aktuellen Trainingsstand können noch vor Beendigung der Lernphase einzelne Bilder getestet werden und Statistiken beziehungsweise die Aktivierungen angezeigt werden.

Die Abbildung 25 visualisiert diese Aktivierung einer Netzwerkebene. Durch Anwenden verschiedener gelernter Filter auf ein Bild werden verschiedene Merkmale aktiviert. Exemplarisch ist die Aktivierung der erste Ebene dargestellt. Viele CNNs bilden in dieser Ebene Filter, welche den Garbor Filter ähneln. Es werden hierdurch Linien und Kanten aktiviert. Die *Activation Map* kann visualisiert werden, um zu überprüfen, welche Merkmale für das Netzwerk wichtig sind. Auch in der gezeigten Visualisierung kann man sehen, dass das Netzwerk auf Kanten und Flächen reagiert. In der Graphik bedeutet Rot eine starke Aktivierung und Blau eine schwächere bis keine Aktivierung.

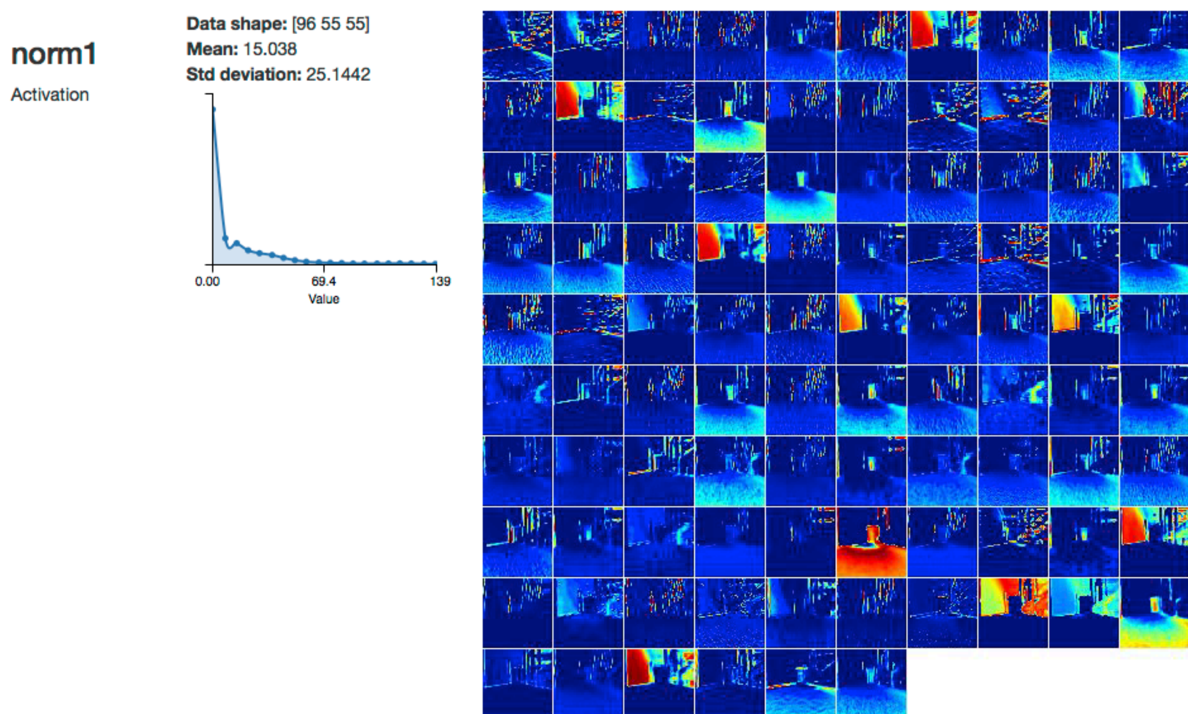

Abbildung 25 DIGITS – Visualisierung der Aktivierung eines Bildes in der ersten Ebene eines Netzes

Nach Beendigung des Lernvorganges erhält man in Form einer .caffemodel-Datei das fertige Netzwerk, welches auf der Fahrzeugplattform zum Klassifizieren verwendet wird. Diese Datei hat eine Größe von 300 MB und beinhaltet die Gewichtung der Neuronen.

3.3.4 Training

Als Architektur des neuronalen Netzwerkes wurde das AlexNet gewählt. Es ist ein viel verwendetes Netzwerk, welches seine Qualität im ILSVRC-2012 Wettbewerb als Gewinner bewiesen hat. Im Vergleich zu den folgenden Gewinnern, GoogLeNet, VGGNet oder ResNet, ist das AlexNet nicht so tief und deshalb einfacher zu verstehen. Dennoch erzielt es für die Anwendung dieser Arbeit eine gute Performanz.

Das AlexNet besteht aus acht Ebenen, fünf *convolutional* Ebenen und drei *fully-connected* Ebenen. Der Ausgang des Netzes ist eine *Softmax*. In Abbildung 26 ist die Architektur des AlexNet dargestellt. Als Eingang wird ein 224 *x* 224 *x* 3 Pixel großes Farbbild gegeben. Die Filter der ersten *convolutional* Ebene sind 11 *x* 11 Werte groß und tasten stückweise um vier Pixel verschoben das Eingangsbild ab (*Stride of 4*). Als Ausgang der ersten Ebene ergibt sich eine 55 x 55 x 96 große *Activation Map*. Diese ist 96 tief, da diese Ebene 96 Filter besitzt. Durch den *Stride* von vier verringert sich die Ausgangsmatrix auf 55 *x* 55 Werte. In der zweiten, dritten und sechsten Ebene des Netzes wir durch Max-pooling ebenfalls die jeweilige Ausgangsmatrix verkleinert. Abgeschlossen wird das Netzwerk durch drei *fully-connected* Ebenen. Die letzte Ebene bricht die Ergebnisse des Netzes auf die drei Klassen auf.

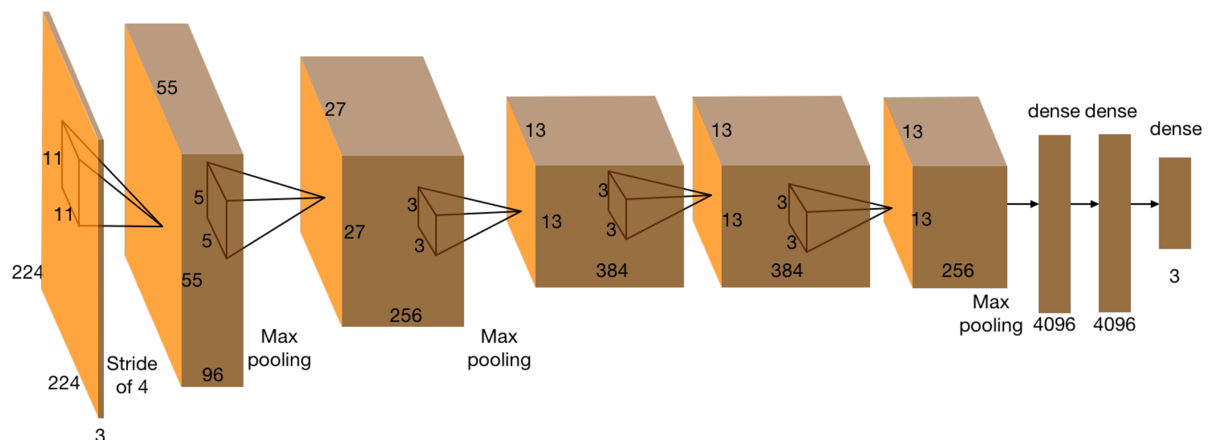

Abbildung 26 Architektur des AlexNet (Abbildung ähnelt einer in [3])

Trainiert wurde das Netzwerk 100 Epochen lang, dies bedeutet das anhand 100 Iterationen des gesamten Datensatzes die Neuronen angepasst werden. Insgesamt dauerte der Lernvorgang zwei Stunden.

Die Lernrate startet bei 0.01 und wurde im Zuge des Lernens auf 0.001 und 0.0001 verringert. Sie ist für das Lernen der Neuronen durch das Verfahren der Fehlerrückführung wichtig. Sie bestimmt die Größe der Schritte, in denen gelernt wird. Am Anfang des Lernens sollen die Neuronen schnell und grob angepasst werden, gegen Ende langsam, aber dafür fein.

3.4 Inference

Inference wird das Ausführen des Netzwerkes auf der Zielplattform genannt. In diesem Fall bedeutet dies, dass das neuronale Netz nach dem Trainieren auf einem GPU-Server zum Klassifizieren von Bildern auf der Fahrzeugplattform verwendet wird. Hierfür werden verschiedene Bibliotheken, Code, Anpassungen und Regelungen benötigt, die im Folgenden erläutert werden und zu einem autonomen Fahren führen.

3.4.1 Bibliotheken

Um ein unter Caffe trainiertes Netzwerk auszuführen, wird im Normalfall ebenfalls Caffe verwendet. Caffe wurde so entwickelt, dass es zum Trainieren auf GPUs, aber ebenso für die Ausführung auf Systemen mit CPUs geeignet ist. Mit Hilfe von cuDNN wird Caffe für Nvidia GPUs optimiert.

Jedoch hat Nvidia speziell nur für die Ausführung von neuronalen Netzwerken die Bibliothek TensorRT entwickelt. Sie bietet eine deutlich bessere Performanz gegenüber Caffe. Dies ist in Abbildung 27 und Abbildung 28 zu sehen. TensorRT hat mehr als die doppelte Performanz im Vergleich zu Caffe bei einem GoogLeNet mit FP16 Modus und einer Batch-Größe von zwei, ausgeführt auf dem Jetson TX1.

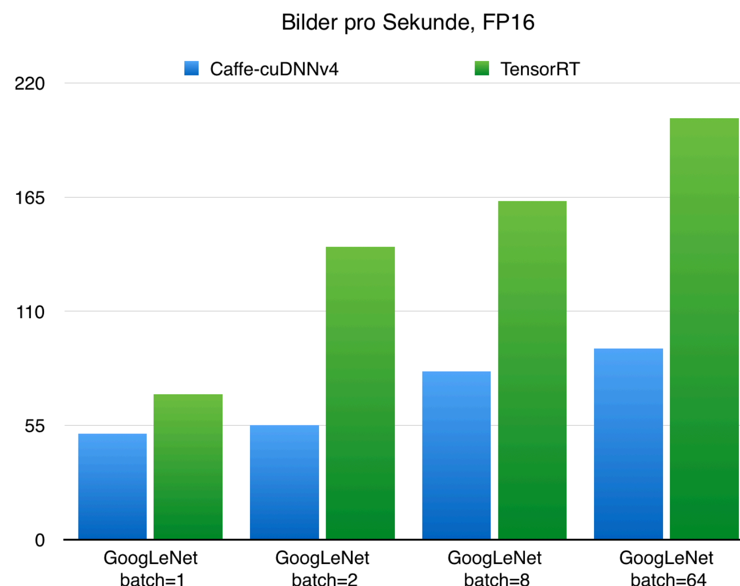

Abbildung 27 Geschwindigkeitsvergleich von Caffe und TensorRT (Informationen von [27])

Auch in Hinsicht des Energieverbrauches ist die Lösung von Nvidia der von Caffe überlegen. Der Unterschied zwischen einem CPU und einem GPU System ist deutlich in Abbildung 28 zu sehen.

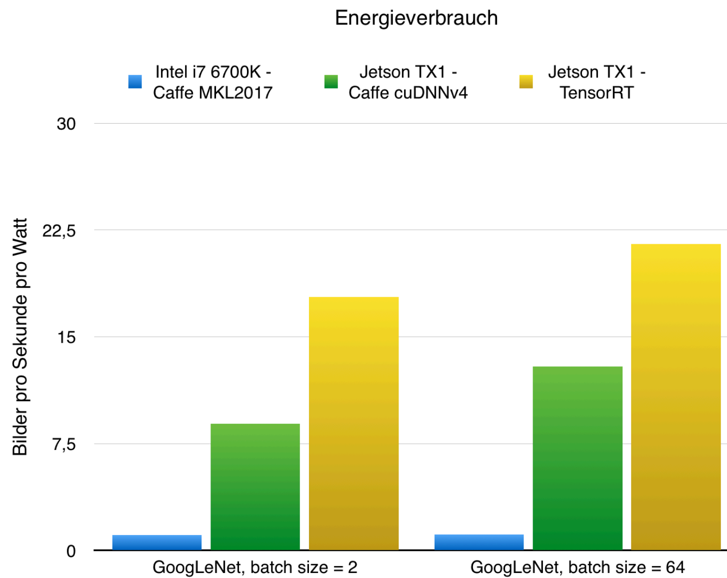

Abbildung 28 Energieverbrauch Caffe vs. TensorRT (Informationen von [27])

Auf Grund dieser Resultate wurde in dieser Arbeit TensorRT zum Ausführen des Netzes auf dem Jetson gewählt.

3.4.2 Code

Der Code zum Ausführen des neuronalen Netzwerkes auf dem Jetson TX1 besteht zum größten Teil aus einem Tutorial von Nvidia, zu finden in [28]. Der Code implementiert TensorRT zur effektiven und schnellen Klassifizierung von Bildern. Anpassungen wurden beim Aufrufen des Netzwerkes vorgenommen, so dass auch selbst trainierte Netze verwendet werden können. Dabei müssen zusätzlich die Mean-Werte aus dem Netzwerk in dem Code geändert werden, da es sonst zu einer falschen Klassifizierung kommt.

Ebenso wurde die UART-Verbindung zu dem Arduino in dem Code implementiert. Je nach Ergebnis der Klassifizierung werden über diese die Fahrsignale gesendet.

3.4.3 Regelung

Das Klassifizieren der Bilder funktioniert sehr genau, jedoch müssen aus den drei Klassen Lenksignale erzeugt werden.

Eine Möglichkeit wäre es, je nach Klassifizierung einen festen Lenkwinkel einzustellen. So könnte man bei der Klasse „g“ die Lenkung auf 90 Grad stellen, also gerade aus, bei der Klasse „l“ 45 Grad und bei „r“ 135 Grad, jeweils Vollausschläge der Lenkung. Es hat sich aber gezeigt, dass das Fahren sehr „digital“ wirkt und starke Schlangenlinien gefahren werden.

Ein Problem ist aufgetreten, wenn die Lenkung bei der Klasse „g“ auf 90 Grad gestellt wird. Das Auto steht nie genau parallel zum Gang und driftet folglich langsam zu einer Seite ab. Oft klassifiziert das Auto „g“, während es schon mit der Wand kollidiert, gezeigt in Abbildung 29.

Abbildung 29 Problem des Abdriftens

Um diese Problematik zu umgehen, muss vor einer Kollision mit der Wand diese erkannt und gegengelenkt werden. Deshalb wird bei einer geraden Klassifizierung die Lenkung auf 85 Grad gestellt, wenn der vorherige Lenkwinkel kleiner 90 Grad war und auf 95 Grad bei größer 90 Grad. Dies provoziert ein stärkeres Driften und ermöglicht dem Auto, die Wand schneller zu erkennen.

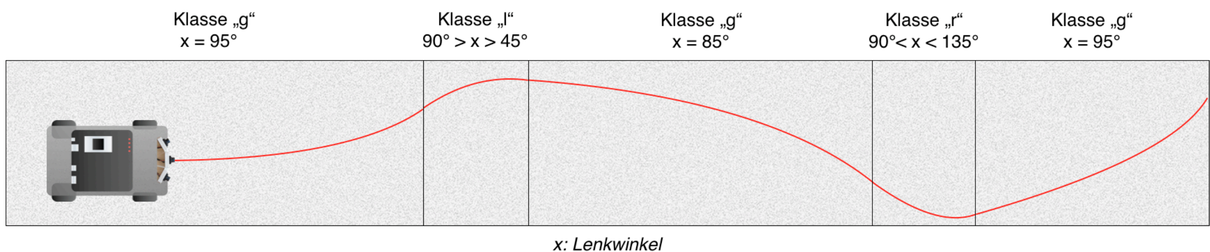

Abbildung 30 Provoziertes Abdriften um Wände zu erkennen

Um ein sanftes Lenken zu ermöglichen, wird der Winkel iterativ verändert. Es wird gegen 90 Grad mit ±5° pro Klassifizierung iteriert und langsamer von 90° weggeregelt mit ±2°. Veranschaulicht ist dies in Tabelle 4.

Tabelle 4 Regelung des Lenkwinkels

Klasse	angestrebter Winkel	aktueller Winkel	Aktion
l	45	Lenkwinkel > 90	-5
		Lenkwinkel < 90	-2
r	135	Lenkwinkel < 90	+5
		Lenkwinkel > 90	+2
g	90	Lenkwinkel > 90	Lenkwinkel = 95
		Lenkwinkel < 90	Lenkwinkel = 85

Ein Video des fahrenden Autos ist auf der Webseite http://bachelorarbeit.scheglmann.me/ bereitgestellt.

3.4.4 Geschwindigkeit des Klassifizierens

Ein schnelles einordnen des aktuellen Kamerabildes in eine der drei Klassen ist essenziell, um ein autonomes Fahren zu ermöglichen.
Durch das Verwenden von TensorRT und AlexNet dauert das Klassifizieren von Kamerabildern im Durchschnitt 0,018 Sekunden. Dies ergibt 55,6 klassifizierte Bilder pro Sekunde. Die Zeitmessungen wurden unter Fahrbedingung an 50 nacheinander folgenden Klassifizierungen der Kamera vorgenommen.
Die Kamera nimmt mit 30 fps auf und somit ist das Klassifizieren schneller, als dass die Kamera Bilder liefern kann. Dies zeigt, dass unter Echtzeit eine Klassifizierung möglich ist und schnell auf Änderungen des Kamerabildes reagiert werden kann.

4. Experimente und Ergebnisse

In diesem Abschnitt wird die Performanz und somit das Fahrverhalten des Autos untersucht. Es wird beschrieben, wie eine neue Umgebung erlernt werden kann. In drei verschiedenen Umgebungen wurden Testfahrten durchgeführt, welche wie folgt referenziert werden:

- „**Gang1**“: ursprüngliche Trainingsumgebung, in der gelernt worden ist.
- „**Gang2**“: Ein für das Auto unbekannter Gang. Der Boden hat eine andere Farbe und ein anderes Muster, als die gelernte Umgebung. Die Wände sind weiß, jedoch ist der Gang breiter.
- „**Weg draußen**“: Ein Weg außerhalb des Gebäudes. Das Testszenario besitzt auf einer Seite des Weges eine Wiese und auf der anderen Seite Bänke und eine Hecke.

Diese Szenarien sind in Abbildung 31 zu sehen. Der Gang1 und der Weg draußen sind im Detail im Kapitel 3.2.1 Datensatz Umgebungen beschrieben.

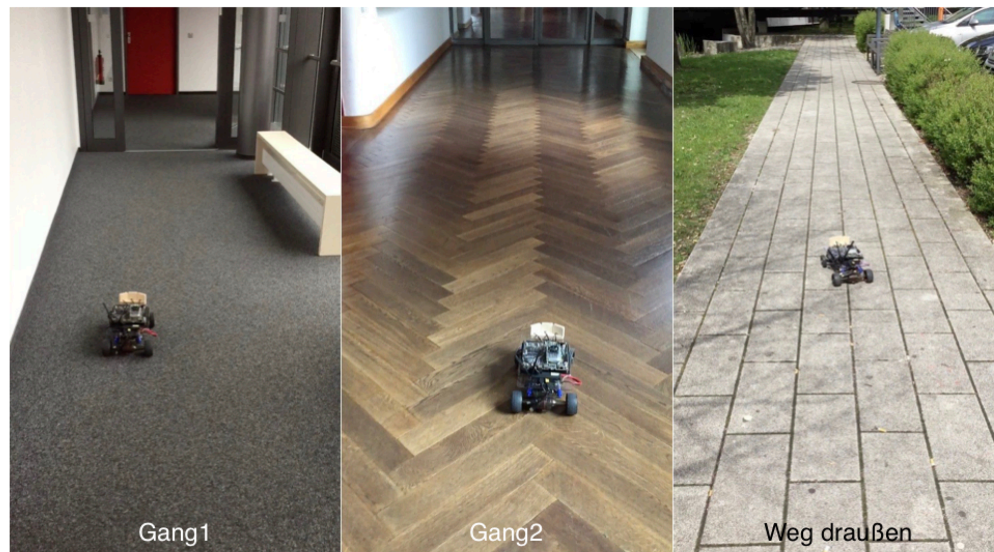

Abbildung 31 Gegenüberstellung der Testumgebungen

Zuerst wurde ein Netzwerk getestet, welches nur mit Trainingsdaten aus dem Gang1 trainierte.

4.1 Fahren in bekannter Umgebung

Es wurde untersucht, ob und wie das Auto in dem bekannten Gang1 autonom fährt. Wie erwartet, ist ein selbstständiges Fahren möglich. Es fährt leichte Schlangenlinien, bei Erkennen einer Wand wird entsprechend eingelenkt, bis das Auto wieder in Richtung des Ganges gerichtet ist. Schwierigere Situationen wie das Durchfahren einer Tür oder das Abbiegen in

dem Gang waren auch möglich. Letztendlich konnte in dem rechteckig aufgebauten Atrium (siehe Abbildung 15) mehrere Runden gefahren werden.
In den rechtwinkligen Kurven des Atriums fährt das Auto verstärkt Schlangenlinien, da die Klasse geradeaus nicht mehr bestimmt wird. Es entsteht der Eindruck, als würde das Fahrzeug die geradeaus Richtung des Ganges suchen. Sobald die Kurve durchfahren ist und die Klasse „g“ wieder bestimmt wird, fährt das Auto wie gewohnt weiter.

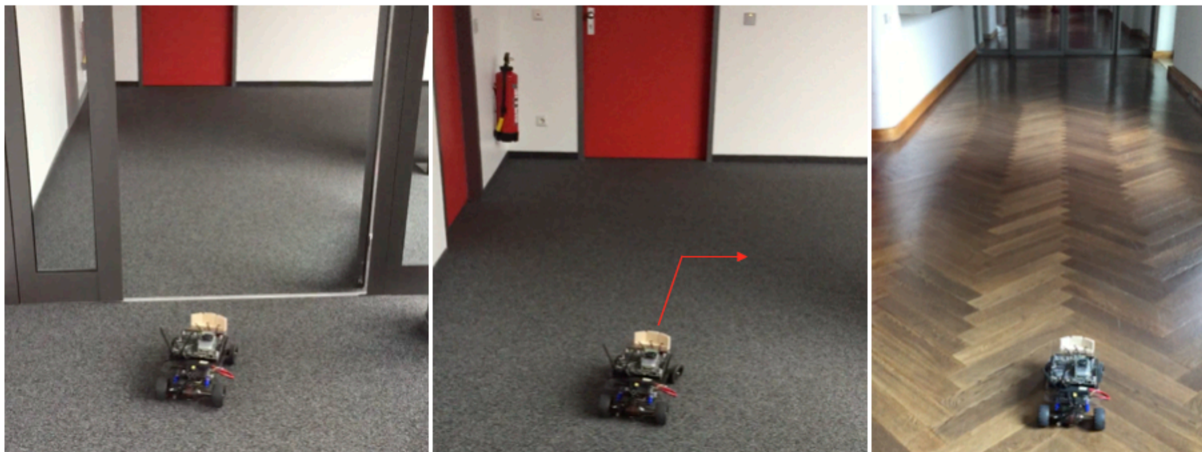

Abbildung 32 Fahren des Autos

4.2 Fahren in unbekannter Umgebung

Gerade für einen autonomen Roboter ist das Fahren in einer unbekannten Umgebung wünschenswert. Es sollen wichtige Merkmale eines Weges verallgemeinert werden, so dass diese in neuen Umgebungen angewandt werden können.
Der Datensatz, mit dem das Netzwerk trainiert hat, war nicht sehr variationsreich, da nur in derselben Umgebung, dem Gang1, Daten aufgenommen wurden. Deshalb hat das Netzwerk nicht nur Merkmale vom Weg gelernt, auch der Hintergrund erscheint für das Netz wichtig, da es durch weiße Wände, rote Türen, Heizkörper und Fenster Aktivierungen gab.
Es wurde das Verhalten des Netzwerkes in neuen Umgebungen untersucht.
Zuerst fuhr das Fahrzeug im Gang2. Das Auto zeigt ein ähnliches Verhalten, wie in der trainierten Umgebung, dem Gang1. Es fährt leichte Schlangenlinien und erkennt die Wände.
Der zweite Test erfolgte auf dem Weg draußen. Auf diesem konnte das Auto nicht selbstständig fahren. Es wurde deutlich, dass es verschiedene Merkmale erkennt und entsprechend lenkt, jedoch gab es viele falsche Klassifizierungen, weswegen das Fahrzeug von der Fahrbahn abwich. Die Umgebung besitzt zu viele Unterschiede zu den gelernten Merkmalen. Der Weg hat eine andere Farbe und Muster, es gibt keine Wände, sondern eine Wiese und eine Hecke und der Hintergrund besitzt auch keine Ähnlichkeit.

4.3 Lernen neuer Umgebungen

Aufgrund der vorherigen Ergebnisse, dass in einer neuen Umgebung, welcher der Trainingsumgebung nur wenig ähnelt, nicht zufriedenstellend autonom gefahren werden kann, stellt sich die Frage, wie eine neue Umgebung erlernt werden kann.
Konkret wurde auf dem Weg draußen Trainingsdaten gesammelt.
Es werden drei Möglichkeiten untersucht, ein Netzwerk zu trainieren, so dass das Auto in der neuen, aber auch in der bisherigen Umgebung fahren kann.
Der erste Versuch verwendet das schon bestehende Netzwerk, welches mit Daten aus dem Gang1 vortrainiert worden ist. Dieses wird nur mit dem neuen Datensatz trainiert.
In dem zweiten Versuch werden die Bilder der äquivalenten Klassen zusammengefasst. So erhält man einen neuen Datensatz, der ebenfalls drei Klassen besitzt. Jede Klasse enthält nun Bilder aus den zwei Aufnahmeumgebungen.
Im letzten Versuch werden aus den neuen Daten drei weitere Klassen erstellt, so dass man die sechs folgenden Klassen erhält:

- g_drinnen
- l_drinnen
- r_drinnen
- g_draußen
- l_draußen
- r_draußen

Jeder der Varianten wurde mit gleichen Parametern bei einer Dauer von 100 Epochen trainiert.

Im Folgenden werden die Varianten in den drei Testumgebungen Gang1, Gang2 und auf dem Weg draußen getestet. Das Fahrverhalten wird subjektiv bewertet.
Die erste Variante ermöglicht ein Fahren in der Umgebung draußen, jedoch ist das Fahren in der bisherigen Umgebung nicht mehr möglich. Die vorher gelernten Merkmale gehen verloren und werden durch die der neuen Umgebung ersetzt.
Bei der zweiten Variante werden drei Klassen aus allen gesammelten Daten gebildet. Es ist ein Fahren im Gang1, im Gang 2 und auf dem Weg draußen möglich.
Die dritte Variante enthält sechs Klassen und verhält sich wie der zweite Versuch, in dem ein Fahren in allen drei Umgebungen möglich ist.

Das Ergebnis des ersten Versuchs zeigt, dass ein einfaches Nachtrainieren mit Daten einer unbekannten Umgebung nicht möglich ist, sondern ein Trainieren mit dem kompletten Datensatz, also den neuen und den alten Daten nötig ist.
Die Versuche zwei und drei zeigen Möglichkeiten, neue Umgebungen zu lernen. In diesem Test sind keine merkbaren Unterschiede zwischen den Varianten aufgefallen. Die zweite Variante, alle vorhandenen Daten in drei Klassen zu teilen, ist in der Verwaltung des Datensatzes leichter.

Auf lange Sicht gesehen, verspricht diese Methode bessere Erfolge, die allgemeinen Merkmale eines Weges zu finden, da der Datensatz je Klasse eine größere Variation besitzt.

5. Zusammenfassung

Entwickelt wurde ein autonomes Modellauto, welches selbstständig auf Wegen und Gängen fahren kann. Der einzige verwendete Sensor ist eine nach vorne gerichtete Kamera. Das auf ein *Convolutional Neural Network* basierende System kann aus dem rohen Kamerabild die Richtung des Weges bestimmen. Diese wird in drei Zustände vereinfacht, ob der Weg „nach links gerichtet", „nach rechts gerichtet" oder „geradeaus gerichtet" ist.
Wichtige Merkmale zum Klassifizieren der Richtung des Ganges erlernte das neuronale Netzwerk anhand eines Datensatzes. Dieser wurde durch manuelles Fahren auf einem Weg und in einem Gang aufgenommen. Die Bilder sind den jeweiligen Labels zugeordnet, sodass ein überwachtes maschinelles Lernen möglich war.
Maßgeblich geprägt ist diese Arbeit von der Integration verschiedener Teilbereiche zu einem autonomen Fahrzeug. Dazu gehört nicht nur das Trainieren des neuronalen Netzwerkes und das Sammeln von Trainingsdaten, sondern auch das Erstellen einer geeigneten Fahrzeugplattform, welche durch das Netzwerk gesteuert werden kann.
Das Resultat ist ein Fahrzeug, welches in einer gelernten Umgebung problemlos fahren kann. Verschiedene Tests ergaben, dass in nicht trainierten Umgebungen ein autonomes Fahren nicht immer möglich war. Daraus ist zu schließen, dass der erstellte Datensatz zu wenig unterschiedliche Umgebungen enthält und das Netzwerk nur spezifische Merkmale einer Umgebung gelernt hat, anstatt allgemeine Merkmale eines Weges zu lernen.

Da das Netzwerk nur drei Zustände unterscheiden kann, ist die Steuerung des Autos entsprechend ungenau. Im Rahmen weiterer Untersuchungen kann in Zukunft das Fahrverhalten noch verbessert werden. Ein vielversprechender Ansatz ist, anstatt drei Klassen, einen Datensatz zu erstellen, welcher den Lenkwinkel als Label den entsprechenden Bildern zuordnet. Das anhand dessen trainierte Netzwerk soll den Lenkwinkel bestimmen können und folglich ein Fahren ohne große Schlangenlinien ermöglichen.
Die Literatur [29] und die Firma comma.ai [30] nutzen solch eine Methode, um einen Spurhalteassistenten für Autos zu realisieren. Es wurde eine große Menge an Videomaterial beim menschlichen Fahren im Straßenverkehr gesammelt und synchron der Lenkwinkel aufgezeichnet. Das Resultat ergibt ein CNN, welches das Fahrverhalten analog dem des Menschen gelernt hat.

Literaturverzeichnis

[1] J. Schmidhuber, Deep learning in neural networks: An overview, Elsevier, 2015.

[2] M. Copeland, „What's the Difference Between Artificial Intelligence, Machine Learning, and Deep Learning?," Juli 2016. [Online]. Available: https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/. [Zugriff am April 2017].

[3] A. Krizhevsky, I. Sutskever und G. E. Hinton, „ImageNet Classification with Deep Convolutional Neural Networks.," *Advances in Neural Information Processing Systems 25 (NIPS2012),* 2012.

[4] J. Beckett, „GPU-Powered Deep Learning Being Used to Speed Colon Cancer Diagnosis," Nvidia, Dezember 2015. [Online]. Available: https://blogs.nvidia.com/blog/2015/12/23/deep-learning-cancer/. [Zugriff am April 2017].

[5] A. Giusti, J. Guzzi, D. Ciresan, F. L. He, J. P. Rodriguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. A. D. Caro, D. Scaramuzza und L. Gambardella, „A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots," *IEEE ROBOTICS AND AUTOMATION LETTERS,* November 2015.

[6] D. H. Hubel und T. N. Wiesel, RECEPTIVE FIELDS, BINOCULAR INTERACTION AND FUNCTIONAL ARCHITECTURE IN THE CAT'S VISUAL CORTEX, D. o. P. H. M. S. Neurophysiology Laboratory, Hrsg., Boston, Massachusetts, 1961.

[7] K. Fukushima, Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position, Springer-Verlag, 1980.

[8] Y. LeCun, L. Bottou, Y. Bengio und P. Haffner, „Gradient-Based Learning Applied to Document Recognition.," 1998.

[9] I. Goodfellow, Y. Bengio und A. Courville, Deep Learning, MIT Press, 2016.

[10] F.-F. Li, J. Johnson und S. Yeung, „CS231n Convolutional Neural Networks for Visual Recognition," Stanford University, [Online]. Available: http://cs231n.github.io/convolutional-networks/#conv. [Zugriff am April 2017].

[11] D. Scherer, A. Müller und S. Behnke, „Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition,“ *International Conference on Artificial Neural Networks,* September 2010.

[12] Y. LeCun, K. Kavukcuoglu und C. Farabet, „Convolutional networks and applications in vision,“ *Proceedings of 2010 IEEE International Symposium on Circuits and Systems,* August 2010.

[13] K. Jarrett, K. Kavukcuoglu, M. Ranzato und Y. LeCun, „What is the Best Multi-Stage Architecture for Object Recognition?,“ *2009 IEEE 12th International Conference on Computer Vision,* Mai 2010.

[14] D. Ciresan, U. Meier und J. Schmidhuber, „Multi-column Deep Neural Networks for Image Classification,“ arXiv, 2012.

[15] V. Nair und G. E. Hinton, „Rectified Linear Units Improve Restricted Boltzmann Machines,“ in *Proc. 27th International Conference on Machine Learning*, 2010.

[16] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever und R. Salakhutdinov, „Dropout: A Simple Way to Prevent Neural Networks from Overfitting,“ in *Journal of Machine Learning Research 15*, 2014, pp. 1929-1958.

[17] C. Szegedy, P. Sermanet, W. Liu, Y. Jia, D. Erhan, S. Reed, D. Anguelov, V. Vanhoucke und A. Rabinovich, „Going deeper with convolutions,“ arXiv, 2014.

[18] D. Ciresan, U. Meier, J. Masci und J. Schmidhuber, „A Committee of Neural Networks for Traffic Sign Classification,“ researchgate.net, 2011.

[19] R. Hadsell, P. Sermanet, J. Ben und A. Erkan, „Learning Long-Range Vision for Autonomous Off-Road Driving,“ Wiley Periodicals, Inc., 2008.

[20] Arduino Team, „Arduino offical website,“ [Online]. Available: www.arduino.cc. [Zugriff am März 2017].

[21] Arduino Team, „Arduino offical website,“ [Online]. Available: www.arduino.cc/en/Main/ArduinoBoardNano. [Zugriff am März 2017].

[22] A. v. d. Lugt, „An iOS 9 app to communicate with a HM10 Bluetooth module,“ [Online]. Available: https://github.com/hoiberg/HM10-BluetoothSerial-iOS. [Zugriff am April 2017].

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama und T. Darrell, „Caffe: Convolutional Architecture for Fast Feature Embedding,“ arxiv, 2014.

[24] NVIDIA, „CUDA Zone," [Online]. Available: https://developer.nvidia.com/cuda-zone. [Zugriff am April 2017].

[25] NVIDIA, „GPU Accelerated Deep Learning," [Online]. Available: https://developer.nvidia.com/cudnn. [Zugriff am April 2017].

[26] NVIDIA, „NVIDIA DIGITS," [Online]. Available: https://developer.nvidia.com/digits. [Zugriff am April 2017].

[27] D. Franklin, „JetPack 2.3 with TensorRT Doubles Jetson TX1 Deep Learning Inference," Nvidia, 12 September 2016. [Online]. Available: https://devblogs.nvidia.com/parallelforall/jetpack-doubles-jetson-tx1-deep-learning-inference/. [Zugriff am April 2017].

[28] D. Franklin, „Deploying Deep Learning," Nvidia, [Online]. Available: https://github.com/dusty-nv/jetson-inference. [Zugriff am Februar 2017].

[29] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao und K. Zieba, „End to End Learning for Self-Driving Cars," arxiv, 2016.

[30] comma.ai, „Steering Angle model," [Online]. Available: https://github.com/commaai/research. [Zugriff am April 2017].

[31] NVIDIA, „NVIDIA® JETSON™," [Online]. Available: http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html. [Zugriff am April 2017].