Fakultät für Elektrotechnik
und Informationstechnik

Hochschule
München
University of
Applied Sciences

HM

Machine Learning Showcase für TQ-Modul

Autonomes Fahren

Machine Learning Showcase for TQ-Module

Autonomous Driving

Bachelorarbeit

Studiengang Elektrotechnik und Informationstechnik

Eingereicht von: Vincent Bamberger
lfd. Nr.: 2157

Bearbeitungsbeginn: 13.10.2021
Abgabetermin: 13.04.2022

Betreuer: Prof. Dr. Manfred Gerstner

Firma: TQ-Systems GmbH
Vertreter: Thomas Waldecker

Hochschule München
Fakultät für Elektrotechnik und Informationstechnik

Erklärungen der/des Bearbeiterin/s:

Bamberger, Vincent

1) Ich erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe.

   Sämtliche benutzte Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate sind als solche gekennzeichnet.

Ort, Datum Unterschrift

2) Ich erkläre mein Einverständnis, dass die von mir erstellte Bachelorarbeit in die Bibliothek der Hochschule München eingestellt wird. Ich wurde darauf hingewiesen, dass die Hochschule in keiner Weise für die missbräuchliche Verwendung von Inhalten durch Dritte infolge der Lektüre der Arbeit haftet. Insbesondere ist mir bewusst, dass ich für die Anmeldung von Patenten, Warenzeichen oder Geschmacksmuster selbst verantwortlich bin und daraus resultierende Ansprüche selbst verfolgen muss.

Ort, Datum Unterschrift

HM

Hochschule München
Fakultät für Elektrotechnik und Informationstechnik

Lfd. Nr. 2157

Bachelorarbeit

Abgabedatum: 13.04.2022

Betreuer/in:	**Studierende/r:**	**Studiengruppe:**
Prof. Dr. Manfred Gerstner	Vincent Bamberger	EI7

Thema:

Machine Learning Showcase für TQ-Modul - Autonomes Fahren

Machine Learning Showcase for TQ-Module - Autonomous Driving

Kurzfassung:

Diese Arbeit wurde extern bei der Firma TQ-System durchgeführt. Die Firma bietet seit kurzem ein Modul an, das über Hardwarebeschleunigung für neuronale Netze verfügt. Das Ziel der Arbeit ist es, für dieses TQ-Modul eine Demoapplikation zu erstellen, die dessen Fähigkeiten mit Machine Learning präsentiert.
Im Zuge dieser Demo ist ein motorisiertes Modellauto in der Lage, selbstständig und nur mithilfe einer Weitwinkelkamera entlang einer abgesteckten Strecke zu steuern. Das Vorbild, dessen Ansatz bei der Entwicklung verfolgt wurde, ist ein 2016 demonstriertes selbstfahrendes Auto von Nvidia.
Das Programm ist vollständig in *Python* geschrieben. Das verwendete Framework, mithilfe dessen das neuronale Netz erstellt und trainiert wurde, ist *TensorFlow*.
Die Steuerung des Fahrzeugs funktioniert trotz begrenztem Entwicklungsaufwand und wenig Training sehr gut. Das Modellauto bleibt zuverlässig auf der Strecke und dank der Hardwarebeschleunigung bleibt die Berechnungszeit des trainierten neuronalen Netzes unter einer Millisekunde.

Abstract:

The work for this thesis was done externally at the company TQ-Systems. Since recently they offer a module that has support for hardware acceleration on neural networks. The goal of this work is to develop a demo application that demonstrates the machine learning capabilities of said module.
The application makes a motorized model car drive on its own along a marked track, only using a single wide angle camera. The example, this application was developed after, is a self-driving car that was demonstrated by Nvidia in 2016. All programming was done in *Python*. The machine learning library *TensorFlow* was used to create and train the neural Network.
Even though the development effort and the amount of training required was limited, the vehicle steering turned out to work very well. The vehicle is able to stay on the track rather reliably and thanks to the hardware acceleration, the processing time of the neural Network is below one millisecond.

Inhaltsverzeichnis

1 Einleitung 6
1.1 Zur TQ-Systems GmbH / Geschäftsbereich TQ Embedded 6
1.2 Bedeutung von maschinellem Lernen in Embedded 7
2 Ziel der Arbeit 9
3 Aufbau der Arbeit 10
4 Technische Grundlagen 11
4.1 Machine Learning und neuronale Netzwerke 12
4.2 Convolutional Neural Networks (CNN) 13
4.3 Software der TQ-Module 14
5 Related Work 15
5.1 Der „End-to-End Deep Learning“-Ansatz von Nvidia 15
5.1.1 Das Sammeln der Trainingsdaten 15
5.1.2 Die Architektur des neuronalen Netzwerks von Nvidia 16
5.1.3 Datenvorbereitung, Trainingsvorgang und Evaluierung des Modells 17
5.2 Der Ansatz von Nvidia im Kleinformat – „DeepPiCar“ 17
5.2.1 Vorgangsweise im Vergleich zu Nvidia 17
5.3 Diese Bachelorarbeit im Vergleich 18
6 Durchführung 20
6.1 Vorbereitung 20
6.2 Sammeln der Trainingsdaten 21
6.2.1 Hardware 21
6.2.2 Software und Steuerung des Fahrzeugs 22
6.3 Training 23
6.4 Fahren mit Hilfe des trainierten Netzwerks 26
6.4.1 Hardware 26
6.4.2 Software 28
6.5 Entwicklungen über die Projektziele hinaus 28
6.5.1 Der „TensorFlow Object Detection Modelmaker” 28
6.5.2 Testen des Programms „eIQ Portal“ von NXP 31
6.5.3 Implementieren von Objekterkennung in das PiCar 31
6.5.4 Video für Anhang an strategische Präsentation 33
7 Ergebnisse 34
7.1 Performance des Fahrzeugs auf der Strecke 34
7.1.1 Genauigkeit des trainierten Netzwerks 34

7.1.2 Leistung des Netzwerks auf unterschiedlicher Hardware ____ 35
7.2 Erkenntnisse durch das Training des Netzwerkmodells ____ 36
7.3 TFOD Modelmaker und Objekterkennung ____ 37
7.4 Ergebnisse der Evaluierung von *eIQ Portal* ____ 38
7.5 Einsatz des selbst erstellten Videos über das Fahrzeug ____ 40
8 Fazit und Ausblick ____ 42
Glossar ____ 43
Abbildungsverzeichnis ____ 47
Literaturverzeichnis ____ 48

1 Einleitung

Das Thema künstliche Intelligenz ist in den vergangenen Jahren immer präsenter geworden. Anfangs kam sie überwiegend in über Server bereitgestellten Diensten zum Einsatz, beispielsweise in Form von immer besser werdenden sprachgesteuerten Assistenten. Auch fanden speziell für das Rechnen mit neuronalen Netzwerken optimierte Hardwarebeschleuniger immer mehr Verbreitung. Besonders in Prozessoren mit niedrigem Stromverbrauch, wie sie vor allem in mobilen Endgeräten wie Smartphones eingesetzt werden, sind sie nach wie vor integriert. Aufgrund von Fortschritten bei den künstlichen neuronalen Netzwerken sowie der steigenden Leistungsfähigkeit der Hardware, wird maschinelles Lernen immer mehr auch für Anwendungen in der Industrie nachgefragt. Hierfür bietet seit kurzer Zeit auch die Firma *TQ-Systems*, bei der diese Arbeit extern durchgeführt wurde, Modullösungen mit Hardwarebeschleunigung für neuronale Netze. Eines dieser Module und dessen Fähigkeiten in Bezug auf maschinelles Lernen soll im Rahmen dieser Bachelorarbeit behandelt werden.

1.1 Zur TQ-Systems GmbH / Geschäftsbereich TQ Embedded

Die *TQ-Systems GmbH* ist ein Technologiekonzern, welcher Elektronik sowohl entwickelt als auch fertigt. *TQ* ist eine Abkürzung und steht für: Technologie in Qualität. Angeboten werden, neben Fertigung und Entwicklung, außerdem das gesamte Paket aus Planung, Entwicklung, Test, Qualifizierung und Fertigung, was bezeichnet wird als *Electronic Engineering Manufacturing Services*, kurz *E²MS* [1].

Der Geschäftsbereich *TQ Embedded* bietet Modullösungen an, mit Prozessoren unterschiedlicher Architekturen wie ARM® und x86, aber in Einzelfällen auch mit Mikrocontrollern. Solche Module sind allgemein als *Embedded-Module* bekannt. Die Firma TQ-Systems nennt ihre eigenen Module *TQ-Module* [2]. Eingebettet in einer größeren Anwendung steuern sie die dort nötigen Abläufe. Embedded-Module können, sobald sie einmal entwickelt wurden, in verschiedenen Baugruppen und Anwendungen eingesetzt werden. So kann einiges an Entwicklungsaufwand gespart werden, besonders da die eingesetzten Prozessoren und deren Peripherie immer schneller und die Entwicklung und das Layout von geeigneten Trägerplatinen dadurch immer komplizierter werden [3]. Damit die Kunden TQ-Module leichter in ihre Anwendung einbauen, beziehungsweise leichter eine Anwendung um das Modul herum entwickeln können, werden sogenannte *Starterkits* angeboten, welche die entsprechenden Module bereits implementieren. In Ausnahmefällen sind für ein Modul sogar zwei oder mehr verschiedene Starterkits verfügbar. Sie legen auch wichtige Schnittstellen der Module, wie Videoausgänge oder ähnliches, bereits auf Anschlüsse, wodurch sie für Evaluierungszwecke in der Softwareentwicklung und als Referenz für die Hardwareentwicklung geeignet sind.

Abbildung 1: TQMa8MPxL - Embedded Modul von TQ-Systems

Bei dieser Arbeit wird das Modul *TQMa8MPxL*, siehe auch Abbildung 1, verwendet. Es implementiert einen *i.MX 8M Plus* Prozessor der Firma *NXP Semiconductors*. Das Besondere an diesem Prozessor ist seine *NPU*, kurz für *Neural Processing Unit*. Dabei handelt es sich um spezialisierte Hardware, die das Rechnen mit neuronalen Netzen beschleunigen soll. Das genannte TQ-Modul wird auf dem Starterkit *STKa8MPxL*, zu sehen in Abbildung 2, eingesetzt [4].

Abbildung 2: STKa8MPxL - Starterkit von TQ-Systems

1.2 Bedeutung von maschinellem Lernen in Embedded

Maschinelles Lernen, in der Fachsprache bezeichnet als *Machine Learning*, kurz *ML*, gewinnt in vielen Bereichen, wie auch in der Industrie und in Embedded-Systemen, zunehmend an Bedeutung. Auch für TQ wird es immer wichtiger sich damit auseinander zu setzen. Die Gründe hierfür werden im Folgenden erläutert, die Informationen stammen dabei aus dem Beitrag *Embedded Machine Learning* des *Fraunhofer IIS* [5].

Prinzipiell gibt es zwei Möglichkeiten maschinelles Lernen in Embedded-Systemen einzusetzen. Entweder kommuniziert das System mit Servern oder einer Cloud, die ihm die Berechnung des neuronalen Netzes abnimmt, oder das System führt die Berechnung mit eigenen Ressourcen selbst durch.

Der Vorteil davon, die Berechnung in die Cloud auszulagern, ist die Möglichkeit, bei Bedarf sehr viel Rechenleistung bereitstellen zu können. In Embedded-Systemen ist die lokal verfügbare Leistung durch die verwendete Hardware, den Stromverbrauch und die Kapazität der Kühlung oft eingeschränkt. Der Nachteil dieser Auslagerung ist die dafür notwendige Kommunikation, da sie für zusätzliche Verzögerung sorgt. Falls über Funk kommuniziert wird, ist sie bei Problemen mit der Netzabdeckung oft nicht zuverlässig.

Dadurch, dass Daten an eine Cloud übertragen werden müssen, entstehen außerdem zusätzliche Aufwände, die Kosten verursachen. Daten, die beim Einsatz von ML an neuronale Netzwerke gegeben werden, können in ihrem Umfang sehr groß werden. Besonders wenn es sich um eine Anwendung mit Kamerainformation handelt, in der Regel als *Vision-Anwendung* bezeichnet. Die Übertragung von großen Datenmengen nimmt viel Bandbreite in Anspruch. Im Falle von Funkübertragung wird noch zusätzliche Leistung für den Sender und Empfänger benötigt und es muss eine geeignete Antenne vorhanden sein.

Letztendlich muss abgewogen werden, welche Einschränkungen in Kauf genommen werden können. Hat die Anwendung also hohe Echtzeitanforderungen, kann die Verzögerung, die durch die Datenübertragung an die Cloud entsteht, zu groß sein. Wird die Berechnung lokal durchgeführt, dauert sie möglicherweise zu lange, wenn das System nicht ausreichend Rechenleistung zur Verfügung stellen kann.

Durch die technologischen Fortschritte in den letzten Jahren, sind die Architekturen der neuronalen Netze, sowie die Prozessoren und Hardwarebeschleuniger auf denen sie laufen immer effizienter, schneller und besser geworden. ML-Anwendungen, die zuvor die Rechenleistung eines Servers benötigten oder die wegen zu hoher Echtzeitanforderungen überhaupt nicht umsetzbar waren, sind heute in vielen Fällen direkt auf Embedded-Systemen, wie dem TQMa8MPxL, lauffähig.

2 Ziel der Arbeit

Das Ziel dieser Bachelorarbeit ist das Entwickeln einer Demoapplikation für das genannte TQ-Modul. Sie soll dessen Fähigkeiten bei tiefem maschinellem Lernen, allgemein Bezeichnet als *Deep Learning*, präsentieren. Dadurch sollen auch Erfahrung mit der Entwicklung von ML-Applikationen für das genannte Modul gesammelt werden. Ebenso ist damit zu testen, ob die von NXP bereitgestellten Softwarekomponenten für das Beschleunigen neuronaler Netze auf ihrem Prozessor wie beworben funktionieren. Es wird erwartet, dass Kunden, die solche Module kaufen, in ihrer Anwendung ein Netz einsetzen und von den vorhandenen Hardwarebeschleunigern wie der NPU Gebrauch machen wollen. Es liegt im Interesse der Firma, Kunden bei der Entwicklung ihrer Anwendung unterstützen zu können. Daher ist es wichtig zu wissen, was auf der angebotenen Hardware möglich ist, welche Leistung zu erwarten ist und wie man bei der Entwicklung am besten vorgeht.

Es wurde auch angestrebt etwas zu entwickeln, das eventuell auf der nächsten Messe *Embedded World* vorgestellt werden kann. Das Resultat der Arbeit soll also idealerweise etwas sein, dass in der Lage ist, bei Betrachtern Eindruck zu hinterlassen.

Dementsprechend wurde eine Demoapplikation festgelegt. Berücksichtigt wurden dabei besonders die Kriterien, die eine typische Anwendung von Embedded Machine Learning ausmachen, wie bereits erläutert in Kapitel 1.2.

Zu entwickeln war ein selbstfahrendes Modellauto, das mit Hilfe einer Weitwinkelkamera in der Lage sein soll über eine markierte Strecke zu steuern. Das Fahrzeug muss jede neue Information von der Kamera nahezu in Echtzeit verarbeiten. Die Anwendung entspricht daher einer mobilen, batteriebetriebenen und sehr zeitkritischen Vision-Applikation, für die Cloud gestützte Berechnung nicht praktikabel und so Embedded Machine Learning umso geeigneter ist.

Abbildung 3: Vorschau auf das selbstfahrende Fahrzeug mit STKa8MPxL

Der Ansatz hierfür stammt aus der Demonstration eines selbstfahrenden Autos von Nvidia, näher beschrieben in 5.1. Die Architektur des neuronalen Netzwerks ist aus dem Artikel von Nvidia übernommen worden. Auch das Sammeln und Aufbereiten der Daten geschieht grundsätzlich nach der dort beschriebenen Vorgehensweise.

Den Anforderungen nach sind am Ende der Arbeit folgende Ergebnisse vorzulegen.

Zum Ersten hat die entwickelte Software für das Sammeln von Trainingsdaten, das Trainieren des neuronalen Netzwerks und die Anwendung des trainierten Modells auf dem Starterkit im firmeninternen *GitLab* verfügbar zu sein. Zum Zweiten wird die Hardware, dass bedeutet ein mit Kameras ausgestattetes Fahrzeug für das Sammeln der Trainingsdaten und für das eigenständige Fahren, abgenommen. Darüber hinaus ist das Sammeln der Trainingsbilder, das Trainieren des Modells und das Nutzen des trainierten Modells zu dokumentieren.

3 Aufbau der Arbeit

Diese Arbeit nimmt Bezug zu maschinellem Lernen, was nicht Teil des Standardwissens des Studiengangs Elektrotechnik ist. Im nächsten Kapitel werden daher kurz die wichtigsten Grundlagen zu neuronalen Netzen und maschinellem Lernen erklärt. Dort wird auch die Software der TQ-Module knapp erklärt.

Das Kapitel 5 behandelt die zwei Artikel, zu denen hier Bezug genommen wird. Zuerst wird der Beitrag zu dem selbstfahrenden Auto von Nvidia, auf dem diese Arbeit aufbaut, strukturiert erklärt. Der zweite Artikel dokumentiert einen dieser Arbeit ähnlichen Versuch, den Ansatz von Nvidia nachzuahmen. Er soll hier unteranderem als Vergleich dienen.

Das Kapitel 6 beinhaltet die Beschreibung aller während der Arbeit durchgeführten Tätigkeiten. Das schließt die Bearbeitung der festgelegten Projektziele in Kapitel 6.1 bis 6.4 ein, sowie auch alle Entwicklungen und Nebenaufgaben die über die Ziele hinausgehen in Kapitel 6.5.

In Kapitel 7 werden die Resultate der durchgeführten Arbeiten präsentiert. Hier wird auch bewertet, wie gut die jeweiligen Vorhaben umgesetzt werden konnten und inwiefern entstandene Produkte bereits eingesetzt wurden. Sofern nötig, werden die Ergebnisse jeweils auch näher erklärt.

Das letzte Kapitel 8 behandelt abschließend den Ausblick auf potenziell nachfolgende Arbeiten sowie die Zukunft der bei dieser Arbeit entstandenen Produkte.

4 Technische Grundlagen

Für den Autor ist diese Bachelorarbeit, trotz fortgeschrittener Programmiererfahrung insbesondere mit der Programmiersprache Python, der erste Kontakt mit der Entwicklung von ML-Applikationen und Machine Learning überhaupt. Das hierfür benötigte Wissen musste zu Beginn der Arbeit durch selbstständige Recherche angeeignet werden.

Durch die Vorgabe des zu verwendenden i.MX 8M Plus Prozessors von NXP, ist das Durchlesen der Dokumentation *i.MX Machine Learning User's Guid*e [6] der logische erste Schritt. Um den Aufwand zu begrenzen, ist es notwendig gewesen, sich möglichst zu Beginn für eines der verfügbaren Frameworks für das Programmieren von ML-Anwendungen zu entscheiden. Nach ersten Recherchen zählen davon *TensorFlow* und *PyTorch* zu den am weitest verbreiteten, die auch auf der vorgegebenen Hardware zur Verfügung stehen [7]. Nach aktuellem Stand der oben referenzierten Dokumentation von NXP, unterstützten die i.MX Prozessoren mit PyTorch noch keine Hardwarebeschleunigung. Da TensorFlow auch seitens NXP allgemein besser dokumentiert ist, war das die endgültige Wahl.

TensorFlow wurde von dem *Google Brain Team* entwickelt und findet seither in vielen kommerziellen Produkten von Google seine Anwendung [8]. Die Veröffentlichung des Frameworks als Open-Source-Projekt geschah im Jahr 2015, die zweite Version *TensorFlow 2*, die auch in dieser Arbeit verwendet wird, wurde 2019 veröffentlicht. Der Begriff *Tensor* beschreibt allgemein eine Datenstruktur beliebiger Dimension, und umfasst einfache Skalare, Vektoren und Matrizen beziehungsweise Arrays. Der Tensor ist die Datenstruktur, die in TensorFlow hauptsächlich verwendet wird [9]. Sowohl mit der vollen Installation von TensorFlow als auch mit der abgespeckten und daher üblicherweise in Embedded-Systemen verwendeten Version *TensorFlow Lite*, kann in Python programmiert werden. Die Verwendung in C++, Java und anderen Sprachen ist ebenfalls möglich, jedoch weniger üblich. Bei dieser Arbeit wird ausschließlich in Python programmiert.

Der nächste Schritt war das grundsätzliche Kennenlernen von TensorFlow in Python, erst einmal ohne den Bezug zu dem TQ-Modul. Die offizielle Internetseite stellt bereits für viele Anwendungsbereiche wie Bild, Text oder Audioerkennung einfache Tutorials, kategorisiert in Anfänger und Fortgeschrittene, zur Verfügung [10]. Es hat sich anfangs allerdings als hilfreich erwiesen, die Grundlagen mit der Hilfe von Videokursen auf YouTube zu lernen. Nennenswert ist hierbei ein knapp sieben stündiges Video auf dem Kanal *freeCodeCamp.org* [11]. Das genannte Video baut auf den Tutorials der offiziellen Internetseite von TensorFlow auf.

Auf die Recherchen und den Videokurs folgten eigene Experimente in temporären Python-Skriptdateien wie zum Beispiel mehrere Versuche eine einfache Gesichtserkennung zu bauen. Die ersteren dieser Versuche basieren auf den offiziellen Beispielen zu einfacher Bildklassifizierung, die letzteren auf denen zur Klassifizierung mittels sogenannter *Convolutional Neural Networks*.

Weitere Experimente wurden mit der ebenfalls von Google entwickelten, auf dem normalen TensorFlow-Framework aufbauenden, *TensorFlow Object Detection API* durchgeführt und ergaben ein zusätzliches Produkt neben den eigentlichen Projektzielen. Darauf wird in Kapitel 6.5.1 näher eingegangen.

Die folgenden zwei Unterkapitel sollen die grundlegenden, für die Arbeit relevanten Erkenntnisse der Recherchen zum Thema Machine Learning erklären. Das dritte Unterkapitel erklärt knapp die Art und Funktionsweise der Software, mit der TQ-Module betrieben werden.

4.1 Machine Learning und neuronale Netzwerke

Was man sich darunter vorstellen kann, und wie sie funktionieren

Die Informationen auf denen die Erklärungen in diesem Kapitel basieren, stammen unteranderem aus dem Beitrag *Neuronale Netze: Ein Blick in die Black Box* auf *informatik-aktuell.de* [12].

Bei traditioneller Programmierung legt ein Mensch Regeln festlegt, um bei einem bestimmten Eingang einen bestimmten Ausgang zu erzielen. Bei Machine Learning ist es gewissermaßen umgekehrt, da der Algorithmus oder das neuronale Netz die Regeln hier anhand von Daten am Eingang und Ausgang selbst erlernt.

Ein künstliches neuronales Netz, kurz *KNN*, besteht typischerweise aus in Schichten gegliederten künstlichen Neuronen, die von Schicht zu Schicht miteinander verbunden sind. Die Neuronen besitzen eine Aktivierungsfunktion, die bestimmt wann, also bei welchen Eingangssignalen, ein Neuron aktiv wird. Die Architektur des Netzwerks ist bestimmt durch die Anzahl der Schichten, die Anzahl der Neuronen pro Schicht und die Verbindungen zwischen den Neuronen. Grundsätzlich hat jedes Netzwerk eine Eingangs- und Ausgangsschicht, dazwischen befinden sich sogenannte versteckte Schichten.

Die Verbindungen zwischen den Neuronen haben Gewichte, die durch Lernen verändert werden können. Um das zu erreichen, werden während eines Trainingsvorgangs sogenannte Trainingsdaten an die Eingangsschicht gelegt. Danach wird die Rückgabe der Ausgangsschicht mit dem erwarteten Ausgang verglichen. Schließlich werden die Gewichte mit Hilfe von Fehlerrückführung angepasst, um den entstandenen Fehler zu minimieren.

Hat ein Netzwerk mehr als eine versteckte Schicht, wird von einem tiefen neuronalen Netzwerk und von *Deep Learning* gesprochen. Mit tiefen Netzen können komplexere Aufgaben gelöst werden. Für einfache Netze müssen beispielsweise die relevanten Eigenschaften der Daten erst von Hand extrahiert werden. Deep Learning zeichnet sich auch dadurch aus, dass dieser Extraktionsprozess selbst erlernt werden kann. Das bedeutet auch, dass es mit großen Mengen von unstrukturierten Daten trainiert werden kann, ohne vorher großen Aufwand bei der Datenaufbereitung verursacht zu haben.

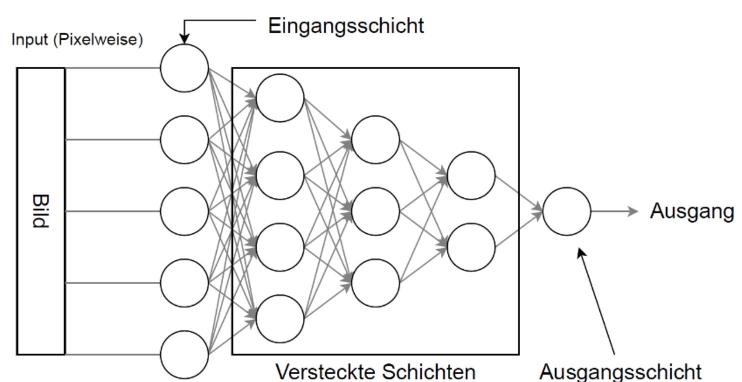


Abbildung 4: Beispiel für ein KNN aus vollständig verbundenen Schichten (vereinfacht)

4.2 Convolutional Neural Networks (CNN)

Was der Begriff bedeutet und um was für eine Art Netzwerk es sich handelt

Die Informationen zu den nachfolgenden Erklärungen sind nachzulesen im *Machine Learning Glossary* der *Google Developers* Website [9].

Eine Faltung, im Englischen sowie in der Fachsprache *Convolution* genannt, ist eine Matrixoperation, bei der eine Matrix über eine zweite geschoben und dabei der überlappende Abschnitt elementweise multipliziert und aufsummiert wird. Angenommen die erste Matrix stellt einen Filter und die zweite Matrix die Pixelwerte eines Bildes dar, so bewirkt die Convolution eine Extraktion der Filtereigenschaften aus dem Bild. Diese extrahierten Eigenschaften werden auch Features genannt.

Die als *convolutional Layers* bezeichneten Schichten in einem *convolutional neural Network* gehen nach diesem Prinzip vor. Die Dimension des Filters und die Anzahl der unterschiedlichen Filter sind dabei frei wählbar. Die einzelnen Einträge in den Filtermatrizen, welche die zu extrahierende Bildeigenschaft charakterisieren, werden anfangs zufällig gewählt und während des Trainingsvorgangs verändert. Dieser Vorgang ist vergleichbar mit der Anpassung von Gewichten eines normalen neuronalen Netzwerks, wie es im Kapitel zuvor bereits erklärt wurde.

Der Aufbau eines convolutional neural Network ist wie folgt. Es beinhaltet auch sogenannte *pooling Layer*, die sich im Netzwerk jeweils hinter einem convolutional Layer befinden. Convolutional und pooling Layer wechseln sich also ab. Die letzten Schichten des Netzwerks sind die sogenannten *dense Layer* oder *densely connected Layer*.

Die pooling Layer reduzieren die Dimension der resultierenden Matrix nach einer Convolution entweder durch Mitteln oder wählen des Maximums. So wird von Schicht zu Schicht die meist unwichtige Information über die Position eines Features herausgefiltert, die Information über dessen Existenz dagegen bleibt erhalten. Das bewirkt, dass die Features nicht für jede mögliche Position in einem Bild neu gelernt werden müssen, weshalb sich CNNs besonders in Vision-Anwendungen durchgesetzt haben.

Die dense Layer sind normale, vollständig miteinander verbundene Schichten aus Neuronen, wie sie auch in normalen KNNs eingesetzt werden. Sie klassifizieren die durch Convolution extrahierten Features und liefern am Ausgang das Ergebnis der Klassifikation.

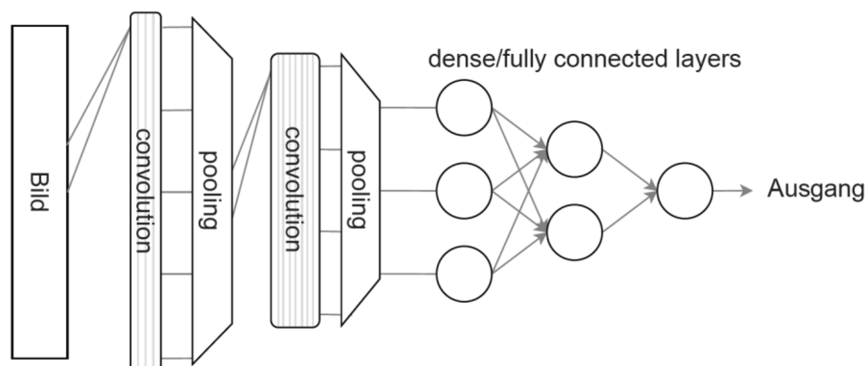


Abbildung 5: Beispiel für ein CNN (vereinfacht)

4.3 Software der TQ-Module

Die auf ARM® oder x86 Architektur basierenden TQ-Module laufen üblicherweise mit einem Linux-Betriebssystem. Um möglichst effizient an die unterschiedlichen Hardware- aber auch Anwendungsanforderungen angepasste Linux-Distributionen bauen und bereitstellen zu können, verwendet TQ das in der Industrie verbreitete Build-System *Yocto Project*, welches in einer Art Schichtsystem arbeitet [13].

Ähnlich wie bei künstlichen neuronalen Netzen, werden die Schichten dabei ebenfalls oft als *Layer* bezeichnet.

Wird zum Beispiel ein Prozessor von NXP eingesetzt, wie es in TQ-Modulen mit ARM® Architektur in der Regel der Fall ist, wird die vom Hersteller des Prozessors bereitgestellte Hardwareschicht verwendet. Der Layer *meta-tq*, welcher die Maschinenkonfiguration für das TQ-Modul bereitstellt, ist von dieser Hardwareschicht abhängig [14]. Zusammen mit der Referenzdistribution werden die Hardwareschicht samt den benötigten Softwarekomponenten und der Treiber in ein sogenanntes *BSP*, kurz für *Board Support Package*, zusammengefasst.

Je nach Anforderung, können dann im *OpenEmbedded Layer Index* [15] aufgeführte, schon existierende Layer hinzugefügt werden. Ebenso können neue Layer erstellt werden, zum Beispiel für die Applikation.

5 Related Work

5.1 Der „End-to-End Deep Learning"-Ansatz von Nvidia

Wie in Kapitel 2 bereits erwähnt, wird im Rahmen der Arbeit ein selbstfahrendes Modellfahrzeug entwickelt, das einem Ansatz aus dem Nvidia Developer Blog folgt.
Der Ansatz wird in dem Artikel *End-to-End Deep Learning for Self-Driving Cars* [16] erklärt. *End-to-End Deep Learning* bedeutet, dass ein tiefes neuronales Netz die Bilder einer Kamera bekommt und ein Steuersignal zurückgibt ohne das dazwischen irgendetwas explizit programmiert werden muss.

Diese Arbeit wird das Experiment von Nvidia deutlich verkleinern und in gewisser Weise auch vereinfachen. Das soll zeigen, dass eine ML-Anwendung nicht schwieriger zu entwickeln ist als andere Anwendungen. Auch der Aufwand hält sich hierbei in Grenzen, es handelt sich schließlich um eine Einmannentwicklung.

Das Vorbild von Nvidia wird in den folgenden Unterkapiteln strukturiert und knapp erklärt.

5.1.1 Das Sammeln der Trainingsdaten

Es wurden Daten in Form von Kamerabildern und Lenkeinschlagswinkeln gesammelt. Sie sollten dem neuronalen Netz zeigen können, wie Menschen auf verschiedenen Straßen unter verschiedenen Bedingung fahren.

Die Fahrzeuge für das Sammeln von Trainingsdaten trugen insgesamt drei Kameras hinter den Windschutzscheiben, eine zentrale Kamera, sowie jeweils eine links und eine rechts. Die Bilder dieser Kameras wurden während der Fahrt in einem 100ms Takt zeitsynchron mit dem aktuellen Lenkeinschlag des Fahrzeugs aufgezeichnet. Die dabei gesammelten Daten entsprechen etwa 70 Fahrstunden aus mehreren Fahrzeugen und verschiedenen Fahrern.

Durch die äußeren Kameras standen Bilder für zwei spezielle Abweichungen von der Straßenmitte direkt zur Verfügung. Sie wurden, unter entsprechender Korrektur des aufgezeichneten Lenkeinschlages, als zusätzliche Datenpunkte verwendet. Weitere Datenpunkte, die das Fahrzeug in anderen Abweichungen und Rotationen zur Straße zeigen, wurden mit einfacher Blickwinkeltransformation erzeugt.

Nvidia zufolge ist es wichtig, dem neuronalen Netz auch zu zeigen, wie Abweichungen vom Ideal zu korrigieren sind. Nur so sei es in der Lage, seine Fehler zu korrigieren. Wäre dieser Schritt weggelassen worden, hätte das ein langsames Abdriften von der Straße zur Folge gehabt.

5.1.2 Die Architektur des neuronalen Netzwerks von Nvidia

Bei dem verwendeten Netzwerk handelt es sich um ein convolutional neural Network. Das Modell wird im Folgenden vom Eingang bis zum Ausgang erklärt. In der Abbildung 6 rechts ist die Modellarchitektur grafisch beschrieben. Der Eingang befindet sich unten, der Ausgang oben.

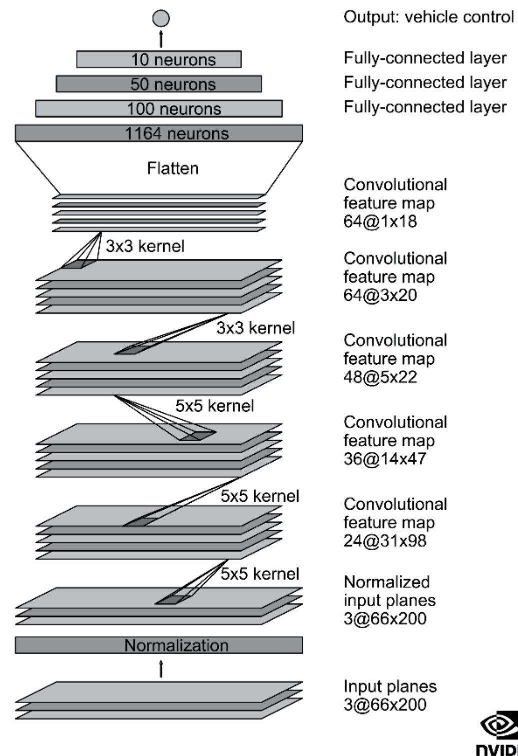


Abbildung 6: Nvidia Modellarchitektur [16]

Da für die Erkennung einer Straße keine allzu hohe Auflösung nötig ist, hat der Eingangstensor des Modells die Dimension von 66x200x3. Das Kamerabild muss also unter Einhaltung des ursprünglichen Seitenverhältnisses auf eine Breite von zweihundert Pixeln runterskaliert werden. Im Netz verwendet werden anschließend nur die untersten sechsundsechzig Pixelreihen, denn nur der untere Teil des Kamerabildes, dort wo sich die Straße abzeichnet, kann relevante Information enthalten. Das Bild muss außerdem drei Farbkanäle haben.

Bei der Angabe von Dimensionen eines Bildes wird sich diese Arbeit weiterhin daran halten, die Höhe vor die Breite zu stellen. Das ist auch die Reihenfolge, in der Bilder in Arrays angeordnet werden. Die Angabe mittels Breite vor Höhe, wie beispielsweise bei Bildschirmauflösungen üblich, ist hier weniger geeignet.

Um die Effizienz des Modells zu steigern, werden die Bilder nicht im typischen RGB-Farbraum, sondern im *YUV*-Farbraum verarbeitet. Das ist bei Vision-Applikationen der allgemein nützlichere und daher auch gängigere Farbraum. Der Hauptgrund dafür ist, dass bei YUV die Helligkeit eines Pixels von nur einem Farbkanal abhängt, während sie sich bei RGB nur aus der Kombination aller drei Kanäle ergibt.

Das Netzwerk hat außerdem eine Normalisierungsschicht direkt nach dem Eingang. Ein Kamerabild mit einer Farbtiefe von 8 Bit hat üblicherweise Werte von 0 bis 255. Eine Normalisierung bringt den gesamten Wertebereich zwischen 0 und 1 in Gleitkommazahlen, was beispielsweise im Fall eines Bildes mit 8 Bit Farbtiefe bedeutet, dass durch 255 geteilt wird.

Wie im Bild zu sehen, hat das Netz insgesamt fünf convolutional Layer. Die ersten drei mit Filter der Dimension 5x5, die letzten beiden mit 3x3. Um die Dimension zu reduzieren, wird bei den ersten drei Faltungen ein sogenannter *Stride* von 2x2 eingesetzt, statt pooling Layer zu verwenden. Das bedeutet, dass die 5x5-Filter bei der Faltung mit einem Zweierschritt über das Bild geschoben werden. Der Vorteil dieses Vorgehens ist, dass so die Anzahl der nötigen Faltungen um den Faktor vier reduziert wird, und damit auch der Rechenaufwand des Modells.

Mit zunehmender Tiefe des Modells wird nicht nur die Dimension reduziert, sondern auch die zu erkennenden Features werden immer komplexer. Daher steigt auch die Anzahl der Filter mit fast jeder Schicht. Im ersten Layer sind es vierundzwanzig verschiedene Filter, im letzten vierundsechzig.

Bevor die Daten im Netzwerk weiterverarbeitet werden können, muss eine *flatten*-Operation folgen, denn im Gegensatz zu einem convolutional Layer hat der Eingang eines dense Layer immer nur eine Dimension. Diese Operation macht einen eindimensionalen Tensor aus dem mehrdimensionalen Ausgang der convolutional Layer. Es spielt dabei keine Rolle, wie genau die Dimensionen des ursprünglichen Tensors umgeordnet werden, da letztendlich jeder Eintrag des resultierenden Tensors mit jedem Neuron der folgenden Schicht verbunden ist. Die letzten Schichten klassifizieren die extrahierten Features und liefern einen Steuerwinkel am Ausgangsneuron in der letzten Schicht.

5.1.3 Datenvorbereitung, Trainingsvorgang und Evaluierung des Modells

Wie bereits in Kapitel 5.1.1 angesprochen, hat Nvidia mit Hilfe der beiden äußeren Kameras und mit Blickwinkeltransformation zusätzliche Datenpunkte erzeugt. Mit diesen wurde dem Modell das Korrigieren einer fehlerhaften Position oder Rotation zur Straße beigebracht. Da nur Spurhalten trainiert werden sollte, mussten alle Daten entfernt werden, bei denen beispielsweise die Spur gewechselt oder abgebogen worden war.

Bevor es auf der Straße getestet wurde, ist das trainierte Modell außerdem in einem Simulator evaluiert worden. Der Beschreibung nach lässt dieser Simulator das Modell den Steuerwinkel schätzen, berechnet dann die nächste Position des Fahrzeugs und simuliert daraus das resultierende neue Kamerabild mit Blickwinkeltransformation. Bei einer Abweichung von über einem Meter von der Mitte der Straße, wird ein menschlicher Eingriff simuliert, der das Fahrzeug wieder in die Mitte setzt. Die Anzahl der menschlichen Eingriffe bestimmt die Qualität des trainierten Modells.

5.2 Der Ansatz von Nvidia im Kleinformat – „DeepPiCar“

Da der im Kapitel zuvor erklärte Ansatz von Nvidia schon im Jahr 2016 veröffentlicht wurde, existieren bereits Versuche, diesen mit einem kleinen Modellfahrzeug nachzuahmen. Einen solchen Versuch beschreibt der Artikel *DeepPiCar — Part 5: Autonomous Lane Navigation via Deep Learning* [17] von 2019. Wie dort vorgegangen wurde, wird im Folgenden erklärt. Aufgrund der Gemeinsamkeiten bei den Versuchsaufbauten, wurde der Artikel zu Beginn der Arbeit vor allem als Bestätigung für die Umsetzbarkeit der festgelegten Ziele gesehen.

5.2.1 Vorgangsweise im Vergleich zu Nvidia

Das verwendete Modellfahrzeug ist ein *PiCar* vom Hersteller *SunFounder*. Es wird mit einem dort aufsteckbaren *Raspberry Pi* gesteuert. Für ein größeres Sichtfeld entschied sich der Autor des Artikels, die bei dem PiCar mitgelieferte 120° durch eine 170° Weitwinkelkamera zu ersetzen.

Um das Fahrzeug beim Sammeln der Daten zu steuern, hat der Autor des Artikels eine zuvor mit konventionellen Methoden, also ohne Machine Learning, programmierte Software

verwendet, mit der das Fahrzeug in der Lage ist, einer Stecke aus blauem Klebeband zu folgen. Das dort trainierte neuronale Netz hat das Fahren also nicht direkt von einem Menschen, sondern nur von einem händisch programmierten Algorithmus gelernt. Die Lenkeinschlagswinkel wurden dort direkt im Dateinamen der zugehörigen Bilder abgespeichert.

Für das Training des Modells wurde dort das ML-Framework *TensorFlow* in seiner ersten Version verwendet und in Python programmiert. Der Artikel beschreibt außerdem, welche zufälligen Bildaugmentationen verwendet wurden, um aus der recht kleinen Datenbasis von ungefähr 200 Bildern künstlich neue Daten zu generieren. Diese beinhalten das Verändern der Helligkeit, Zoomen, Schwenken und Weichzeichnen, aber auch das Spiegeln der Lenkeinschlagwinkel und des jeweils zugehörigen Bildes an der vertikalen Achse.

Die Netzwerkarchitektur wurde in TensorFlow getreu dem Vorbild von Nvidia nachgebaut, nur die Normalisierungsschicht wurde weggelassen, da die Daten schon normalisiert vorliegen sollten. Es sind außerdem zusätzlich sogenannte *Dropout*-Schichten eingebaut worden.

Da von Nvidia selbst nicht erwähnt wird, welche Aktivierungsfunktion in ihrem neuronalen Netzwerk zum Einsatz kommt, hat der Autor die Funktion *Exponential Linear Unit*, kurz *ELU*, verwendet. Er erklärt, dass die gängige Alternative *ReLU*, kurz für *Rectified Linear Unit*, sogenannte tote Neuronen verursacht. Wie nachzulesen im *ML Glossary* auf der Website *readthedocs.io* [18], entstehen tote Neuronen, wenn der Eingang des Neurons kleiner als null und so wegen der ReLU-Funktion der Ausgang gleich null wird. Die Gewichte von Neuronen, die null am Ausgang liefern, können bei der Fehlerrückführung nicht mehr angepasst werden.

Das Trainieren von Fehlerkorrekturmaßnamen, durch die Bereitstellung der Daten von drei Kameras, wurde im Artikel ignoriert. Die Folge daraus war, genau wie Nvidia in ihrer Veröffentlichung für diesen Fall vorhersagt, das langsame Abweichen von der Straße: „...doing fine most of the way but veered off the lane a bit toward the end.“ [17]. Übersetzt: „...fuhr gut den größten Teil des Weges, aber ist am Ende etwas von der Spur abgedriftet.“. Der Artikel kommt zu dem Ergebnis, dass dies auf eine zu geringe Menge an Trainingsdaten zurückzuführen wäre, was jedoch nicht der Fall ist. Die Ergebnisse dieser Arbeit werden diese Behauptung ebenfalls unterstützen, siehe Kapitel 7.2.

5.3 Diese Bachelorarbeit im Vergleich

Ein Ziel der Arbeit ist es, den Ansatz von Nvidia so detailgetreu wie möglich mit TQ-Hardware umzusetzen. Da ein Student für eine Bachelorarbeit vergleichsweise wenige Ressourcen zur Verfügung hat, sind gewisse Einschränkungen zu erwarten.

Der erste und offensichtlichste Unterschied zu dem Experiment von Nvidia ist, dass hier statt einem PKW auf einer echten Straße, nur ein Modellauto auf einer Teststrecke aus Klebeband fährt, ebenso wie das DeepPiCar.

Die Wahl des fahrbaren Untersatzes ist ein nicht trivialer Schritt und kann, bei einer schlechten Wahl, durchaus unvorhersehbare Schwierigkeiten verursachen. Daher wird hier

dieselbe Hardware für das Modellfahrzeug verwendet, wie für das DeepPiCar. Die Empfehlung, die mitgelieferte 120° durch eine 170° Weitwinkelkamera zu ersetzen, wird ebenfalls berücksichtigt.

Grundsätzlich hält sich diese Arbeit genauer an die Vorgehensweise von Nvidia, als der Versuch aus Kapitel 5.2, denn die Daten werden mit drei Kameras gesammelt, nicht nur mit einer. Wie vorher angesprochen ist es entscheidend, dem Modell beizubringen, wie es Fehler zu korrigieren hat. Um die Komplexität des Unterfangens zu reduzieren, wird jedoch auf Blickwinkeltransformation verzichtet. Es werden lediglich die Daten der äußeren Kameras mit einer festen Abweichung vom aufgezeichneten Lenkeinschlag für das Trainieren der Fehlerkorrektur verwendet.

Das Testen des Modells in einem Simulationsprogramm vor dem Einsatz auf der Strecke soll hier ebenfalls übersprungen werden. Die Modelle werden stattdessen direkt nach dem Training mit der Zielhardware auf der abgeklebten Teststrecke evaluiert.

Bei dieser Arbeit soll das Modell tatsächlich von einem Menschen lernen, wie es zu fahren hat, nicht von einem Algorithmus. Daher wird das Modellauto während dem Sammeln der Daten ferngesteuert.

Die Lenkeinschlagswinkel werden im Dateinamen der aufgezeichneten Bilder festgehalten, wie das der Artikel aus Kapitel 5.2 vormacht. Ebenso übernommen wird die Verwendung von ELU-Aktivierungsfunktionen im neuronalen Netzwerk. Die Netzwerkarchitektur wird ebenfalls ohne Normalisierungsschicht in TensorFlow nachgebaut und trainiert, verwendet wird aber bereits das TensorFlow-Framework in seiner zweiten Version. Auch die Daten werden zum größten Teil anders vorbereitet, weshalb kein Code von diesem Versuch übernommen wurde.

Ein Teil der Datenvorbereitung, der für das Training des DeepPiCar eingesetzt wurde, wird in dieser Arbeit aufgegriffen und weiterentwickelt. Genau geht es dabei um das Spiegeln der Lenkwinkel und Bilder. Dem Vorbild nach soll bei jeder Iteration des Trainings zufällig entschieden werden, ob der Datenpunkt gespiegelt wird oder nicht. Bei dieser Arbeit wird das systematischer umgesetzt, indem jeder der verfügbaren Datenpunkte sowohl einmal normal als auch gespiegelt vorliegen wird.

Im Modell wird ebenfalls ein Dropout eingesetzt. Der genannte Artikel ist aber nicht als Vorbild dafür zu sehen, da es sich hierbei um eine gängige Praxis handelt. Was ein Dropout überhaupt ist, wird in Kapitel 6.3 näher erklärt.

Die Menge der gesammelten Trainingsdaten ist deutlich größer als die, mit denen das DeepPiCar trainiert wurde. Es wurden aber dennoch weniger Daten gesammelt als bei dem Experiment von Nvidia. Das Modell wird auch nur auf eine eindeutig erkennbare Fahrbahnmarkierung trainiert. Nvidia hingegen hat beispielsweise auch Daten verwendet, die dem Modell das Steuern des Fahrzeugs durch parkende Autos auf einem Parkplatz zeigen sollen.

6 Durchführung

Da diese Arbeit mit einem der TQ-Module zu tun hat, wurde sie in der Abteilung *Embedded Support* durchgeführt. So steht die nötige Unterstützung für Yocto Project jederzeit zur Verfügung. Dieses Kapitel behandelt zunächst die nötigen Vorbereitungen und schließlich die Durchführung des Vorhabens.

6.1 Vorbereitung

Das BSP für das TQMa8MPxL beinhaltet standardmäßig nicht mehr als die für die wichtigsten Funktionen notwendigen Komponenten. So wird Speicherplatz gespart und der Softwarepflegeaufwand minimiert. Deshalb müssen einige Layer für die Nutzung speziellerer Funktionen, wie beispielsweise Machine Learning, erst noch eingebunden werden. Für das Einbinden aller nötigen Layer, sowie das Zurechtfinden mit dem Yocto Build-System, war die Unterstützung durch den Embedded Support maßgeblich.

Der ausschlaggebende Layer für Machine Learning ist der *meta-ml*, welcher von NXP unter dem Layer *meta-imx* veröffentlicht wurde [19]. Es bestehen zudem Abhängigkeiten zu den ebenfalls aus meta-imx stammenden Layern *meta-bsp* und *meta-sdk*, sowie zu *meta-freescale-distro* und *meta-python2*, weshalb diese ebenfalls in das Build-System eingebunden werden müssen.

Es stellte sich heraus, dass für Beschleunigung durch die NPU, diese erst mit einem sogenannten *Kernelpatch* aktiviert werden muss. Die Beschleuniger sind in dem von NXP bereitgestellten Layer standardmäßig deaktiviert. Aus diesem Grund konnten die neuronalen Netze anfangs nur mit der CPU berechnet werden. Die Lösung für dieses Problem wurde für längere Zeit nicht gefunden, weshalb ein Austausch mit dem Support von NXP, und schließlich mit der BSP-Entwicklung bei TQ nötig war.

Als fahrbarer Untersatz wird ein Modellbaufahrzeug namens *PiCar* vom Hersteller *Sunfounder* verwendet. Es ist für die Nutzung mit einem Raspberry Pi vorgesehen. Wie bereits in Kapitel 5.3 angesprochen, wurde die Eignung dieser Hardware für das Vorhaben bereits bestätigt. Lediglich die Übertragbarkeit der Steuerung von einem Raspberry Pi auf ein TQ Starterkit musste vorher überprüft werden.

Das PiCar bietet zur Steuerung eine API in Python, welche auf GitHub verfügbar ist [20]. Nach einer gründlichen Inspektion der enthaltenen Dateien ist erkennbar, dass für die Kommunikation mit dem Fahrzeug das Python-Paket *python3-smbus* für die Regelung des Lenkeinschlages und der Geschwindigkeit benutzt wird. Für das Umschalten zwischen Vorwärts- und Rückwärtsgang werden einfache *GPIOs*, kurz für *General Purpose Input Output*, verwendet. Da das Fahrzeug in seiner vorgesehenen Anwendung nicht rückwärtsfahren wird und es nach Standardeinstellung Vorwärts fährt, werden die GPIOs nicht benötigt. Für die Verwendung auf dem Starterkit wurden die Zeilen mit Bezug zu den GPIOs im Code der API daher auskommentiert.

Das Paket python3-smbus ist im *OpenEmbedded Layer Index* [15] aufgelistet und kann daher in das Betriebssystem des TQ-Moduls mit eingebaut werden. Es arbeitet im Hintergrund mit einer I2C-Schnittstelle. Daher muss lediglich ein freier I2C-Bus ausgewählt

werden, welcher auf dem Starterkit über Pins oder einen Stecker zugänglich ist. Ebenfalls benötigt werden die Pakete *python3-numpy*, *v4l-utils* und *python3-pip*. Letzteres ermöglicht es, auf dem laufenden System weitere Python-Pakete herunterzuladen. In den Build miteingebaut wird natürlich auch *packagegroup-imx-ml* aus dem zuvor in das Build-System eingebundenen Layer meta-ml.

Zuletzt müssen auf dem laufenden Starterkit noch einige wenige Schritte durchgeführt werden. Die modifizierte API zur Steuerung des PiCar muss installiert werden, damit sie in Python verwendet werden kann. Zuletzt wird mit Hilfe des Python-Moduls *pip3* das Python-Paket *opencv-python-headless* heruntergeladen und installiert.

Die hier beschriebenen Vorbereitungen wurden im firmeninternen Wiki dokumentiert. Zusätzlich wurde als grober Leitfaden für die Funktionsweise des Fahrzeugs eine Entwicklungsdokumentation im Wiki des zugehörigen *GitLab*-Repository erstellt. Beide Dokumentationen wurden zu PDF-Dateien gedruckt und sind im Zusatzmedium zu finden. Sie heißen dort *Installation auf STKa8MPxL.pdf* und *Entw. Doku.pdf*.

6.2 Sammeln der Trainingsdaten

6.2.1 Hardware

Für eine zweckgemäße Nachbildung des Fahrzeugs von Nvidia, muss es mit Weitwinkelkameras versehen werden und von einem Menschen steuerbar sein. Das Anbringen von mehr als einer Kamera ist nicht ohne weiteres möglich. Da für das Sammeln der Trainingsdaten jedoch drei benötigt werden, wurde eine Art Halterung aus einer zugeschnittenen Lochrasterplatine gebaut. Dort können alle drei Kameras befestigt werden. Sie benötigen außerdem je eine USB-Schnittstelle.

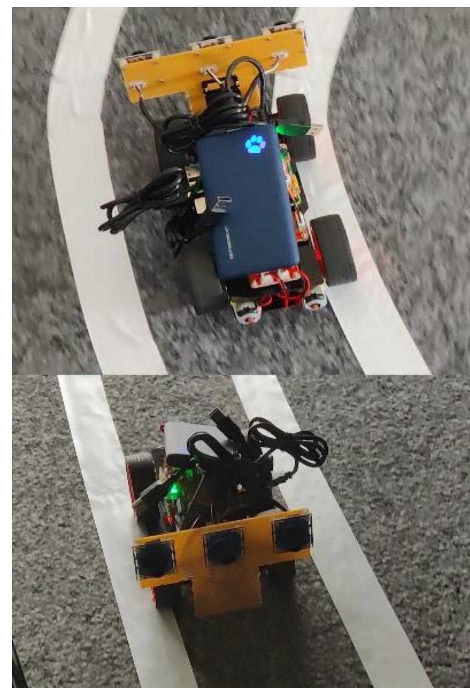


Abbildung 7: Data-Collection-Fahrzeug

Um Bilder für das Training zu sammeln, wurde erst ein *Raspberry Pi 4* verwendet, wie für das PiCar eigentlich vorgesehen. Da dieser mit Bluetooth ausgestattet ist, kann ohne besonderen Aufwand ein *PlayStation® 4 Controller* verbunden werden. Dieser hat zwei *Analogsticks*, also analoge Steuerhebel, mit denen das Fahrzeug sehr präzise ferngelenkt werden kann. Damit das auch funktioniert, muss ein entsprechendes Programm geschrieben und auf dem Raspberry Pi zum Laufen gebracht werden.

Der Raspberry wird durch die im PiCar verbauten Batterien mit Strom versorgt. Diese Versorgung reicht jedoch nicht mehr aus, sobald alle drei Weitwinkelkameras verbunden sind. Das macht sich in der Steuerung des Fahrzeugs bemerkbar, da es nur noch sehr langsam auf die Steuersignale des Controllers reagiert. Der Raspberry Pi muss deshalb zusätzlich über den USB-C-Anschluss extern von einer Powerbank versorgt werden.

6.2.2 Software und Steuerung des Fahrzeugs

Nach dem Aufbau der Hardware muss dafür gesorgt werden, dass der Raspberry Pi die für das Sammeln des Trainingsmaterials nötigen Aufgaben ausführt.

Das Betriebssystem des Raspberry ist mit einer grafischen Oberfläche ausgestattet. Ist er einmal mit einem Wifi-Netzwerk verbunden, kann mit einer Software für Remotedesktopzugriff, hier verwendet wurde *AnyDesk* [21], aus der Ferne auf die Desktopumgebung zugegriffen werden. So lassen sich sehr einfach und schnell alle notwendigen Aktionen von einem Laptop aus durchführen, ohne ein Kabel mit dem Fahrzeug verbinden zu müssen. Zu diesen Aktionen gehören die Bluetooth-Verbindung zwischen dem Raspberry Pi und dem Controller herzustellen, Skripte zu starten, stoppen und verändern, sowie hinterher die von dem Fahrzeug aufgenommenen Bilder zu übertragen.

Die Fahrzeugsteuerung mithilfe der Python-API, die Kommunikation mit dem Controller und das Sammeln von Daten der drei Kameras mit zugehörigem Lenkwinkel werden in einem einzigen Python-Skript gelöst. Die Datei trägt den Namen *collectTrainImages.py* und ist im Zusatzmedium zu dieser Arbeit einsehbar.

Die API des PiCar akzeptiert den Lenkinput in Form eines Steuerwinkels. Ein Winkel von 90° bedeutet hierbei Geradeaus, kleinere Winkel bedeuten einen Einschlag nach links und größere einen Einschlag nach rechts. Der mechanisch maximal mögliche Lenkeinschlag des Fahrzeugs beträgt nicht viel über 45 Grad von der Mittelstellung abweichend. Da es nicht möglich ist, die genaue Kalibrierung der Lenkung nur mechanisch durchzuführen, muss hierfür in der Software ein Offset addiert werden. Die Geschwindigkeit wird mit Werten zwischen 0 und 100 geregelt. Da die Kraft der Motoren bei Werten unter 15 noch nicht genügt, um das Fahrzeug fortzubewegen, wird der Wertebereich auf 15 bis 100 beschränkt.

Um die USB-Kameras in Python auslesen zu können, sowie um die Bilder abzuspeichern, wird die dafür gängige *Computer Vision* Bibliothek *OpenCV* [22] verwendet. Im Python-Code trägt sie die Bezeichnung `cv2`. Die Auflösung der Kameras wird auf 240x320 Pixel gestellt, da die Bilder nur mit einer Auflösung von 66x200 gespeichert werden müssen. Werden alle drei Kameras auf ihre volle Auflösung von 480x640 Pixeln gestellt, sorgt das aufgrund zu hoher Bandbreite für Probleme. Die Bilder werden in einer Dauerschleife in festgelegte Ordner gespeichert. Der Lenkeinschlag wird im Dateinamen des jeweiligen Bildes festgehalten. Die maximale Anzahl der Bilder, die pro Sekunde gespeichert werden können, werden auf 10 begrenzt, genau wie im Vorbild von Nvidia.

Um den mit Bluetooth verbundenen Controller verwenden zu können, wird die Bibliothek *pyPS4Controller* verwendet, die Dokumentation inklusive Beispiel sind auf *PyPi* [23] zu finden. Durch die Ableitung von der dort enthaltenen Klasse `Controller`, lassen sich Methoden definieren, die bei Betätigung von Tasten oder beim Bewegen der Analogsticks ausgeführt werden.

Das Sammeln der Bilder und die Kommunikation mit dem Controller geschehen nebenläufig in Threads. Für die notwenige Kommunikation zwischen den Threads werden globale Variablen verwendet.

Die Lenkung des Fahrzeugs wird mit dem linken Analogstick, bezeichnet als *L3*, gesteuert. Die Geschwindigkeit wird mit der linken und rechten Schultertaste gesteuert. Die rechte Taste, bezeichnet als *R2*, bewegt das Fahrzeug vorwärts. Die linke Taste, bezeichnet als *L2*, bewegt es rückwärts. Da R2 und L2 analoge Tasten sind, kann die Geschwindigkeit in kleinen Stufen geregelt werden. Zusätzlich wurde programmiert, dass sich mit der digitalen Schultertaste *R1* die Geschwindigkeit auf den Wert 30 fixieren lässt, um beim Sammeln der Trainingsbilder einfacher eine konstante Geschwindigkeit halten zu können.

Abbildung 8: Steuerung mit PlayStation® Controller

Um die Trainingsbilder letztendlich aufnehmen zu können, wurden 2 Möglichkeiten programmiert. Solange die X-Taste gedrückt wird, werden kontinuierlich Bilder gesammelt. Bei einmaligem betätigen der Kreis-Taste wird hingegen genau einmal ein Bild von jeder der drei Kameras gespeichert. Die Funktionalität einzelne Bilder aufzunehmen war vorgesehen, um nach Bedarf möglichst einfach einzelne Bilder von spezielleren Abweichungen von der Straßenmitte machen zu können, zum Beispiel durch händisches Setzen des Fahrzeugs auf eine bestimmte Position. Tatsächlich wurde von dieser Funktion nie Gebrauch gemacht.

Gesammelt wurden die Bilder auf einer Strecke, die mit weißem Klebeband auf einem grauen Teppich aufgeklebt worden war. Diese wurde jeweils zwei Mal in jede Richtung abgefahren wobei 3340 Bilder je Kamera gesammelt wurden. Anschließend wurde der Kameraaufbau des Fahrzeugs für das selbstständige Fahren umgebaut. Das benötigt nur noch eine Kamera, nur für das Sammeln der Trainingsdaten werden drei Kameras benötigt. Das wird in Kapitel 6.4 aufgegriffen und näher beschrieben. Aus mechanischen Gründen musste die Position der einzelnen Kamera gegenüber der mittleren des vorherigen Aufbaus geändert werden. Deshalb wurde die Strecke erneut einmal in jede Richtung abgefahren und dabei weitere 1621 Bilder mit nur einer Kamera gesammelt. Durch diese Daten soll dem Modell die leicht veränderte Kameraposition angelernt werden.

6.3 Training

Das Training des neuralen Netzwerks läuft in der auf dem Zusatzmedium vorhandenen Datei *NvidiaModelTrain.py* ab. Sie ist auf einem Desktopcomputer mit der vollen Installation von TensorFlow in Python auszuführen. Der Trainingsvorgang kann auch mit einer schnellen Grafikkarte beschleunigt werden. Aufgrund der noch relativ kleinen Datenbasis, ist der hierbei verwendete Prozessor vom Typ *Intel Core i7-8700* jedoch ausreichend. Für den Fall, dass keine ausreichend schnelle Hardware zur Verfügung steht, lassen sich ML-Modelle alternativ auch Servergestützt trainieren. Ein nennenswertes Beispiel ist hierbei der kostenlose Dienst *Google Collab*, welcher auch Grafikkartenbeschleunigung anbietet.

Um das Modell des Netzwerks trainieren zu können, muss es in TensorFlow dem zuvor beschriebenen Vorbild entsprechend nachgebaut werden. Der grundsätzliche Aufbau wurde oben in Kapitel 5.1.2 erklärt. Wie dasselbe Modell im Code aussieht zeigt der folgende

Codeabschnitt. Im Gegensatz zu der Darstellung von Nvidia, sind hierbei die Eingangsschicht oben und die Ausgangsschicht unten.

```
nvidia_model = Sequential (
    [
        # Params Conv2D: Number of filters, Dimension, Stride=(1,1)
        layers.Conv2D(24, 5, (2, 2), input_shape=(66, 200, 3), activation='elu'),
        layers.Conv2D(36, 5, (2, 2), activation='elu'),
        layers.Conv2D(48, 5, (2, 2), activation='elu'),
        layers.Conv2D(64, 3, activation='elu'),
        layers.Conv2D(64, 3, activation='elu'),
        layers.Flatten(),
        layers.Dropout(0.05),                          #Dropout (reduce overfitting)
        # Param Dense: Number of Neurons
        layers.Dense(100, activation='elu'),
        layers.Dense(50, activation='elu'),
        layers.Dense(10, activation='elu'),
        layers.Dense(1)
    ],
    name='Nvidia_Model'
)
```

Codeabschnitt 1: Nvidia CNN in TensorFlow nachgebaut (NvidiaModelTrain.py Zeile 232 bis 249)

Die zusätzliche Schicht *Dropout* bewirkt das willkürliche zu null setzen von Ausgängen der vorhergehenden Schicht während des Trainingsvorgangs. Der angegebene Parameter bestimmt den Anteil an Ausgängen der abzuschalten ist. Typisch wären zum Beispiel 10% oder 20%, jedoch gibt es hierbei keine festgelegten Regeln. Nach mehreren Versuchen wurde ein Dropout von 5% als ausreichend erachtet, siehe im Codeabschnitt 1 oben der Parameter von `layers.Dropout(0.05)`.

Das Vorbild hierfür ist das Tutorial zu Bildklassifizierung auf der offiziellen Website von TensorFlow in der Sektion für Fortgeschrittene [10]. Auch die nachfolgenden Erklärungen diesbezüglich basieren auf dieser Quelle. Das verwenden einer Dropout-Schicht kann dem Modell helfen, besser zu generalisieren und sogenanntes *Overfitting* zu vermeiden. Das Phänomen beschreibt den Vorgang, bei dem ein neuronales Netzwerk beginnt triviale Merkmale der Trainingsdaten, wie beispielsweise Rauschen, auswendig zu lernen. Je nach dessen Ausprägung leidet darunter die Fähigkeit, relevante Merkmale zu klassifizieren. Das Auftreten dieses Phänomens kann überprüft werden, indem Teile der Trainingsdaten nicht für das Training verwendet und nur für eine Validierung des Modells vorbehalten werden. Ist mit den Validierungsdaten eine Verschlechterung der Vorhersage zu erkennen, während sie mit den Trainingsdaten weiter besser wird, so tritt Overfitting auf. Man erkennt Verschlechterungen oder Verbesserungen anhand der Abweichung der Vorhersage vom erwarteten Ergebnis, auch als *loss* bezeichnet.

Um möglichst viel Kontrolle über den Trainingsvorgang selber zu bekommen, wurde ein sogenannter *train data generator* geschrieben. Der ist für die Datenvorbereitung und die Bereitstellung der Trainingsdaten zuständig. Wie ein solcher Generator funktionieren muss, ist in gewissem Maße vorgegeben und kann aus Tutorials, wie beispielsweise *Write your own Custom Data Generator for TensorFlow Keras* [24], entnommen werden.

Der Generator wurde so programmiert, dass er bei seiner Erzeugung bereits einen festlegbaren Anteil der Daten als Validierungsdaten abzweigt und sie in einer Liste abspeichert, die dem Generator zuvor als Referenz übergeben wurde. Anschließend wird die Liste an einen zweiten Generator, den *validation data generator*, weitergegeben.

Im Generator müssen auch die in Unterordnern aufgeteilten Trainingsdaten von den drei unterschiedlichen Kameras behandelt werden. Die Ordner werden auf JPEG-Dateien durchsucht, jedem Bild der Lenkwinkel aus dessen Dateinamen extrahiert.

Auf die Lenkwinkel der Bilder von den beiden äußeren Kameras wird hier dann die Korrektur addiert. Wie bereits in Kapitel 5.1 erklärt, wird dem Modell so gezeigt, wie es wieder in die Mitte der Spur zurückzusteuern hat. Der Korrekturwinkel hat in der finalen Version einen Betrag von 10 Grad in Richtung Spurmitte. Dieser Betrag wurde im Verlauf der Arbeit mehrfach angepasst, um die Qualität des resultierenden Modells zu verbessern.

Durch das Erzeugen des Generators entstehen also zwei Listen, die Pfade zu den jeweiligen Bildern der Trainings- und Validierungsdaten und die zugehörigen Lenkeinschlagswinkel beinhalten.

Die zwei Generatoren für Training und Validierung gleichen sich beinahe vollständig. Das Sammeln der Bilderpfade und Lenkwinkel entfällt in dem zur Validierung, da der Trainingsgenerator diese Daten bereits für den Validierungsgenerator ausweist.

Der Zweck der Generatoren ist es, die Daten für das Training und die Validierung bereitzustellen. Diese Daten werden gruppiert in sogenannten *Batches*, aus dem Speicher geladen und an das zu trainierende Modell gegeben. Wie groß die Batches sein sollen, und damit wie viele Bilder pro Iteration gleichzeitig vom Speicher gelesen werden, kann eingestellt werden.

Im diesem Fall ist die tatsächliche Batchgröße, mit der das Modell intern trainiert wird, größer als angegeben. Der Grund dafür sind die Tricks, mit deren Hilfe aus bestehenden Daten neue Datenpunkte generiert werden. Einer wurde bereits in 5.3 angesprochen.

Es werden die Lenkeinschläge und die Bilder an der vertikalen Achse gespiegelt, was möglich und sinnvoll ist, da es sich hier um ein Achsensymmetrisches Problem handelt. Jedes Bild soll einmal gespiegelt und ungespiegelt vorliegen, anders als für das DeepPiCar, beschrieben in Kapitel 5.2. Dort wurden die Bilder bei jeder Iteration zufällig gespiegelt. Durch das systematischere Vorgehen werden sich die Daten, bei denen links beziehungsweise rechts gelenkt wird, anteilig gleichen. Das soll systematische Fehler in eine Richtung, auch *Bias* genannt, möglichst reduzieren.

Darüber hinaus entstand während der Entwicklung der Gedanke, dass der Boden, auf dem das Fahrzeug zum Beispiel bei einer Messe fahren würde, auch hell sein kann und dann schwarzes Klebeband für die Strecke verwendet werden müsste. Die Trainingsbilder beinhalten jedoch nur weißes Klebeband auf dunklem Boden. Um dieses Problem zu lösen, wurde das Netzwerk ebenfalls mit dem Negativ jeden Bildes trainiert. So wird aus heller Strecke auf dunklem Hintergrund, dunkle Strecke auf hellem Hintergrund, ohne die Trainingsdaten tatsächlich erweitern zu müssen.

Da nun jedes Bild gespiegelt, nicht gespiegelt, und jede Version davon negativ und nicht negativ in den Batch geladen werden soll, ist die tatsächliche Batchgröße viermal so groß wie die Anzahl der Bilder, die aus dem Speicher geladen werden.

Das Transformieren von Daten um neue Datenpunkte aus bestehenden zu generieren, wird auch als *data augmentation* bezeichnet [9]. Noch vor der Vervierfachung der Datenpunkte bei der Erstellung eines Batches, werden der Kontrast, die Helligkeit und die Sättigung des Bildes zufällig verändert. Das bewirkt, dass sich die Trainingsdaten auch von Epoche zu Epoche leicht unterscheiden. Die zufällige Modifikation der oben genannten Parameter simuliert das Auftreten unterschiedlicher Lichtverhältnisse.

Als Trainingsepoche bezeichnet man einen Durchlauf des Trainings durch alle verfügbaren Trainingsdaten, unabhängig von zufälligen Augmentationen. Die Iterationen einer Epoche hängen also direkt mit der Anzahl der Datenpunkte zusammen, beziehungsweise entsprechen der Anzahl der Datenpunkte geteilt durch die Batchgröße [9].

Vor Beginn des Trainingsvorgangs wird ein sogenannter *Checkpoint Callback* erstellt. Dieser hat den Zweck, nach einer Trainingsepoche mit darauffolgender Validierung die Gewichte des Modells zu speichern. Das allerdings nur, wenn das Ergebnis der Validierung sich gegenüber dem besten Ergebnis im bisherigen Trainingsverlauf verbessert hat.

Es wird also zum Schluss nur das beste Ergebnis behalten, nach Ende des Trainings der Checkpoint geladen und so alle Epochen, die nur Overfitting verursacht haben, wieder verworfen.

Mit einer Quantisierung kann die Leistung des neuronalen Netzwerks verbessert werden. Dessen Gewichte liegen üblicherweise als 32 Bit große Gleitkommazahlen vor. Darüber hinaus müssen die Gewichte des Modells in 8 Bit großen Integern vorliegen, um auf der NPU überhaupt lauffähig zu sein. Bei einer Quantisierung auf Ganzzahlen nach dem Training, auch als *post training quantization* bekannt, müssen der minimale und maximale Wert der Gleitkommatensoren bekannt sein. Hierfür müssen repräsentative Daten vorliegen, die den typischen Eingang des Modells zeigen, weshalb dafür in der Regel ein Ausschnitt aus den Trainingsdaten verwendet wird [25]. In diesem Fall wird dafür ein weiterer Generator geschrieben, der einen kleineren Teil der Trainingsdaten verwendet. Die Augmentationen beschränken sich dort auf Negativbilder.

Zuletzt wird das trainierte und quantisierte Modell im *tflite*-Format, also für die Nutzung mit TensorFlow Lite, abgespeichert und kann so, auch beschleunigt durch die NPU, auf dem Starterkit laufen.

6.4 Fahren mit Hilfe des trainierten Netzwerks

6.4.1 Hardware

Die trainierten Modelle sind zu Beginn auch mit dem Raspberry Pi und dem Aufbau mit drei Kameras getestet worden. Da das Steuern des Fahrzeugs durch ein Starterkit jedoch zu den Projektzielen gehört und die beiden äußeren Kameras für das Selbstfahren nicht mehr benötigt werden, bedarf es für den finalen Aufbau einiger Änderungen.

Eine einzelne Kamera lässt sich vergleichsweise einfach vorne am Fahrzeug anbringen, siehe Abbildung 9 rechts. Da das PiCar aber für den Raspberry Pi entwickelt wurde und nicht für ein Starterkit von TQ-Systems, war hier erneut Kreativität gefragt.

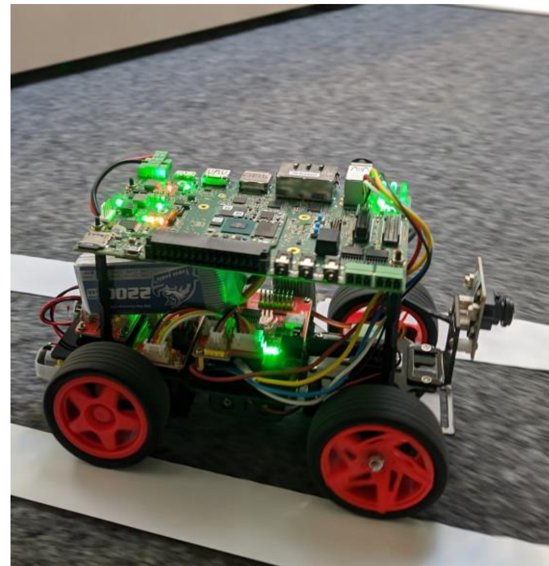


Abbildung 9: selbstfahrendes Fahrzeug

Zuerst musste die Kommunikation zwischen dem PiCar und dem Starterkit hergestellt werden. Als geeigneter I2C-Bus wurde *I2C_1*, unter Linux die Busnummer 0, festgelegt. Die Pins für I2C_1 sind auf einem Stecker, welcher ebenfalls Masse, 5 Volt und 3 Volt Pins beherbergt. Das PiCar ist hardwareseitig sehr wenig bis gar nicht dokumentiert, dennoch war feststellbar, dass es den 5 Volt zu 3 Volt Spannungswandler des Raspberry Pi benutzt. Sobald der Raspberry extern versorgt wird, fangen nämlich die Leuchtdioden auf einigen Komponenten des PiCar an zu leuchten, auch ohne das Fahrzeug vorher einzuschalten.

Anhand der Pinbelegung des Raspberry Pi lässt sich auf die des PiCar schließen. Mit Hilfe dieser Information wurde ein Kabel gebaut, das am entsprechenden Stecker des Starterkits die Pins für I2C_1, Masse, 3 Volt, und 5 Volt mit einer Stiftleiste verbindet. Diese ist der des Raspberry Pi nachempfunden und kann stattdessen in das PiCar gesteckt werden.

Das nächste Problem war die Stromversorgung des Starterkits. Da es einen Weitbereichseingang von 16 bis 24 Volt Spannung besitzt, wurde hierfür ein Modellbau Lithium-Akku mit 5 Zellen und somit einer Nennspannung von 18,5 Volt verwendet.

Für die mechanische Verbindung wurden neue Löcher in das Plastik des Fahrzeugs gebohrt und das Starterkit mit Abstandshaltern so festgeschraubt, dass der Akku darunter noch Platz hat.

Zur Veranschaulichung des Aufbaus wurde ein Blockschaltbild erstellt, siehe Abbildung 10.

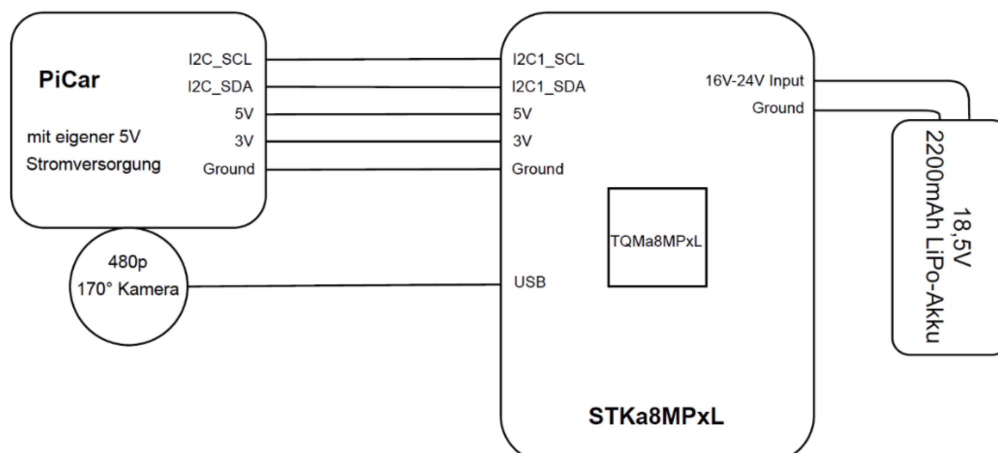


Abbildung 10: Blockschaltbild Aufbau des selbstfahrenden Fahrzeugs mit Starterkit

6.4.2 Software

Das selbstständige Fahren wird in der Datei *drivePicar.py* gesteuert, wie alle Skriptdateien ist sie im Zusatzmedium einsehbar. Grundlegend unterscheidet sich der Aufbau des Programms nicht sonderlich zu *collectTrainImages.py*, welches für das Sammeln der Bilder geschrieben wurde. Die Steuerung durch den Controller und das Abspeichern der Bilder entfällt natürlich.

Die Bilder der Kamera werden zunächst wieder auf eine Auflösung von 66x200 Pixeln skaliert, um an den Eingangstensor des zuvor trainierten Modells gelegt werden zu können. Der Ausgangstensor gibt den Lenkeinschlag aus, zu welchem wieder ein Offset für die Kalibierung addiert wird. Das Resultat wird an die API des PiCar gegeben. So bleibt das Fahrzeug letztendlich auf der Spur.

In die finale Version des Fahrzeugs wurde Objekterkennung implementiert. Da das nicht Teil der zu Beginn festgelegten Projektziele ist, wird die Software des Fahrzeugs erst in Unterkapitel 6.5.3 des nächsten Kapitels wieder aufgegriffen.

6.5 Entwicklungen über die Projektziele hinaus

Während der Entwicklung des Projekts sind zusätzliche Entwicklungen entstanden und das eigentliche Projekt wurde leicht über die Projektziele hinaus entwickelt. Es wurden auch weitere Aufgaben im Zusammenhang mit der hier durchgeführten Arbeit erledigt.

6.5.1 Der „TensorFlow Object Detection Modelmaker"

Die *TensorFlow Object Detection API*, abzukürzen mit *TFOD*, ist ein ebenfalls von Google entwickeltes Framework, welches auf TensorFlow aufsetzt [26]. Es soll das Trainieren von Netzwerkmodellen zur Objekterkennung, bezeichnet als *Detection*-Modelle, auf verhältnismäßig einfachem Weg ermöglichen.

Der Unterschied zwischen normaler Bildklassifizierung und Objekterkennung besteht darin, dass Ersteres keine Information über die Position eines Objektes im Bild liefern kann und nur das gesamte Bild für sich bewertet. Das Zweite kann Objekte innerhalb eines Bildes identifizieren und eine Aussage über dessen Position machen. Mit Objekterkennung ist das Markieren von Objekten mit einfachen viereckigen Boxen, bis hin zu genauem Segmentieren und Trennen des Objekts vom Hintergrund möglich.

Zu Beginn der Arbeit wurde primär zu Machine Learning recherchiert und Erfahrungen gesammelt. Wie in der Einleitung zu Kapitel 4 angesprochen, ist dabei eine Nebenentwicklung entstanden. Bei dem *TensorFlow Object Detection Modelmaker*, oder auch kurz *TFOD Modelmaker*, handelt es sich um eine Anwendung für Windows. Sie verpackt die wichtigsten Funktionalitäten der eben angesprochenen API in eine Anwendung mit grafischer Oberfläche.

Das Ziel des TFOD Modelmaker ist es, das Erstellen von Trainingsdaten sowie das Trainieren und Exportieren eines Detection-Modells noch einfacher und unabhängig von Programmierkenntnissen zu gestalten.

Die Anwendung wurde ursprünglich um bereits existierenden Code herumentwickelt. Die Quelle ist der auf GitHub veröffentlichte, in *Jupyter Notebooks* verpackte Code zu einem TFOD-Tutorial auf dem YouTube-Kanal *Nicholas Renotte* [27]. Das Video ist der Grund, weshalb sich zu Beginn dieser Arbeit überhaupt mit TFOD beschäftigt wurde. Der Großteil des Codes in dem referenzieren GitHub-Verzeichnis hat seinen tatsächlichen Ursprung in dem *TensorFlow 2 Object Detection API tutorial* auf *readthedocs.io* [28]. Er wurde allerdings für diese Anwendung stark modifiziert und ergänzt. Der Code für die grafische Oberfläche, sowie für die Datenvorbereitung wurde selbst entwickelt. Nur kleine Teile, genauer die Zeilen in denen mit TFOD direkt interagiert wird, sind teilweise unverändert übernommen worden.

TFOD hat viele Abhängigkeiten zu anderen Python-Paketen und ist sogar bezüglich Python selbst sehr versionskritisch. Um eine einfache Handhabung zu erreichen, muss also auch die Installation der Anwendung so einfach wie möglich gestaltet sein. Sie besteht letztendlich nur aus dem Herunterladen und Installieren von Python in der Version 3.6, dem Installieren der *C++ Build Tools* von Microsoft und dem Ausführen eines Skripts, welches TFOD und alle notwendigen Python-Pakete in der richtigen Version installiert. Das Installationsskript *pipInstallRequirements.bat*, die Datei *START_PROGRAMM.bat* und das eigentliche Python-Skript der Anwendung *tfodModelMaker.py* ist im Zusatzmedium dieser Arbeit einsehbar. Um die Nutzung der Anwendung zu erleichtern, wurde im Wiki des GitLab-Repository eine Anleitung verfasst. Auch die ist als PDF-Datei im Zusatzmedium unter dem Namen *TFODModelmaker_Anleitung.pdf* einsehbar.

Für die grafische Oberfläche wird die selbst entworfene Bibliothek *guiLibAdvanced.py* verwendet. Sie ermöglicht das einfache und schnelle Erstellen von Anwendungen mit einer GUI. Sie wurde vom Autor bereits vor dieser Arbeit entworfen und kam zuvor schon in Anwendungen zum Einsatz. Da sie nicht im Rahmen dieser Arbeit entstanden ist, wird sie im Folgenden nicht weiter behandelt.

Die GUI ist in vier Tabs unterteilt. Der Erste trägt den Namen *Collect* und behandelt das Aufnehmen von Bildern, die für das Training verwendet werden sollen. Der Zweite heißt *Label*, da hier die zu erkennenden Objekte in den Bildern markiert werden. Im dritten namens *Train* findet das Training des Modells statt. Der letzte Tab mit dem Namen *Test/Export* ist für das Testen und Exportieren des trainierten Modells zuständig.

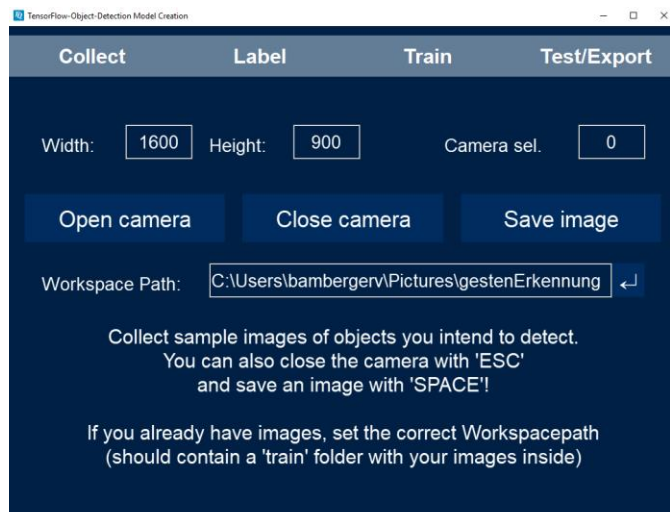


Abbildung 11: TFOD Modelmaker (Collect Tab)

Alle Tabs sind mit Anweisungen und Erklärungen versehen, um die Nutzung so weit wie möglich zu vereinfachen.

Im Collect-Tab wird der Pfad für sowohl das Speichern der Trainingsdaten als auch das Exportieren der trainierten Modelle festgelegt. Außerdem lassen sich hier verbundene Webcams auswählen, starten, deren Kamerafeed anzeigen und Bilder aufnehmen.

Im Label-Tab ist ein Knopf, welcher die Python-Anwendung *LabelIMG* [29] startet. LabelIMG ist ein Werkzeug, mit dem sich Objekte in einem Bild sehr einfach markieren und mit einem Label versehen lassen. Der Zweck ist, Detection-Modellen zeigen zu können, was für Objekte sich wo in den Trainingsbildern befinden. Nach dem Training muss das Modell diese Objekte in neuen Bildern selbst erkennen können.

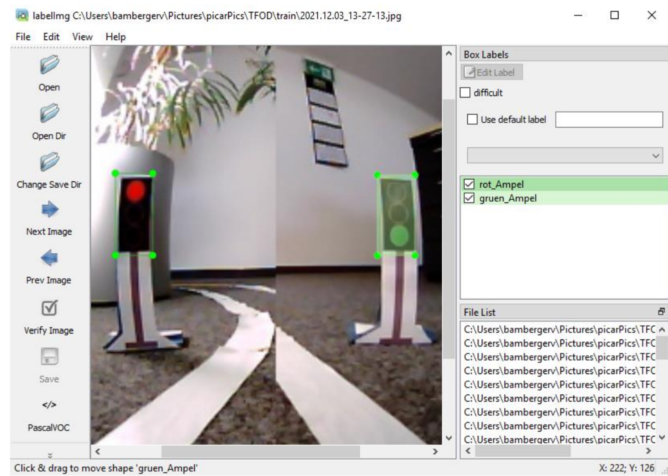


Abbildung 12: LabelIMG (Labeln roter und grüner Ampeln)

Sind alle Trainingsbilder mit einem Label versehen worden, werden per Knopfdruck alle durchsucht und in einer sogenannten *Label Map* abgespeichert. Das neuronale Netzwerk kann die Labelbezeichnungen nicht direkt als Zeichenkette ausgeben, sie müssen der Ausgabe des Netzwerks erst zugeordnet werden.

Im Label-Tab wird auch ein vortrainiertes Modell ausgewählt. Damit wird ein Teil der Vorbereitung für das eigentliche Trainieren abgehandelt. Das vortrainierte Modell ist nichts weiter, als ein Detection-Modell, das bereits mit einer großen Menge an Daten trainiert wurde. Während komplexere Features und die Klassifizierung der neuen Objekte noch antrainiert werden müssen, können die einfacheren Features, wie Kanten und andere geometrische Grundformen, aus dem schon trainierten Modell übernommen werden. So profitiert das eigene Detection-Modell von einer großen Menge Training, das nicht selbst durchgeführt werden musste. Der noch aufzubringende Trainingsaufwand fällt so deutlich geringer aus.

Im Train-Tab lässt sich vor dem Starten des Trainings die Anzahl der Iteration einstellten, die während dem Trainingsvorgang durchgeführt werden sollen. Außerdem kann mit einem Knopfdruck das sogenannte *Tensorboard* gestartet werden. Dieses Tool ist ein Teil von TensorFlow und kann zur Visualisierung von Trainingsabläufen verwendet werden.

Der Test/Export-Tab ermöglicht das Testen des trainierten Modells mit dem Kamerafeed der Webcam, die im Collect-Tab eingestellt wurde. Das soll auch die Validierung ersetzen. Da hier immer nur schon vortrainierte Modelle trainiert und so die Anzahl der Trainingsdurchläufe relativ niedrig gehalten werden, ist das Auftreten von Overfitting sehr unwahrscheinlich. Dem entsprechen zumindest die Erfahrungen, die während dieser Arbeit gesammelt werden konnten.

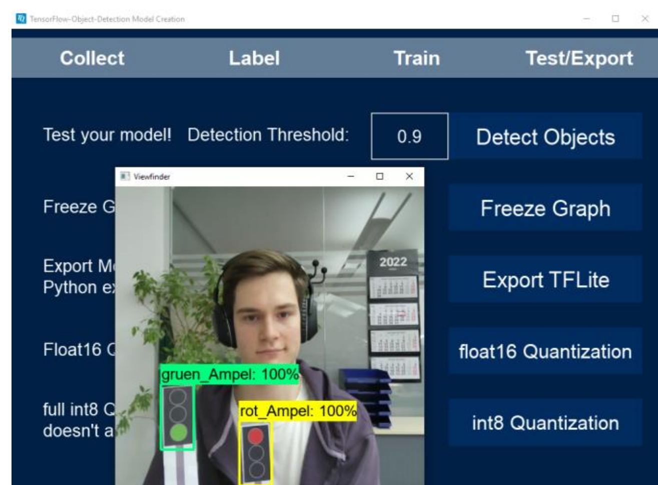


Abbildung 13: TFOD Modelmaker (Testen eines Modells, dahinter der Test/Export-Tab zu sehen)

Im letzten Tab kann die Anwendung das trainierte Modell auf verschiedene Arten exportieren. Die erste Möglichkeit ist ein normales TensorFlow-Modell. Ebenso möglich ist auch der Export in das tflite-Format, entweder als normales Modell mit 32 Bit großen Gewichten als Gleitkommazahlen, quantisiert auf 16 Bit oder quantisiert auf 8 Bit große Integer-Werte.

Bei Export des Modells in ein tflite-Format, wird automatisch eine Datei mit Beispielcode in das selbe Verzeichnis gespeichert. Der Code ist gut kommentiert und soll demonstrieren, wie das exportierte Modell eingesetzt werden könnte.

Die Datei heißt *objDetectTflite_example.py* und ist auch im Zusatzmedium einsehbar. Sie erkennt mit dem exportierten Modell Objekte im Bild einer Webcam, umrandet sie und setzt das entsprechende Label darüber. Damit ähnelt sie rein funktional auch der Testfunktion, die in den Modelmaker eingebaut ist, dargestellt in Abbildung 13 oben.

Der Beispielcode basiert auf *detect_picamera.py*, einem Beispiel für Objekterkennung mit dem Raspberry Pi, das ursprünglich auf dem GitHub von TensorFlow einsehbar war. Die Datei existierte noch, als der TFOD Modelmaker programmiert wurde, jedoch wurde sie am 19. Oktober 2021 entfernt und duch ein überarbeitetes Beispiel ersetzt, siehe die Versionshistorie [30]. Der Code enthält einige Funktionen, die ohne Änderung aus dem dortigen Beispiel übernommen wurden, der größere Teil wurde jedoch modifiziert oder selbst geschrieben.

6.5.2 Testen des Programms „eIQ Portal“ von NXP

Wie zu einem späteren Zeitpunkt während der Arbeit entdeckt wurde, bietet NXP selbst ein Tool mit grafischer Oberfläche, mit dem neuronale Netze sowohl für Bildklassifizierung als auch Detection einfach zu trainieren, validieren und exportieren sein sollen. Es trägt den Namen *eIQ Portal* und ähnelt dem selbst entwickelten TFOD Modelmaker was seine Funktion betrifft.

Die Software ist Teil des *eIQ Toolkit*, laut NXP ein Werkzeugkasten für das Entwickeln von ML-Modellen [31].

Das Tool soll getestet werden, um für die Implementierung von Objekterkennung in das PiCar eine Alternative für das Erstellen von Detection-Modellen zu haben. Die mit eIQ Portal und dem TFOD Modelmaker trainierten tflite-Modelle sollen auch verglichen werden. Das Modell das die besten Ergebnisse liefert, wird in das Fahrzeug implementiert.

6.5.3 Implementieren von Objekterkennung in das PiCar

Hier wird das Kapitel 6.4.2 wieder aufgegriffen. Die bisherige Arbeit mit Objekterkennung soll nun auch für das Fahrzeug nutzbar gemacht werden. Das gesetzte Ziel ist, Ampeln oder Verkehrsschilder zu erkennen und infolgedessen die Geschwindigkeit entsprechend anzupassen.

Da mit dem TFOD Modelmaker und eIQ Portal bereits Tools für das Trainieren von Detection-Modellen zur Verfügung stehen, muss dieser Prozess hier nicht weiter behandelt werden. Idealerweise soll sowohl das Modell für die Lenkung, als auch das Detection-Modell

hardwarebeschleunigt werden. Der Ansatz, der hierfür verfolgt wird, ist die Objekterkennung durch die GPU und das Lenkmodell durch die NPU beschleunigen zu lassen.

Aus dem schon in Kapitel 4 referenzierten User Guide von NXP [6] geht hervor, dass von Hardwarebeschleunigung durch die NPU zu Beschleunigung durch die GPU umgeschaltet werden kann, indem die Umgebungsvariable `USE_GPU_INFERENCE` gesetzt wird.

Um diese Variable für Fahrzeugsteuerung und Objekterkennung getrennt setzen zu können, muss beides in getrennten Prozessen ablaufen, daher bekommt die Objekterkennung eine eigene Datei namens *objDetectTflite.py*. Sie kann im Zusatzmedium eingesehen werden.

Dass das Fahrzeug für seinen Betrieb nun zwei Prozesse benötigt, bringt eigene Schwierigkeiten mit sich, die überwunden werden müssen.

Mit der USB-Kamera kann nur in einem Prozess direkt kommuniziert werden, es benötigen jedoch beide Prozesse die Bildinformationen um zu funktionieren. Es müssen auch beide Prozesse Einfluss auf die Fahrzeugsteuerung und damit die API des PiCar nehmen können.

Beides kann mit Interprozesskommunikation gelöst werden. Mit den Standardbibliotheken von Python lässt sich eine solche Verbindung über sogenannte *Sockets* herstellen. Damit der Nutzer außerdem nicht zwei Python-Skripte von Hand starten muss, startet das eine Skript automatisch das zweite. In der Datei *drivePicar.py* wird die zuvor genannte Umgebungsvariable gesetzt und die zweite Datei *objDetectTflite.py* ausgeführt, indem aus dem Skript heraus Befehle auf der Linux-Shell ausgeführt werden.

Die Prozesse sind nebenläufig, müssen aber durch die bereits erwähnte Interprozesskommunikation synchronisiert werden. Diese Kommunikation folgt einem festen Ablauf.

Der Prozess welcher das Skript *drivePiCar.py* ausführt, sendet die aktuellste Bildinformation der Kamera an den zweiten Prozess. Dieser führt die Objekterkennung durch und schickt die relevanten Ergebnisse zurück an den ersten Prozess. Nach Erhalt der Ergebnisse wird sofort das nächste Bild gesendet und im Anschluss, entsprechend der vorliegenden Ergebnisse aus dem letzten Bild, die Geschwindigkeit des Fahrzeugs verändert.

Wie zu Beginn des Kapitels angesprochen, sollen Ampeln und Verkehrsschilder erkannt werden. Für diesen Zweck wurden auch Spielzeugampeln und Schilder angeschafft. Es stellte sich jedoch heraus, dass die Fähigkeit zur Erkennung dieser Objekte durch die niedrige Qualität der verwendeten Kamera stark eingeschränkt ist. Geschwindigkeitsschilder konnten nicht zuverlässig unterschieden werden. Auch die verwendete Ampel war zu klein, um zuverlässig und rechtzeitig erkannt zu werden. Daher wurden eine grüne und eine rote Ampelgrafik erstellt, groß ausgedruckt und auf Pappkarton aufgeklebt. Die Ampeln waren bereits in den Abbildung 12 und Abbildung 13 zu dem TFOD Modelmaker auf Seite 30 zu sehen.

Bei roter Ampel reduziert das Fahrzeug die Geschwindigkeit auf null. Sobald die grüne Ampel erkannt wird oder die rote für mindestens 2 Sekunden verschwindet, fährt das Fahrzeug wieder weiter.

Die Datei *objDetectTflite.py* ist von dem Beispiel *objDetectTflite_example.py*, das durch den TFOD Modelmaker erzeugt wird, abgeleitet. Hinzugefügt wurde neben der Fähigkeit zur Kommunikation mit einem weiteren Prozess, auch eine Funktion für das Entfernen der Bildbereiche, die für die Erkennung uninteressant sind.

Ampeln stehen immer an einem der beiden Fahrbahnränder, jedoch nie in der Mitte. Daher schneidet die Funktion jeweils auf der linken und rechten Hälfte des Weitwinkelbildes einen rechteckigen Teil heraus und setzt beide Teile zu einem 320x320 Pixel großen Bild zusammen. Das enspricht der Dimension des Eingangstensors des trainierten Detection-Modells.

Gesammelt wurden die Bilder für den Trainingsvorgang mit Hilfe eines weiteren Python-Skripts. Es trägt im Zusatzmedium den Namen *collectTrainImagesTFOD.py*. Hierfür musste erneut der Raspberry Pi mit der am Fahrzeug befestigten Kamera verbunden werden. Da das Fahrzeug dabei nicht fahren können muss, ist es nicht notwendig, das komplette Fahreug umzubauen. Die Ampeln wurden einfach an verschiedenen Positionen vor dem Fahrzeug aufgestellt. Durch Remotedesktopzugriff auf den Raspberry konnten die Bilder leicht aufgenommen werden. Das oben genannte Skript ist so programmiert, dass bei betätigen der Leertaste ein Bild gespeichert wird. Die Trainingbilder werden vor dem Abspeichern mit der selben Funktion vorbereitet, die im Absatz zuvor erklärt wurde.

Im letzten Schritt wurden die Bilder in den TFOD Modelmaker und in eIQ Portal importiert, um wie in Kapiteln 6.5.1 bis 6.5.2 beschrieben, ein Detection-Modell zu trainieren. Es wurde als tflite-Modell exportiert und auf dem Fahrzeug eingesetzt.

6.5.4 Video für Anhang an strategische Präsentation

TQ Embedded hat an einem Wettbewerb teilgenommen, um in das *Early Access Programm* für die kommende *i.MX 9* Generation an Prozessoren von NXP zu gelangen. Der Begriff Early Access bedeutet frühzeitigen Zugang, in diesem Fall zu der neuesten, noch nicht frei auf dem Markt erhältlichen Hardware von NXP.

Der Eintrag von TQ in diesem Wettbewerb bestand aus einer Präsentation mit einem angehängten Video. Im Video sollte zu sehen sein, wie das im Rahmen dieser Arbeit entwickelte Fahrzeug über die Demostecke fährt. Es sollte auch ein Text, der dem Zuschauer ein wenig über das Projekt erzählt, in englischer Sprache verfasst und schließlich auf das Video gesprochen werden.

Das Ziel hierbei war selbstverständlich den Wettbewerb zu gewinnen. Die Aufgabenstellung bestand darin, das Video nicht mit technischen Details zu überladen, sondern zu zeigen, dass sich TQ-Systems mit Machine Learning beschäftigt und welche Erfolge hierbei erzielt wurden. Es sollte auch ein gewisses Maß an Begeisterung über die Leistungsfähigkeit des Prozessors von NXP geäußert werden.

Das Video wurde mit der Handykamera aufgenommen, der zuvor verfasste Text eingesprochen und anschließend alles zusammengeführt, geschnitten und produziert.

Das fertige Video trägt im Zusatzmedium den Namen *TensorflowCar.mp4*.

7 Ergebnisse

Die zu Beginn festgelegten Projektziele wurden alle erreicht. Die Erwartungen an die Fähigkeit des Modellautos auf der festgelegten Spur zu bleiben, wurden weit übertroffen. Im Folgenden werden die Erfolge und Misserfolge der Tätigkeiten behandelt, die im Rahmen dieser Arbeit durchgeführt und bereits in vorherigen Kapiteln beschrieben worden sind.

7.1 Performance des Fahrzeugs auf der Strecke

7.1.1 Genauigkeit des trainierten Netzwerks

In der finalen Version ist das Fahrzeug in nahezu allen Situationen in der Lage, zuverlässig auf der Strecke zu bleiben, an der das Netzwerk trainiert wurde. Wird das Fahrzeug neben die Fahrbahn gesetzt, so dass sie aber noch im Kamerabild zu sehen ist, findet es von selbst wieder zurück in die Spur und fährt normal weiter. Es bleibt auch eher gut in der Mitte der Spur, Abschweifen in begrenztem Maß lässt sich jedoch nicht vollständig verhindern, da das Modellauto ein gewisses Lenkspiel aufweist.

Um die Anwendbarkeit des trainierten Modells auf anderen Strecken zu prüfen, wurden weitere Teststrecken aufgebaut. Eine Strecke wurde in Bezug auf die Gleichmäßigkeit der Bahnbreite nachlässiger gestaltet als die Trainingsstrecke, sonst wurde nach wie vor weißes Klebeband auf dunkelgrauem Teppich verwendet. Die zweite Strecke wurde auf einem Parkettboden aufgebaut. Verwendet wurde hauptsächlich wieder dasselbe Klebeband, für einen kleinen Teil wurde allerdings dunkelbraunes Paketband benutzt, siehe Abbildung 14 rechts.

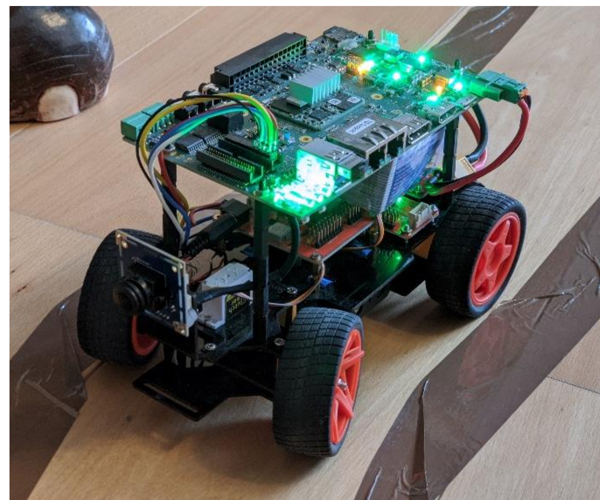


Abbildung 14: PiCar auf Parkettboden mit brauner Fahrbahnmarkierung

Die erste Teststrecke war für das Fahrzeug ebenso einfach zu bewältigen wie die Trainingsstrecke. Erst die zweite Strecke auf Parkettboden lieferte gemischte Ergebnisse.

Die andere Farbe des Untergrunds scheint keinen Einfluss auf die Zuverlässigkeit der Lenkwinkelschätzung zu haben. Auch auf dem Abschnitt mit dunkelbraunem Klebeband auf Parkett fuhr das Fahrzeug beinahe genauso gut, wie auf den Abschnitten mit weißen Markierungen. Das lässt darauf schließen, dass die Verwendung von Negativbildern ihren Zweck erfüllt hat.

Probleme haben alleine die Muster auf dem Parkettboden verursacht, wie zum Beispiel Astlöcher und die Kanten der einzelnen Bretter. In manchen Fällen entschied sich das Fahrzeug, entlang eines Stücks Parkett zu fahren, statt auf der Strecke zu bleiben. Dabei schien es nicht relevant zu sein, ob die eigentliche Fahrbahnmarkierung weiß oder

dunkelbraun ist. Da für das Training farblich einheitlicher Teppichboden genutzt wurde, kann das Modell nun nicht gut unterscheiden, was auf einem Parkettboden wirklich eine Fahrbahnmarkierung ist und was nicht.

Ähnliche Symptome treten auf, sobald ein harter Schatten über die Strecke geworfen wird. Scheint die Sonne durch ein Fenster auf die Strecke, folgt das Fahrzeug der Linie des Schattens statt der Spur.

Diese Probleme könnten behoben werden, indem die Trainingsdaten durch Beispiele mit unterschiedlichen Untergründen und Schatten auf der Fahrbahn erweitert würden. Der durch die Schatten verursachte Helligkeitsunterschied übersteigt aber möglicherweise auch den Dynamikumfang der verwendeten Kamera, wodurch sie nur noch eingeschränkt brauchbare Informationen liefern würde.

7.1.2 Leistung des Netzwerks auf unterschiedlicher Hardware

Die von Nvidia vorgeschlagene Netzarchitektur liefert nicht nur auf der Strecke sehr gute Ergebnisse, sondern ist auch in Bezug auf die benötigte Rechenleistung sehr effizient. Auf dem Raspberry Pi 4, auf dem die trainierten Modelle Anfangs getestet wurden, steht keine Hardwarebeschleunigung für neuronale Netze zur Verfügung. Dennoch braucht das Modell auf der CPU des Raspberry nur zwischen 11 und 16 Millisekunden. Da die Kamera nur 30 Bilder in der Sekunde liefert, also nur alle 33 Millisekunden ein neues Bild zur Verfügung steht, ist das ausreichend.

Eine kürzere Berechnungsdauer hat trotzdem Vorteile, denn je länger die Berechnung dauert, desto größer ist die Verzögerung, mit der das Fahrzeug auf eine neue Situation reagiert. Eine passende Analogie wäre hier die Totzeit in einem Regelsystem. Sie sollte mit der Hardwarebeschleunigung auf dem TQ-Modul reduziert werden.

Wie bereits erwähnt, standen die Hardwarebeschleuniger nicht von Anfang an zur Verfügung, da sie erst mit einem Kernelpatch aktiviert werden mussten. Tatsächlich ist die Performance des Modells auch auf der CPU des TQ-Moduls schon etwas schneller als auf der des Raspberry Pi 4. Die benötigte Zeit beträgt hier nur noch 9 bis 12 Millisekunden.

Der im TQ-Modul verwendete NXP i.MX 8M Plus Prozessor implementiert vier *Cortex A53* Kerne, der Raspberry Pi 4 aber vier *Cortex A72*, welche eigentlich mehr Leistung bringen sollten. Dass der i.MX 8M Plus hier dennoch schneller ist, sorgte für Überraschung. Für diesen Umstand gibt es zwei mögliche Erklärungen.

Die erste fußt auf der Tatsache, dass auf dem TQ-Modul ein minimalistisches und hoch performantes Betriebssystem läuft. Auf dem Raspberry Pi hingegen läuft die dort gängige Distribution *Raspbian* mit grafischer Oberfläche und Desktopumgebung.

Die wahrscheinlichere Ursache für diese Beobachtung ist jedoch, dass der Cortex A53 bei einer der mathematischen Grundoperationen für die Berechnung von neuronalen Netzen schneller ist als der Cortex A72. Diese Information entstammt einem Beitrag zu dem weit verbreiteten Softwarepaket *Mathematica* [32]. Dort wird dessen Leistung auf dem Raspberry Pi 4 und dem Raspberry Pi 3B, welcher ebenfalls einen Cortex A53 implementiert, miteinander verglichen. Den dort durchgeführten Tests nach ist der A72 bei meisten

Operationen um etwa den Faktor 2 schneller als der A53, bei Matrixmultiplikationen aber um den Faktor 0.8 langsamer. Das deckt sich ausreichend genau mit dem Faktor von ungefähr 0.81, der bei dem Vergleich der schnellsten Berechnungszeiten des ML-Modells auf der CPU des Raspberry und der des TQ-Moduls errechnet wurde.

Sobald die Hardwarebeschleuniger zur Verfügung standen, wurde Beschleunigung durch GPU sowie NPU getestet. Allein die GPU senkte die benötigte Zeit schon auf 1.6 Millisekunden, die NPU konnte die Zeit noch weiter auf etwa 0.9 Millisekunden senken. Durch die schnellere Reaktionszeit bei Nutzung der NPU hat sich die Performance auf der Strecke weiter verbessert, zumindest dem subjektiven Eindruck nach.

7.2 Erkenntnisse durch das Training des Netzwerkmodells

Das Netzwerkmodell wurde nicht nur einmal trainiert und auf dem Fahrzeug getestet. Trainiert wurden mehrere Versionen mit unterschiedlichen Trainingsparametern. Variiert wurden beispielsweise die Anzahl der Epochen, der Betrag der Lenkeinschlagskorrektur für die Bilder der äußeren Kameras, der Parameter der Dropout-Schicht und die Batchgröße. Tricks für die Erzeugung neuer Datenpunkte, wie das bereits erklärte Spiegeln der Trainingsbilder, wurden ebenfalls noch nicht bei der ersten Version des trainierten Modells eingesetzt.

Die erste Version des Modells wurde nur mit dem Raspberry Pi getestet. Die Erwartungshaltung war, dass das Fahrzeug weitestgehend auf der Strecke bleiben kann. Es wurde jedoch damit gerechnet, dass es nach kurzer Zeit von der Stecke abkommen könnte, da es nur mit einem kleinen Datensatz trainiert wurde. Tatsächlich ist schon das erste Modell in allen Situationen zuverlässig auf der Strecke geblieben. Auch das Zurückfahren in die Spur funktionierte schon mit der ersten Version des trainierten Modells. Der einzige Mangel, der erkannt werden konnte, war ein Linksdrall oder auch Bias nach Links, welcher durch nachträgliches Justieren mit einem Offset in der Software allerdings vollständig eliminiert werden konnte.

Die unerwartet gute Leistung des ersten Modells war der Grund, weshalb kaum mehr Trainingsdaten gesammelt wurden. Erst nach dem Umbau von drei zu nur einer Kamera, welcher die Steuerung aufgrund einer leicht geänderten Kameraposition etwas verschlechterte, wurden mit der einzelnen Kamera weitere Bilder gesammelt, wie bereits in Kapitel 6.2.2 erklärt.

Mit weiteren Versionen wurde vor allem getestet, welche Auswirkungen die Veränderungen von Parametern haben. Im Folgenden wird sich stellenweise auf spezielle Vorgehensweisen und Methoden bezogen, die in Kapitel 6.3 bereits ausführlich erklärt wurden.

Für die Fehlerkorrektur wurden unterschiedliche Korrekturwinkel von 10 bis 20 Grad getestet. Wird die Lenkwinkelkorrektur zu groß gewählt, äußert sich das auf der Strecke durch starkes Überreagieren. Wird das Fahrzeug neben die Strecke gesetzt, verursacht das mit 20 Grad Korrekturwinkel trainierte Modell vehementes Zucken der Lenkung, während zurück in die Spur gefahren wird. Im Vergleich dazu steuert ein mit 10 Grad Korrekturwinkel trainiertes Modell das Fahrzeug immer noch zügig, aber geschmeidiger zurück in die Spur. Daher wurde das finale Modell auf einen Korrekturwinkel von 10 Grad trainiert.

In einem Fall wurden auch die Datenpunkte der äußeren Kameras komplett weggelassen und das Modell nur mit Daten der mittleren Kamera trainiert. So sollte die Effektivität der Fehlerkorrekturmethode überprüft werden. Ohne die Daten der äußeren Kameras ist das Modell weiterhin in der Lage das Fahrzeug auf der Strecke zu halten. Es tendiert so jedoch sichtbar stärker dazu in Schlangenlinien zu fahren, und fährt mal deutlich zu weit Links und mal zu weit Rechts. Auch das Zurückfahren in die Spur, sobald sich das Fahrzeug neben der Spur befindet, funktioniert ohne diese Information nicht mehr zuverlässig.

Hier kann zurück verwiesen werden auf das DeepPiCar im letzten Absatz von Kapitel 5.2.1, denn auch das wurde nur mit den Daten einer Kamera trainiert. Der Autor des dort referenzierten Artikels schlussfolgert, das Abdriften des DeepPiCar von der Strecke läge an zu wenig Daten. Dem muss hier widersprochen werden, denn die hier gesammelten Datenmengen nur der mittleren Kamera sind etwa fünfundzwanzigmal so groß, wie die Daten mit denen das DeepPiCar trainiert wurde. Das Fahrzeug zeigt trotzdem dasselbe Verhalten, sobald es ohne die Daten der äußeren Kameras trainiert worden ist.

Was die Batchgröße angeht, konnte kein Zusammenhang mit der Qualität des trainierten Modells festgestellt werden. Wenn jedoch während dem Trainingsvorgang die Auslastung der CPU deutlich unter 100% bleibt, kann durch eine Erhöhung der Batchgröße die Auslastung maximiert und so der Trainingsvorgang beschleunigt werden.

Das systematische Vorgehen beim Spiegeln der Trainingsdaten hat tatsächlich den Bias des Modells, nach links zu lenken, reduziert. Um ihn weiter zu reduzieren, war mehr Training und so eine höhere Zahl von Trainingsepochen nötig. Das Augmentieren der Daten mit Negativbildern hat die Anzahl der nötigen Trainingsepochen, um zu einem zufriedenstellenden Ergebnis zu gelangen, ebenfalls erhöht.

Zu Beginn wurde mit nur 10 oder 20 Epochen trainiert, was zu einer Abweichung von bis zu 6 Grad führt. Um die Zahl der Epochen nach oben setzten zu können, ohne Probleme mit Overfitting zu bekommen, wurde für das finale Modell der Checkpoint Callback erzeugt und die Anzahl der Trainingsepochen auf 50 gesetzt. Trotz der hohen Zahl an Epochen konnte jedoch kein nennenswerter Overfitting-Effekt erfasst werden. Die Gewichte, die aus dem Checkpoint letztendlich geladen und in das tflite-Modell exportiert wurden, stammen aus der Trainingsepoche 48. Das geringe Overfitting ist auch ein Grund, weshalb der im Modell eingebaute Dropout nur 5% beträgt. Der Fehler durch das letzte trainierte Modell beträgt nur noch 1 bis 2 Grad.

7.3 TFOD Modelmaker und Objekterkennung

Der TFOD Modelmaker, beziehungsweise die TensorFlow Object Detection API in der verwendeten Version allgemein, ist nicht komplett fehlerfrei. Sie wird dem Anschein nach nicht so sorgfältig gepflegt, wie das bei dem Basisframework TensorFlow der Fall ist. Das ist auch an mehreren Warnungen, verursacht durch die Verwendung von Methoden, die in zukünftigen TensorFlow-Versionen entfernt werden sollen, zu erkennen. Ein Problem diesbezüglich konnte im Rahmen der Arbeit nicht gelöst werden. Nach dem Quantisieren funktionieren die exportierten tflite-Modelle nicht wie erwartet. Es können also nur normale Modelle mit Gewichten in 32 Bit großen Gleitkommazahlen exportiert werden. Die können jedoch nicht durch die NPU des TQ-Moduls beschleunigt werden.

Die ausgedruckten Ampeln konnten von dem System gut genug erkannt werden. Da das Ausführen des Models ohne Beschleunigung jeweils knapp eine Sekunde gebraucht hat, musste anfangs eine niedrigere Geschwindigkeit eingestellt werden.

Obwohl es sich um ein nicht quantisiertes Modell handelt, ist zumindest eine teilweise Beschleunigung mit der GPU möglich. Ein Test zeigt eine Reduktion der Zeit von einer Sekunde auf 0.6 Sekunden. Das wichtigere Ergebnis ist jedoch, dass es tatsächlich möglich ist, in zwei Prozessen zwei verschiedene ML-Modelle zu beschleunigen. Eines läuft auf der NPU und das andere auf der GPU, ohne dass eines das andere blockiert, oder Performanceeinbußen beim anderen verursacht.

Falls das Quantisierungsproblem des TFOD Modelmaker behoben werden kann, oder auf andere Weise ein quantisiertes Detection-Modell erstellt werden kann, ist es also möglich, zeitgleich die NPU und GPU Beschleunigung zu nutzen.

Vorerst kann das 32 Bit Modell durch die Nutzung von optimierten Instruktionen bei Berechnung mit der CPU noch schneller werden, als das mit der GPU der der Fall ist. Dafür kann auf dem verwendeten System das sogenanntes *XNNPACK Delegate*, welches Operationen mit Gleitkommazahlen auf der CPU optimiert, verwendet werden.

Die Zeit für eine Berechnung mit dem XNNPACK sinkt auf 0.3 Sekunden, was aktuell die schnellst mögliche Zeit für das Detection-Modell ist. Werden die CPU-Ressourcen jedoch anderweitig gebraucht, kann eine Berechnung, die durch die GPU gestützt wird, möglicherweise dennoch sinnvoller sein.

Das Fahrzeug in seiner finalen Version bleibt nun also sehr zuverlässig und mittig auf der Straße und ist in der Lage an der roten Ampel anzuhalten, sowie bei Grün loszufahren. Bei einer Berechnungszeit von 0.3 Sekunden ist das Fahrzeug immerhin in der Lage, etwas mehr als 3 Bilder in der Sekunde zu bearbeiten. Diese Berechnungszeit genügt, um das Fahrzeug mit 50 Prozent seiner Maximalgeschwindigkeit fahren zu lassen.

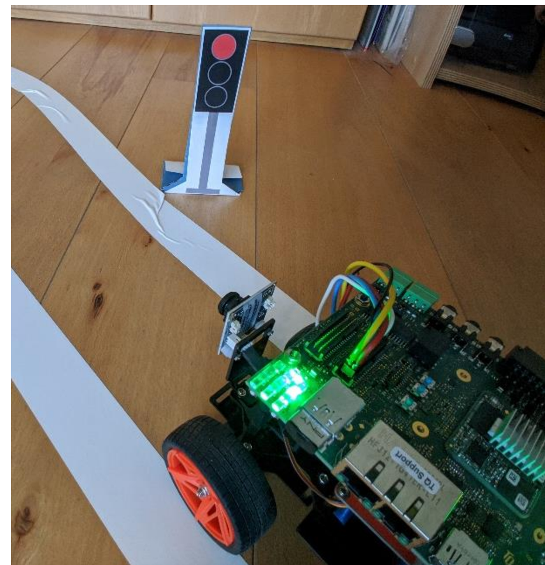


Abbildung 15: Das PiCar, nachdem es an der Ampel gehalten hat.

Der einzige wirkliche Schwachpunkt aktuell, ist eine regelmäßige Fehlerkennung der roten Ampel, die dazu führt, dass das Fahrzeug hin und wieder unnötig für 2 Sekunden stehen bleiben muss. Um das zu beheben, müsste das Modell mit mehr Bildern trainierte werden, insbesondere vor unterschiedlichen Hintergründen.

7.4 Ergebnisse der Evaluierung von *eIQ Portal*

Das Tool von NXP integriert alle für das Erstellen von Detection-Modellen übliche Schritte. Dazu gehören das Markieren und Labeln von Objekten in den Trainingsbildern, das Aufteilen

in Daten für Training und Validierung, Einstellungen für Bildaugmentationen, wählen einer Netzarchitektur für drei unterschiedliche Anforderungen von Geschwindigkeit bis Genauigkeit, das Training, die Validierung und das Exportieren des Modells.

Was die Funktionalität und Einstellungsmöglichkeiten angeht, ist das Programm um einiges umfangreicher als der selbst programmierte TFOD Modelmaker. Der Versuch, die exportierten Modelle einzusetzen, verursachte jedoch unerwartete Schwierigkeiten.

Es wurde mehrfach ein Detection-Modell für die Erkennung von Ampeln trainiert und mit verschiedenen Einstellungen exportiert.

Die ersten Schwierigkeiten offenbaren sich schon bei der Interpretation des Ausgangstensors der exportierten tflite-Modelle. Dieser hat, wenn das Modell auf 2 Klassen trainiert wurde, eine Dimension von 2034x7. Die Bedeutung dieses Tensors ist tatsächlich nirgendwo offiziell dokumentiert.

Durch Recherche in Foren der Community von NXP und durch Interpretationsarbeit an C-Code von NXP konnte zumindest eine sehr gute Vermutung erarbeitet werden. Das Modell macht, ersichtlich an der ersten Dimension, wohl immer 2034 verschiedene Erkennungen. Die zweite Dimension setzt sich dann aus mehreren unterschiedlichen Informationen zusammen. Die ersten drei Werte entsprechen vermutlich den Wahrscheinlichkeiten dafür, dass sich an einer Position ein Objekt entweder der ersten Klasse, der zweiten Klasse oder einfach nur der Hintergrund befindet. Die letzten vier Einträge enthalten wahrscheinlich Informationen über die Position und Dimension des erkannten Objektes im Bild. Eine Bestätigung hierfür konnte, durch Experimente anhand der Wahrscheinlichkeiten für die zwei Klassen, zwar ansatzweise erahnt werden, jedoch konnte aus den letzten vier Werten nie eine Markierung an der richtigen Position des Bildes erzeugt werden.

Darüber hinaus wurde eine weitere Abnormität im Programm festgestellt. Der Verlauf für die Genauigkeit des Modells während dem Training kann nie einen Wert von 66.7 Prozent überschreiten. Er wird bei exakt diesem Wert abgeschnitten, siehe Abbildung 16 rechts.

Abbildung 16: Ausschnitt aus dem GUI von eIQ Portal

In der Hoffnung die Probleme beheben zu können, wurde zu einem späteren Zeitpunkt eine aktuellere Version von *eIQ Toolkit* installiert. Diese Version liefert jedoch schon bei der Validierung nach dem Training innerhalb des Programms keine sinnvollen Ergebnisse mehr.

Anscheinend besteht ein Problem entweder mit dem Programm selbst oder mit dem System auf dem das Programm ausgeführt wurde, eine Kombination aus beidem kann natürlich nicht ausgeschlossen werden. Möglicherweise wurde auch nur aufgrund von mangelnder Dokumentation zur Nutzung exportierter tflite-Modelle die Sache komplett falsch angegangen. Da jedoch für weitere Experimente die Zeit fehlte, sind alle weiteren Tests mit eIQ Portal eingestellt und das Unterfangen als Misserfolg abgestempelt worden.

7.5 Einsatz des selbst erstellten Videos über das Fahrzeug

Das Video entstand zu einem Zeitpunkt, bei dem die Hardwarebeschleuniger auf dem TQ-Modul noch nicht verwendet werden konnten. Auch ohne sie fährt das Fahrzeug bereits sehr gut und bleibt nur in seltenen Fällen erst hinter der roten Ampel stehen, weil die Verarbeitungszeit der Objekterkennung so lange dauert.

Da zum Zeitpunkt der Aufnahme noch keine Erkenntnisse zur den Beschleunigern gewonnen werden konnten, erwähnt das Video auch keine Details zu dessen Leistungsfähigkeit. Da aber bereits erkannt worden war, dass ML-Modelle auf der CPU des TQ-Moduls messbar schneller berechnet werden als auf der des Raspberry Pi, wurde stattdessen auf diese Tatsache hingewiesen.

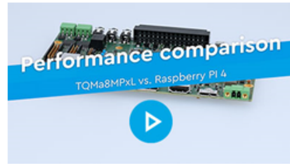
Der Wettbewerb für das Early Access Programm, bei dem sich die Firma beworben hat, konnte letztendlich gewonnen werden. Die Firma TQ-Systems hat dadurch frühen Zugang zu der neuen *i.MX 9* Prozessorgeneration von NXP bekommen. So können Module schon entwickelt werden, bevor die dafür verwendeten Prozessoren offiziell auf dem Markt erscheinen.

Das Video war nur der Anhang einer Präsentation gewesen, dennoch kann behauptet werden, dass es seinen Beitrag zu diesem Erfolg geleistet hat.

Nachdem die Hardwarebeschleuniger funktionierten und Ergebnisse zu deren Leistung vorlagen, sollten diese Ergebnisse mit einer kurzen Beschreibung des Projekts für einen Newsletter zusammengeschrieben werden. Der Text wurde vom Autor dieser Arbeit entworfen und vom Marketing nur stellenweise abgeändert. Das Video wurde qualitativ als gut genug bewertet, um im Newsletter verlinkt zu werden, obwohl es ursprünglich nur für die strategische Präsentation entworfen wurde.

Ein Ausschnitt des Newsletters ist in der folgenden Abbildung zusehen.

TQMa8MPxL sprintet Raspberry Pi 4 davon



TQ-Embedded hat auf dem TQMa8MPxL-Modul, das auf NXPs i.MX 8M Plus basiert, eine Machine Learning Applikation für einen Vergleichstest realisiert - mit sehr beeindruckenden Ergebnissen: Obwohl der Raspberry Pi 4 einen Quad Cortex A72 nutzt, überzeugt TQMa8MPxL mit seinem Quad Cortex A53 Core durch seine Effizienz und eine bessere Performance sowohl mit als auch ohne NPU-Beschleunigung.

Ein Lenkassistent für Level 2 autonomes Fahren wurde auf dem TQMa8MPxL und dem Raspberry Pi 4 ausgeführt. Beide Geräte nutzen die TensorFlow Lite Runtime unter Python. Während der Raspberry Pi 4 für das 8-Bit-Integer quantisierte Modell eine Zeit von 11 ms bis 16 ms benötigt, schafft der i.MX 8M Plus auf dem TQ-Modul es bereits in 9 ms bis 12 ms - unter Nutzung der Neural Processing Unit (NPU) sind es sogar nur 0,9 ms, also eine mehr als zehnfache Beschleunigung.

Jetzt Video zum Performance-Vergleich ansehen

Abbildung 17: Ausschnitt aus dem Newsletter von TQ-Embedded vom 17.03.22

Das Video wurde von der Marketing Abteilung leicht überarbeitet. Die Abschnitte am Anfang, die das Modul und Starterkit zeigen, waren in der ursprünglichen Version unbewegte Produktbilder, das Video enthielt auch keine TQ-Logos und Hintergrundmusik. Ansonsten entspricht das Video dem, wie ursprünglich vom Autor erstellt. Es wurde als nicht gelistet auf dem YouTube-Kanal der TQ-Group veröffentlicht [33].

Im Video wird eine annähernd doppelte Leistungssteigerung gegenüber dem Raspberry Pi erwähnt. Damit nimmt es, stark übertrieben, Bezug auf den Unterschied der Leistung bei Berechnung mit der CPU. Der Unterschied ist mit der anfangs auf dem TQ-Modul benutzten BSP-Revision tatsächlich etwa eine Millisekunde größer gewesen. Das Angebot, einen neu formulierten Text auf das Video zu sprechen und dabei auch die Leistung der Beschleuniger zu erwähnen, wurde jedoch dankend abgelehnt, da es genügt, dass die genauen Daten im Newsletter zu sehen sind.

8 Fazit und Ausblick

Zu Beginn sorgten die Schwierigkeiten bei der Inbetriebnahme der NPU für ein wenig Unsicherheit, da diese Komponente eines der Hauptverkaufsargumente des Moduls und der Anlass für die entwickelte Demosoftware ist. Die Tatsache, dass das Selbstfahren von Anfang an nahezu perfekt funktioniert hat, hat diese Unsicherheit größtenteils wieder aufgelöst.

Der hier verwendete End-to-End-Ansatz ist einfach und effektiv, im Kontext von autonomem Fahren allerdings in seinen Möglichkeiten begrenzt, da lediglich das Halten der Spur bewerkstelligt werden kann. Wenn die Spur gewechselt, abgebogen oder zum Beispiel ein Weg durch oder vorbei an Hindernissen gefunden werden muss, müsste das Fahrzeug auf einen *Step-by-Step*-Ansatz zurückgreifen. Hierbei müssen im ersten Schritt die sogenannten *Roadfeatures* wie Fahrbahnmarkierungen, die Straße und weitere Dinge erst explizit erkannt werden. Diese Informationen würden dann in weiteren Schritten in einer eigens programmierten Steuerung verwendet werden, um das Fahrzeug zu lenken. Bei diesem Ansatz können zwischen der Roadfeature-Erkennung und der Steuerung Signale, von beispielsweise einem Navigationssystem, hinzugegeben werden, um die richtige Entscheidung, zum Beispiel an einer Weggabelung, treffen zu können.

Auch wenn die hier gewählte Vorgehensweise ihre Aufgabe sehr effektiv bewältigt, ist ein Step-by-Step-Ansatz entschieden mächtiger, er erfordert jedoch auch deutlich mehr Aufwand. Alleine schon das Trainieren eines Modells auf die Erkennung der Roadfeatures ist aufwändiger, da hier in allen verwendeten Trainingsbildern zuerst die zu erkennenden Eigenschaften manuell markiert werden müssten. Damit wäre eine Herausforderung, einen solchen Ansatz in das PiCar zu integrieren, als Thema für eine nachfolgende Arbeit allerdings durchaus denkbar.

Die in Kapitel 2 angesprochene Technologiemesse Embedded World, welche ursprünglich im Zeitraum dieser Arbeit hätte stattfinden sollen, wurde auf den Juni 2022 verschoben. Es ist mittlerweile klar, dass der auf der Messe reservierte Platz leider nicht ausreicht, um das Fahrzeug fahren zu lassen. Stattdessen sollen zusammen mit dem Marketing bessere Aufnahmen von dem Fahrzeug gemacht und ein neues Video erstellt werden, das an einem Messestand zu sehen sein wird. Die Aufnahmen für das bereits existierende Video wurden nur mit einer Handykamera gemacht.

Zusätzlich soll noch eine stationäre ML-Demo für das TQMa8MPxL erstellt werden. All das fällt nun allerdings nicht mehr in den zeitlichen Rahmen dieser Arbeit.

Das zusätzliche Vorhaben, bezogen auf die Objekterkennung, ist am Ende nur bedingt erfolgreich gewesen. Das im Verlauf der Arbeit gewonnene Verständnis und die neue Begeisterung für Machine Learning wird allerdings zwangsläufig dazu führen, dass sich mit diesem Thema weiter auseinandergesetzt wird. Über kurz oder lang wird der nebenbei entwickelte TFOD Modelmaker wahrscheinlich alle seine momentan vorgesehenen Funktionen fehlerfrei anwenden können.

Glossar

API	„Eine **API** (Application Programming Interface) ist ein Satz von Befehlen, Funktionen, Protokollen und Objekten, die Programmierer verwenden können, um eine Software zu erstellen oder mit einem externen System zu interagieren.“ [34]
ARM® Architektur	Von „Advanced RISC Machines“ (kurz: ARM®) lizenzierte RISC Architektur (Reduced Instruction Set Computer) [35].
Array	Ein **Array** ist eine Datenstruktur. Diese Arbeit meint mit **Array** in allen Fällen den aus der Python-Bibliothek *NumPy* importierten Datentyp `ndarray` [36].
batch	Ein Set aus Beispielen die einem Netzwerk während des Trainingsvorgangs in einer Iteration gezeigt werden [9].
Bias	Ein Systematischer Fehler, oder Abweichung vom Ausgang [9].
Build-System	„Als Build bezeichnet die Softwareentwicklung einerseits den gesamten Prozess der Erzeugung einer kompletten, eigenständig lauffähigen Software und andererseits das Ergebnis dieses Prozesses: das oder die ausführbaren Programme einschließlich aller eventuell benötigten Ressourcen“ [37]
CNN	Kurz für **Convolutional Neural Network**
Computer Vision	Ähnlich **Vision-Anwendung**, bzw. Bezeichnung für dessen Einsatz.
convolution	Englisch für Faltung. Erklärung siehe Kapitel 4.2.
convolutional layer	Eine faltende Schicht in einem **CNN.** Erklärung siehe Kapitel 4.2.
Convolutional Neural Network	Ein faltendes neuronales Netzwerk. Erklärung siehe Kapitel 4.2.
CPU	**C**entral **P**rocessing **U**nit (Prozessor)
data augmentation	Transformieren existierender Beispiele, um neue Beispiele zu generieren [9].
Deep Learning	Fachsprache für tiefes Lernen, eine Unterkategorie von maschinellem Lernen, bei dem neuronale Netze mit mehr als eine versteckten Schicht mit Neuronen einsetzt

	werden, und große Mengen unstrukturierter Daten verarbeitet werden können [12].
dense layer oder **densely connected layer**	Vollständig verbundene Schicht, bedeutet jedes Neuron der Schicht ist mit jedem Neuron der benachbarten Schichten verbunden [9].
Dropout	Ein Dropout kann zwischen die Schichten eines neuronalen Netzwerks gesetzt werden. Es bewirkt dort das willkürliche zu null setzen von einigen Ausgängen der vorhergehenden Schicht während des Trainingsvorgangs. Ein **Dropout** kann die Wahrscheinlichkeit für **Overfitting** reduzieren, und das Modell robuster machen.
ELU	Aktivierungsfunktion, kurz für *Exponential Linear Unit* [18].
Embedded Machine Learning	Einsatz von **Machine Learning** auf **Embedded-Systemen**
Embedded-Modul oder **Embedded-System**	Eingebettetes Modul/System welches üblicherweise mit ARM® oder x86 Prozessoren, manchmal aber auch Mikrocontrollern ausgestattet ist, und die Abläufe der Anwendung steuert, in der es eingebettet wird [3].
End-to-End	Englisch für Ende zu Ende, hier bezogen auf den Fall in dem ein neuronales Netz alle Arbeitsschritte von Daten Eingang bis zu dem gewünschten Ergebnis selbst übernimmt.
Epoche	Als Trainingsepoche bezeichnet man einen Durchlauf des Trainings durch alle verfügbaren Trainingsdaten, unabhängig zufälliger Augmentationen. Die Iterationen einer Epoche hängen also direkt mit der Anzahl der Datenpunkte zusammen, beziehungsweise entsprechen der Anzahl der Iterationen geteilt durch die Batchgröße [9]
Framework	Frameworks bieten Bibliotheksfunktionen und Schnittstellen in Programmiersprachen, und erleichtern so die Softwareentwicklung [38].
GPU	Graphics Processing Unit (Grafikprozessor)
GUI	Kurz für *Graphical User Interface*, also eine grafische Benutzeroberfläche.
Hardwarebeschleunigung	Beschreibt das Verlagern einer Rechenaufgabe auf für diese spezialisierte Hardware [39].
KNN	Kurz für künstliches neuronales Netzwerk

Layer	Englische Bezeichnung für *Schicht*. In dieser Arbeit wird sich mit **Layer** auf die Fachsprache um **Yocto Project** Bezogen.
Linux Shell	„Die Shell ist nur ein Linux-Programm. Sie dient als Verbindungsstelle zwischen dem Benutzer und dem Betriebssystem[…]“ [40]
Lochrasterplatine	“Als Lochrasterplatine bezeichnet man eine universelle Platine mit Kupferstreifen oder -punkten, die in einem Raster aufgebracht sind. Die englische Bezeichnung ist *Stripboard* oder *Veroboard*. Diese industriell gefertigten Platinen werden gerne zum Aufbau von Prototypen benutzt.” [41]
loss	Englisch Begriff für die Abweichung bzw. den Fehler zwischen Schätzung des neuronalen Netzwerks und des erwarteten Ergebnisses.
Machine Learning	Fachsprache für maschinelles Lernen, bezeichnet das selbstständige (maschinelle) Lernen aus Daten [12]
ML	kurz für **Machine Learning**
NPU	Abkürzung für Neural Processing Unit, Bezeichnung für den auf neuronale Netze spezialisierten Hardwarebeschleuniger in dem i.MX 8M Plus Prozessor von NXP.
Open Source	„**Open Source**, das heißt offener Quellcode und meint gemeinhin Software, die jeder nach Belieben studieren, benutzen, verändern und kopieren darf.“ [42]
Overfitting	Ein neuronales Netzwerk wird mit einer zu kleinen Menge an Trainingsdaten zu viel trainiert, sozusagen übertrainiert und beginnt die Trainingsdaten auswendig zu lernen. Siehe Kapitel 6.3 Absatz 3 bis 4.
pooling layer	Eine Schicht in einem **CNN**, zur Reduzierung der Dimension der Matrix einer vorhergehenden Schicht durch Mittelung oder wählen des Maximums [9].
ReLU	Aktivierungsfunktion, kurz für Rectified Linear Unit [18]
Roadfeatures	Anglizismus. Bezieht sich auf die Merkmale einer Straße, hier spezieller auf visuelle Merkmale.
Setter Methode	In der objektorientierten Programmierung eine Methode für den Zugriff auf ein Objektattribut, hier

	um das Attribut zu setzen. Analog hierzu die *Getter Methode* mit der sich das Objektattribut abfragen lässt.
SMD	Kurz für „Surface Mounted Device", also Bauteile die direkt auf die Platinenoberfläche gelötet werden.
Starterkit	Kit zur Evaluierung für **TQ-Module**
Step-by-Step	Englisch für Schritt für Schritt, hier exaktes Gegenteil zu **End-to-End**.
Tensor	Der Begriff **Tensor** beschreibt allgemein eine Datenstruktur beliebiger Dimension, und umfasst einfache Skalare, Vektoren, Matrizen oder **Arrays**. Der Tensor ist die Datenstruktur, die in TensorFlow hauptsächlich verwendet wird. [9].
TensorFlow	Ein Open Source **Framework** für **Machine Learning** entwickelt von Google [8].
TensorFlow Lite	Eine abgespeckte Version von TensorFlow für mobile Geräte sowie Embedded-Systeme im Allgemeinen.
tflite	Abkürzung für **TensorFlow Lite** und Dateiendung von **ML**-Modellen, die innerhalb von **TensorFlow Lite** laufen können.
THT	Kurz für „Through Hole Technology", also Bauteile die durch die Platine gesteckt gelötet werden.
TQ-Modul	Bezeichnung eines von der Firma TQ-Systems hergestelltes **Embedded-Moduls** in Kurzform.
Vision-Anwendung	Eine Anwendung welche die Informationen einer Kamera verarbeitet, zum Beispiel mit einem neuronalen Netz.
x86 Architektur	Eine ursprünglich von Intel entwickelte CISC Architektur (Complex Instruction Set Computer) [35].
Yocto Project	Yocto Projekt ist ein Werkzeug um eigene Linux-Distributionen zu erstellen und zu bauen [13].

Abbildungsverzeichnis

Abbildung 1: TQMa8MPxL - Embedded Modul von TQ-Systems 6
Abbildung 2: STKa8MPxL - Starterkit von TQ-Systems 7
Abbildung 3: Vorschau auf das selbstfahrende Fahrzeug mit STKa8MPxL 9
Abbildung 4: Beispiel für ein KNN aus vollständig verbundenen Schichten (vereinfacht) 12
Abbildung 5: Beispiel für ein CNN (vereinfacht) 13
Abbildung 6: Nvidia Modellarchitektur [16] 16
Abbildung 7: Data-Collection-Fahrzeug 21
Abbildung 8: Steuerung mit PlayStation® Controller 23
Abbildung 9: selbstfahrendes Fahrzeug 27
Abbildung 10: Blockschaltbild Aufbau des selbstfahrenden Fahrzeugs mit Starterkit 27
Abbildung 11: TFOD Modelmaker (Collect Tab) 29
Abbildung 12: LabelIMG (Labeln roter und grüner Ampeln) 30
Abbildung 13: TFOD Modelmaker (Testen eines Modells, dahinter der Test/Export-Tab zu sehen) 30
Abbildung 14: PiCar auf Parkettboden mit brauner Fahrbahnmarkierung 34
Abbildung 15: Das PiCar, nachdem es an der Ampel gehalten hat. 38
Abbildung 16: Ausschnitt aus dem GUI von eIQ Portal 39
Abbildung 17: Ausschnitt aus dem Newsletter von TQ-Embedded vom 17.03.22 41

Literaturverzeichnis

[1] TQ-Systems GmbH, „TQ-Group," [Online]. Available: https://www.tq-group.com/de/unternehmen/ueber-uns/. [Zugriff am 11 Februar 2022].

[2] TQ-Systems GmbH, „TQ | Embedded," [Online]. Available: https://www.tq-group.com/de/produkte/tq-embedded/. [Zugriff am 11 Februar 2022].

[3] K. Zöpf, „Embedded-ARM-Module: Nutzen und Markttrends," 27 Juli 2017. [Online]. Available: https://www.all-electronics.de/elektronik-entwicklung/embedded-arm-module-nutzen-und-markttrends.html. [Zugriff am 15 Februar 2022].

[4] TQ-Systems GmbH, „TQ | STKa8MPxL," [Online]. Available: https://www.tq-group.com/de/produkte/tq-embedded/arm-architektur/stka8mpxl/. [Zugriff am 11 Februar 2022].

[5] M. Breiling und J. U. Garbas, „Embedded Machine Learning," [Online]. Available: https://www.iis.fraunhofer.de/de/ff/kom/ki/embedded-ml.html. [Zugriff am 11 Februar 2022].

[6] NXP Semiconductors, „i.MX Machine Learning User's Guide," [Online]. Available: https://www.nxp.com/docs/en/user-guide/IMX-MACHINE-LEARNING-UG.pdf. [Zugriff am 14 Februar 2022].

[7] S. Hari, „Best Machine Learning Frameworks(ML) for Experts in 2022," 7 Januar 2022. [Online]. Available: https://hackr.io/blog/machine-learning-frameworks. [Zugriff am 16 Februar 2022].

[8] M. Abadi, A. Agarwal, P. Barham und E. Brevdo, „TensorFlow: Large-scale machine learning on heterogeneous systems," 9 November 2015. [Online]. Available: Software available from tensorflow.org.

[9] „Machine Learning Glossary," [Online]. Available: https://developers.google.com/machine-learning/glossary. [Zugriff am 17 Februar 2022].

[10] tensorflow.org, „Tutorials," [Online]. Available: https://www.tensorflow.org/tutorials. [Zugriff am 16 Februar 2022].

[11] T. Ruscica, „TensorFlow 2.0 Complete Course - Python Neural Networks for Beginners Tutorial," 3 März 2020. [Online]. Available: https://www.youtube.com/watch?v=tPYj3fFJGjk. [Zugriff am 16 Februar 2022].

[12] N. Schaaf, „Neuronale Netze: Ein Blick in die Black Box," 14 Januar 2020. [Online]. Available: https://www.informatik-aktuell.de/betrieb/kuenstliche-intelligenz/neuronale-netze-ein-blick-in-die-black-box.html. [Zugriff am 17 Februar 2022].

[13] E. Brown, „Real-World Build Tips for Yocto," 11 April 2018. [Online]. Available: https://www.linux.com/topic/embedded-iot/real-world-build-tips-yocto/. [Zugriff am 15 Februar 2022].

[14] M. Niebel, M. Schiffer und G. Herburger, „GitHub | meta-tq," [Online]. Available: https://github.com/tq-systems/meta-tq.git. [Zugriff am 11 Februar 2022].

[15] m.-t. r. s. p. n. und a. , „OpenEmbedded Layer Index," [Online]. Available: https://layers.openembedded.org/layerindex/branch/master/layers/. [Zugriff am 11 Februar 2022].

[16] M. Bojarski, B. Firner, B. Flepp, L. Jackel, U. Muller, K. Zieba und D. D. Testa, „End-to-End Deep Learning for Self-Driving Cars," 17 August 2016. [Online]. Available: https://developer.nvidia.com/blog/deep-learning-self-driving-cars/. [Zugriff am 14 Februar 2022].

[17] D. Tian, „DeepPiCar — Part 5: Autonomous Lane Navigation via Deep Learning," 2 Mai 2019. [Online]. Available: https://towardsdatascience.com/deeppicar-part-5-lane-following-via-deep-learning-d93acdce6110. [Zugriff am 21 Februar 2022].

[18] B. Fortuner, P. Piprotar, I. Riley und deepjoy2002, „Activation Functions — ML Glossary documentation," 2 September 2021. [Online]. Available: https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html. [Zugriff am 21 Februar 2022].

[19] NXP Semiconductors, „IMXLXYOCTOUG - i.MX Yocto Project User's Guide," 17 Dezember 2021. [Online]. Available: https://www.nxp.com/docs/en/user-guide/IMX_YOCTO_PROJECT_USERS_GUIDE.pdf. [Zugriff am 4 März 2022].

[20] j. und C. Lee, „sunfounder/SunFounder_PiCar," 16 November 2016. [Online]. Available: https://github.com/sunfounder/SunFounder_PiCar. [Zugriff am 21 Februar 2022].

[21] AnyDesk Software GmbH, „AnyDesk," [Online]. Available: https://anydesk.com/de/downloads. [Zugriff am 25 Februar 2022].

[22] OpenCV team, „opencv.org," [Online]. Available: https://opencv.org/. [Zugriff am 25 Februar 2022].

[23] A. Spirin, „pyPS4Controller · PyPI," [Online]. Available: https://pypi.org/project/pyPS4Controller/. [Zugriff am 25 Februar 2022].

[24] F. Chamaki, „Medium - Write your own Custom Data Generator for TensorFlow Keras," 24 März 2021. [Online]. Available: https://medium.com/analytics-vidhya/write-your-own-custom-data-generator-for-tensorflow-keras-1252b64e41c3. [Zugriff am 2 März 2022].

[25] tensorflow.org, „TensorFlow - Post-training quantization," 17 November 2021. [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_quantization. [Zugriff am 9 März 2022].

[26] J. Huang, V. Rathod, V. Birodkar, A. Myers , Z. Lu , R. Votel , Y.-h. Chen und D. Chow , „GitHub - Tensorflow Object Detection API," [Online]. Available: https://github.com/tensorflow/models/tree/master/research/object_detection. [Zugriff am 2022 März 10].

[27] N. Renotte, „GitHub - nicknochnack/TFODCourse," 3 April 2021. [Online]. Available: https://github.com/nicknochnack/TFODCourse. [Zugriff am 10 März 2022].

[28] L. Vladimirov, T. Schouten, N. Aherne, A. L und S. Dhull, „TensorFlow 2 Object Detection API tutorial," 13 September 2021. [Online]. Available: https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html. [Zugriff am 10 März 2022].

[29] T. Lin, „PyPi - LabelIMG," 11 Oktober 2021. [Online]. Available: https://pypi.org/project/labelImg/. [Zugriff am 11 März 2022].

[30] T. WANG und M. Daoust, „GitHub - Raspberry PI,“ 24 September 2019. [Online]. Available: https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/raspberry_pi. [Zugriff am 14 März 2022].

[31] NXP Semiconductors, „NXP - eIQ® Toolkit,“ 18 Januar 2022. [Online]. Available: https://www.nxp.com/design/software/development-software/eiq-ml-development-environment/eiq-toolkit-for-end-to-end-model-development-and-deployment:EIQ-TOOLKIT. [Zugriff am 22 März 2022].

[32] J. Wells, „Mathematica 12 Available on the New Raspberry Pi 4—Wolfram Blog,“ 11 Juli 2019. [Online]. Available: https://blog.wolfram.com/2019/07/11/mathematica-12-available-on-the-new-raspberry-pi-4/. [Zugriff am 25 März 2022].

[33] TQ-Systems GmbH, „YouTube | TQ-Group - Performance-Vergleich Raspberry PI 4 vs. TQMa8MPxL,“ 15 März 2022. [Online]. Available: https://www.youtube.com/watch?v=WANShskrws8. [Zugriff am 24 März 2022].

[34] Talend Germany GmbH, „Was ist eine API? Einfach erklärt!,“ [Online]. Available: https://www.talend.com/de/resources/was-ist-eine-api/. [Zugriff am 16 Februar 2022].

[35] B. Leitenberger, „CISC und RISC - Die Gegensätze der Rechnerarchitekturen.,“ [Online]. Available: https://www.bernd-leitenberger.de/cisc-risc.shtml. [Zugriff am 15 Februar 2022].

[36] NumPy Developers, „Array objects — NumPy v1.22 Manual,“ [Online]. Available: https://numpy.org/doc/stable/reference/arrays.html. [Zugriff am 2 März 2022].

[37] S. Augsten, „Dev-Insider - Definition „Code Build“ Was ist ein Build?,“ 27 April 2018. [Online]. Available: https://www.dev-insider.de/was-ist-ein-build-a-702737/. [Zugriff am 4 März 2022].

[38] D. Kranz und J. Landwehr, „Was ist ein Application Framework und welche gibt es?,“ 20 Dezember 2019. [Online]. Available: https://it-talents.de/it-wissen/application-framework/. [Zugriff am 16 Februar 2022].

[39] Routech, „Was ist Hardwarebeschleunigung und warum ist sie wichtig?,“ 3 November 2020. [Online]. Available: https://www.routech.ro/de/was-ist-hardwarebeschleunigung-und-warum-ist-sie-wichtig/. [Zugriff am 15 Februar 2022].

[40] O. Vanhoefer, „Linux.Fibel.org - 4.1 Was ist eine Shell?,“ [Online]. Available: http://fibel.org/linux/lfo-0.6.0/node51.html. [Zugriff am 8 März 2022].

[41] E. und S. , „Lochrasterplatine – Mikrocontroller.net,“ 24 November 2014. [Online]. Available: https://www.mikrocontroller.net/articles/Lochrasterplatine. [Zugriff am 23 Februar 2022].

[42] Bundeszentrale für politische Bildung, „Open Source | bpb.de,“ [Online]. Available: https://www.bpb.de/themen/digitalisierung/opensource/. [Zugriff am 16 Februar 2022].