Universität Bremen

Bachelorarbeit

Approximate Computing für Neuronale Netze in selbstfahrenden Autos

Autor
Christian Friedrich Coors
Informatik

Gutachter
Prof. Dr. Rolf Drechsler
Dr. Felix Putze

Abgabe
2019-02-25

Eidesstattliche Erklärung

Ich versichere, den Bachelor-Report oder den von mir zu verantwortenden Teil einer Gruppenarbeit[1] ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den _____

_____

(Unterschrift)

[1] Bei einer Gruppenarbeit muss die individuelle Leistung deutlich abgrenzbar und bewertbar sein und jeweils den Anforderungen aus § 12 Abs. 1 der Prüfungsordnung für den Studiengang *Bachelor of Science Informatik* entsprechen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Künstliche Intelligenz (KI) wird heutzutage vielfältig eingesetzt. Sie wird als Lösungsansatz für viele sehr unterschiedliche Problemkategorien genutzt, beispielsweise dem Erkennen von Fußgängern in Videos [4], dem Spielen von Go [22] oder in der Unterstützung der Verarbeitung natürlicher Sprache [23]. Wichtiger Vertreter der Modelle der künstlichen Intelligenz sind künstliche Neuronale Netze (KNNs). Für Szenarien der Bildverarbeitung haben sich Convolutional Neural Networks (CNNs) als geeignete Netzarchitektur erwiesen [19] [15]. Diese bestehen aus speziellen „Convolutional Layern", welche mithilfe von Filtern bestimmte Features aus Bilddaten extrahieren, deren Existenz und Position dann im weiteren Berechnungsverlauf von Bedeutung ist. CNNs wurden hauptsächlich für Mustererkennung innerhalb von Bildern verwendet, dienen mittlerweile aber auch als Komplettlösung zur Lenkung autonomer Fahrzeuge [2].

Ein großes Problem von CNNs ist, dass sie in ihrer Verwendung sehr rechenintensiv sind. Bereits in kleinen Netzen gehen tausende Parameter in die Berechnung ein. Um die Auswertung von CNNs zu beschleunigen wurden bisher verschiedene Ansätze angewendet, sehr große Geschwindigkeitsvorteile bietet beispielsweise die Auslagerung eines Großteils der Arbeit auf spezielle Hardware, etwa Grafikprozessoren oder spezielle FPGAs/ASICs.

Ein neuer Ansatz ist die Verwendung von approximierter Berechnung (Approximate Computing). Beim Approximate Computing werden Berechnungen nicht exakt ausgeführt, stattdessen werden die Ergebnisse nur bis zu einer gewissen Genauigkeit berechnet. Ein Beispiel ist in Tabelle 1.1 zu finden. Dort werden exemplarisch zwei reelle Zahlen multipliziert und die Ergebnisse mit approximierten Ergebnissen verglichen. Zur Approximation wird die Anzahl der Exponenten- und Mantissenbits, die im Rechner zur Darstellung der Zahl im IEEE-754-Format stattfindet, reduziert. In der zweiten Spalte ist der benötigte Speicher pro Ergebnis angegeben. Aus der Tabelle ist ersichtlich, dass durch die Reduktion des Speicherverbrauchs die Genauigkeit des Ergebnisses abnimmt. Zusätzlich zur Reduktion des benötigten Speichers können bei entsprechender Hardware, die diese Datentypen implementiert, weitere Vorteile erzielt werden. Die Chipfläche kann reduziert werden, insgesamt verringert sich der Stromverbrauch und die Berechnungen können schneller erfolgen [37].

Diese Art der Berechnung eignet sich auch für KNNs, da diese naturgemäß nie exakt rechnen sondern selber nur Approximationen an Funktionen sind, auf die sie trainiert wurden. Zusätzlich sind sie verhältnismäßig tolerant gegenüber fehlerhaften Berechnungen [38]. In dieser Arbeit wird untersucht, wie sich die Reduktion der Rechengenauigkeit in einem CNN auf einen konkreten Anwendungsfall auswirkt: selbstfahrende Autos. Für selbstfahrende Autos werden gerne Neuronale Netze verwendet, da der mögliche Input bei entsprechender Sensorik (insbesondere Kameras) auch bei niedrigen Auflösungen bereits sehr groß ist. Um dies zu untersuchen wurde die sogenannte „Donkey Car"-Plattform gewählt, eine Hardware- und Softwareplattform für selbstfahrende Modellautos [6]. In das Neuronale Netz, welches in der Software dieser Platt-

a = 2,5752 b = 5,552

Datentyp	Bits	a · b
exaktes Ergebnis	-	14,2975104
double	64	14,297510400000000174713932210579514503479003906 25
float	32	14,2975101470947265625
4 Exponenten-, 16 Mantissenbits	21	14,296875
4 Exponenten-, 8 Mantissenbits	13	14,125
4 Exponenten-, 4 Mantissenbits	9	12
2 Exponenten-, 2 Mantissenbits	5	3

Tabelle 1.1.: Vergleich der Berechnungen von IEEE-754 Fließkommadatentypen mit Approximationen durch Bitreduktion

form verwendet wird, wurde approximierte Berechnung eingebaut.

In KNNs gibt es sehr viele Designparameter, von denen die Performance stark abhängig ist. Diese Arbeit beschränkt sich daher explizit auf die Auswirkungen der Genauigkeit der Fließkommaoperationen, nicht auf andere Methodiken zur Verbesserung der Performance.

2. Grundlagen

2.1. Approximate Computing

Fließkommazahlen werden heute in Rechnern sehr häufig nach dem IEEE-754-Standard abgebildet. Die Idee dieses Standards ist es, jede Zahl x in einen Exponenten E, eine Mantisse T und ein Vorzeichenbit S zu zerlegen, sodass gilt $x = (-1)^S \cdot 2^E \cdot T$. Der IEEE-754-Standard gibt mehrere Formate für Fließkommazahlen vor, die sich in der Anzahl der Bits für den Eponenten und der Mantisse unterscheiden. Die wichtigsten beiden sind einfache Genauigkeit (float, 8 Bit Exponent und 23 Bit Mantisse) und doppelte Genauigkeit (double, 11 Bit Exponent und 52 Bit Mantisse). Mit dem zusätzlichen Bit für das Vorzeichen ergeben sich insgesamt 32 Bit für einfache und 64 Bit für doppelte Genauigkeit [12]. Es gibt noch einige Sonderfälle, die im Rahmen dieser Arbeit aber nicht von Relevanz sind.

Ein üblicher Ansatz beim Approximate Computing ist es, die Anzahl der Bits für Exponent und insbesondere der Mantisse zu reduzieren [37]. Die Reduktion der Anzahl der Bits für den Exponenten reduziert effektiv die größte darstellbare Zahl und die Reduktion der Mantissenbits reduziert die Auflösung, also die Anzahl der möglichen Werte zwischen dem aktuellen und nächsten Exponenten. Dadurch ergibt sich, dass es prinzipiell einfacher und sicherer ist, in einem bestehenden System die Mantissenbits zu reduzieren. In diesem Fall werden im Betrag sehr große Zahlen nur noch ungenauer, befinden sich jedoch weiterhin im darstellbaren Zahlenraum.

Ein Vergleich der Präzision von Fließkommazahlen einfacher und doppelter Genauigkeit in Abhängigkeit von ihren Werten in doppelt logarithmischer Darstellung findet sich in Abbildung 2.2. Auf der X-Achse ist der zu speichernde Wert aufgetragen, auf der Y-Achse wird die Präzision, mit der der Wert gespeichert werden kann, abgebildet. Um 10^0, also um 1, können Fließkommazahlen in einfacher Genauigkeit mit einem Fehler in der Größenordnung 10^{-7} gespeichert werden, also mit etwa 7 relevanten Nachkommastellen. Bei doppelter Genauigkeit erhöht sich die Anzahl der Stellen auf knapp 16. Durch eine Veränderung der Anzahl der Mantissenbits kann der Graph also auf der Y-Achse verschoben werden.

Die Idee beim Approximate Computing ist, dass die Anzahl der relevanten Nachkommastellen in manchen Anwendungsfällen unterschiedlich ist und beispielsweise einfache Genauig-

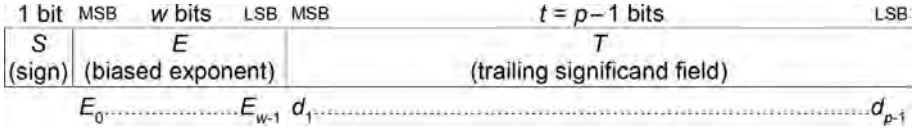


Abbildung 2.1.: Darstellung der IEEE-754-Fließkommazahlen im „binary interchange format“ [12, S. 9]

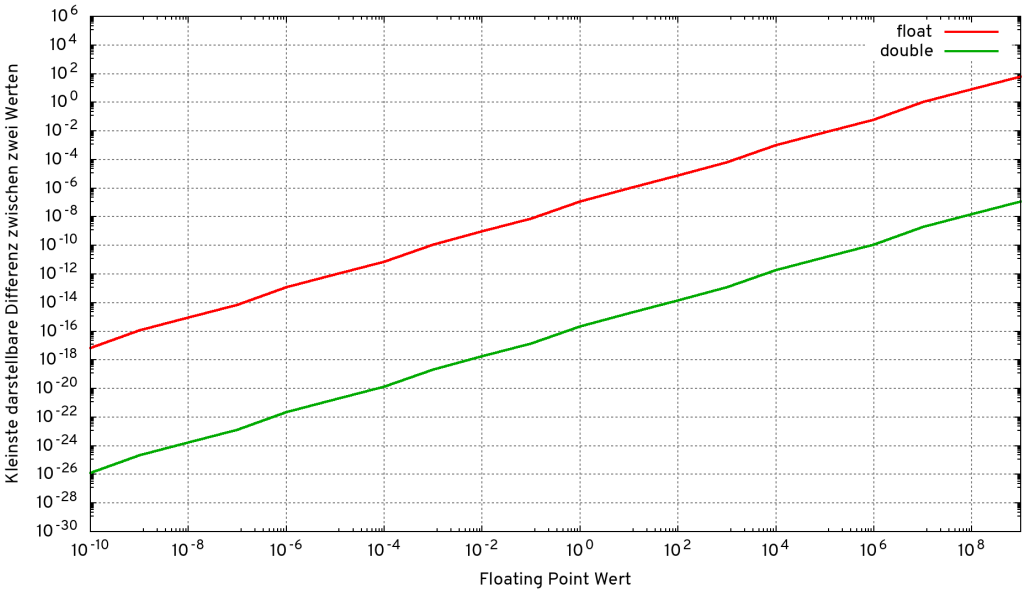


Abbildung 2.2.: Die Genauigkeit von IEEE-754 floats und doubles in Abhängigkeit von ihrem Wert in doppelt logarithmischer Darstellung

keit bereits zu präzise ist [37]. Durch die Einsparung einiger Bits würden sich Berechnungen für diese Anwendungsfälle nicht merklich verschlechtern und gleichzeitig können Bits gespart und die Berechnungsgeschwindigkeit erhöht werden. Bei der Simulation von approximierter Berechnung in Software gibt es verschiedene Ansätze, beispielsweise kann weiter mit normalen floats gerechnet werden, aber nach jeder Zuweisung die zu ignorierenden niederwertigsten Bits des Exponenten und der Mantisse verworfen werden. Ein anderer Ansatz ist es, den Exponenten und die Mantisse separat als Integer mit variabler Breite zu speichern. Solche Softwarelösungen sind immer langsamer als echte Hardwareimplementierungen, da diese nicht nativ in Hardware in einem Schritt auf einer FPU durchgeführt werden. Dies macht es schwierig, den tatsächlichen Anstieg der Berechnungsgeschwindigkeit ohne approximerende Hardware abzuschätzen, erlaubt es aber, den Einfluss von approximierter Berechnung auf die Genauigkeit von Berechnungsverfahren und Algorithmen zu evaluieren.

Eine Softwarebibliothek für approximierte Berechnungen ist ctfloat. ctfloat wurde vom INRIA (Institut national de recherche en informatique et en automatique) entwickelt und ist zum Zeitpunkt dieser Veröffentlichung nicht öffentlich verfügbar [5]. ctfloat hat als Ziel, kompatibel zu CatapultC, einer Hardwaresynthesesprache zu sein, um die Approximation auch in Hardware umsetzen zu können. Es definiert eine C++-Templateklasse, der Approximationsparameter als Templateparameter übergeben werden. Ein Beispiel ist in Listing 2.1 zu sehen.

```
1 #include <iostream>
2 #include <iomanip>
3 #include <ctfloatB.h>
4 int main() {
5     ct_float<4, 4> in1, in2, out;
6     in1 = 2.5752;
7     in2 = 5.552;
```

```
8       out = in1 * in2;
9
10      std::cout << std::setprecision(128);
11      std::cout << "ct_float<4, 4>: " << res_ct_2 << "\n";
12
13      return 0;
14  }
```

Listing 2.1: Beispiel-Code mit ctfloat, der einen Eintrag in Tabelle 1.1 berechnet hat

Dort werden in Zeile 5 die floats mit 4 Exponenten- und 4 Mantissenbits deklariert. In Zeilen 6 und 7 werden ihnen die zu multiplizierenden Zahlen zugewiesen. Intern approximiert ctfloat an dieser Stelle bereits und reduziert die Anzahl der Mantissenbits auf 4. In Zeile 8 wird die Multiplikation durchgeführt. Dazu ist in der Klasse der `operator*` überladen worden, der einen `ct_float` zurückgibt. Die Ausgabe erfolgt in den Zeilen 10 und 11.

2.2. TensorFlow

TensorFlow ist eine Softwareplattform für maschinelles Lernen, die von Google als freie Software veröffentlicht wurde. Mit TensorFlow können Datenflussgraphen erzeugt werden, deren Knoten aus Operationen bestehen und an deren Kanten Tensoren weiter durch den Graphen gegeben werden („Flow“). Ein Tensor im TensorFlow-Kontext ist ein n-dimensionales Array. Ein eindimensionaler Tensor wird auch Vektor genannt, ein zweidimensionaler Tensor auch Matrix. In einem Tensor hat jeder Eintrag den gleichen Datentyp, zum Beispiel `int32`, `string` oder `float32`. Die Operationen rechnen mit diesen Tensoren, wobei jeder Operation eine Signatur für kompatible Tensoren hat. So nimmt etwa die Matrixmultiplikation `MatMul` zwei zweidimensionale Eingabetensoren und erzeugt einen zweidimensionalen Ausgabetensor. Die Signatur lässt nur numerische Werte zu, einen `string`-Tensor kann die Operation nicht verarbeiten. Ein weiteres Beispiel ist die `Const` Operation, die keine Eingabe und nur eine Ausgabe, einen konstanten Tensor hat [1].

Für viele Operationen steht in TensorFlow Broadcasting zur Verfügung, was die implizite Erweiterung der Dimensionen von Tensoren bedeutet. Werden beispielsweise zwei Tensoren unterschiedlicher Dimensionalität addiert, so erweitert TensorFlow den kleineren Tensor [31]:

$$\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix} + \begin{pmatrix}7 & 8 & 9\end{pmatrix} \Rightarrow \begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix} + \begin{pmatrix}7 & 8 & 9\\ 7 & 8 & 9\end{pmatrix} = \begin{pmatrix}8 & 10 & 12\\ 11 & 13 & 15\end{pmatrix}$$

Die Architektur von TensorFlow ist in Abbildung 2.3 dargestellt. Der untere große Kasten beschreibt den Kern von TensorFlow. Dieser ist in C++ implementiert und hat verschiedene Aufgaben. Dazu zählen unter anderem die Abstraktion über diverse Devices, also CPUs, GPUs und TPUs (Tensor Processing Units), der Implementierung von Operationen und deren Kerneln für unterschiedliche Devices, sowie die Parallelisierung des Datenflusses durch den Graphen. Dabei können mehrere Operationen gleichzeitig auf unterschiedlichen Devices ausgeführt werden. Auf dem TensorFlow Kern setzt aus Kompatibilitätsgründen eine C API auf, auf deren Basis wiederum eine Python- und C++ API aufsetzen. Der Umweg über die C API vereinfacht die Verwendung von TensorFlow aus anderen Programmiersprachen. Auf die Python API wiederum setzen High-Level-Bibliotheken wie Keras auf [14].

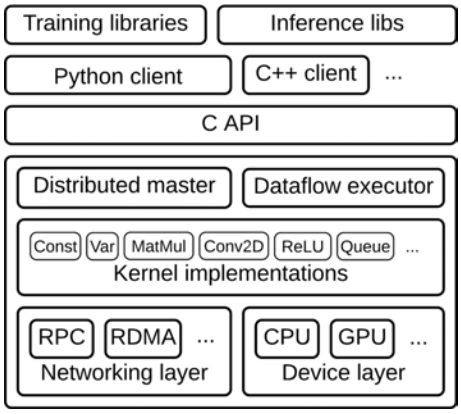


Abbildung 2.3.: Eine High-Level Ansicht auf die TensorFlow-Architektur [1]

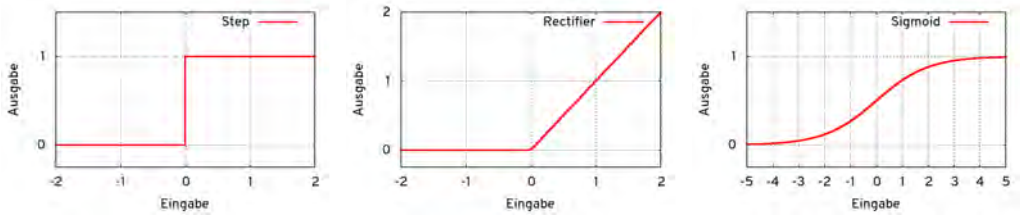


Abbildung 2.4.: Verschiedene Aktivierungsfunktionen für Neuronen [16, Nach Abbildungen 5.1 und 5.2]

2.3. Künstliche Neuronale Netze

Neuronale Netze bestehen aus Layern (Schichten) von Neuronenmodellen. Ein Neuronales Netz hat einen Input Layer, einen Output Layer sowie beliebig viele hidden Layer. Ein Neuron ist eine Verarbeitungseinheit. Ein Beispiel für ein Neuron ist das Perzeptron (auch Schwellenwertelement genannt), welches einen Vektor von Eingabewerten x_i bekommt und einen einzelnen Ausgabewert y berechnet. Jedem Eingabewert ist ein Gewicht w_i zugeordnet, mit dem er multipliziert wird. Das Neuron hat zusätzlich einen Schwellenwert τ [16].

$$y = \left\{ \begin{array}{ll} 1, & \text{falls } \sum_{i=1}^{n} w_i x_i \geq \tau \\ 0, & \text{sonst} \end{array} \right\}$$

Die Ausgabe eines Neurons ist die Eingabe in ein Neuron des nächsten Layers, oder im Output Layer ein Element des Ausgabevektors. Je mehr Neuronen ein Neuronales Netz beinhaltet, desto komplexer kann die Funktion sein, die durch das Netz berechnet wird. Die Gewichte w_i und der Schwellenwert τ werden in KNNs mit unterschiedlichen Methoden trainiert („gelernt“), die hierbei verwendeten Verfahren sind im Rahmen dieser Arbeit jedoch nicht relevant.

Im Gegensatz zum Perzeptron, welches nur zwei mögliche Ausgabewerte besitzt, kann die Ausgabe von Neuronen theoretisch beliebig sein. In der Praxis werden häufig Ausgaben zwischen 0 und 1 oder -1 und +1 verwendet. Dazu kommt eine Aktivierungsfunktion zum Einsatz, die aus der Summe der Eingaben und Gewichte $x = \sum w_i x_i$ eine Ausgabe berechnet. Einige Aktivierungsfunktionen sind in Abbildung 2.4 zu sehen.

Die erste Aktivierungsfunktion (Step, Schrittfunktion) erzeugt die gleiche Ausgabe wie das

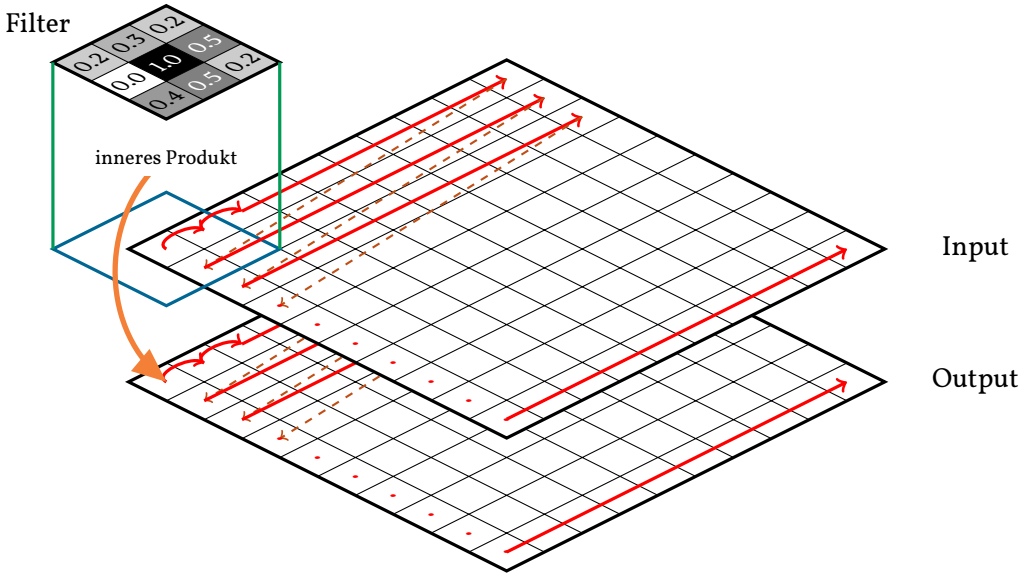


Abbildung 2.5.: Grafische Darstellung einer 2D Convolutional Operation mit einem 3×3 Filter auf einem 10×10 Input (Grafik basierend auf [10])

Perzeptron. Die zweite Aktivierungsfunktion (Rectifier) $y = \max(0, x)$ setzt alle negativen Eingaben auf 0. Die dritte Aktivierungsfunktion (Sigmoid) $y = \frac{1}{1+e^{-x}}$ nähert sich den biologischen Neuronen an, die auch mehr oder minder stark aktiviert sein können [16].

2.4. Convolutional Neural Networks

CNNs erweitern KNNs um Convolutional Layer. Ein Convolutional Layer in einem Neuronalen Netz bekommt als Eingabe eine Menge von Input-Tensoren, eine Menge von Filter-Tensoren sowie einige Optionen. Im Fall der zweidimensionalen Convolution sind alle Tensoren zweidimensional. Die Input-Tensoren werden auch als Channel bezeichnet, da sie in einem Bild etwa drei Farbkanäle repräsentieren können. Für jede Input-Filter-Kombination wird dann die Convolution durchgeführt (siehe Abbildung 2.5). Bei der Convolution wird der Filter schrittweise über die Inputmatrix geschoben. Zu jedem Schritt wird das innere Produkt des Filters und des darunterliegenden Ausschnitts der Matrix (blaues Quadrat) gebildet. Der sich ergebende Wert wird in die Outputmatrix geschrieben. Am Rand des Inputs muss die Matrix erweitert werden (Padding), dies kann über verschiedene Methoden geschehen (beispielsweise Setzen auf 0, Kopie des nächsten Wertes o. Ä.), dieses Verhalten wird über die Optionen der Operation gesteuert. Über die Optionen können weiterhin die Schrittweiten (Strides) in X- und Y-Richtung bestimmt werden, bei Werten größer 1 verkleinert sich der Ausgabetensor entsprechend. Zudem kann ein Subsampling erfolgen, welches auch die Größe des Output-Tensors verringert. Dabei wird das Maximum von zwei nebeneinanderliegenden Ausgabewerten gebildet und zusammengefasst (Pooling) [20].

Das innere Produkt ist für die Convolution so definiert, dass jeder Eintrag des Filters mit dem entsprechenden Eintrag des Inputs multipliziert wird und diese Ergebnisse aufsummiert werden. Dies geschieht analog zum Skalarprodukt von Vektoren. Hier am Beispiel von zwei 2×2

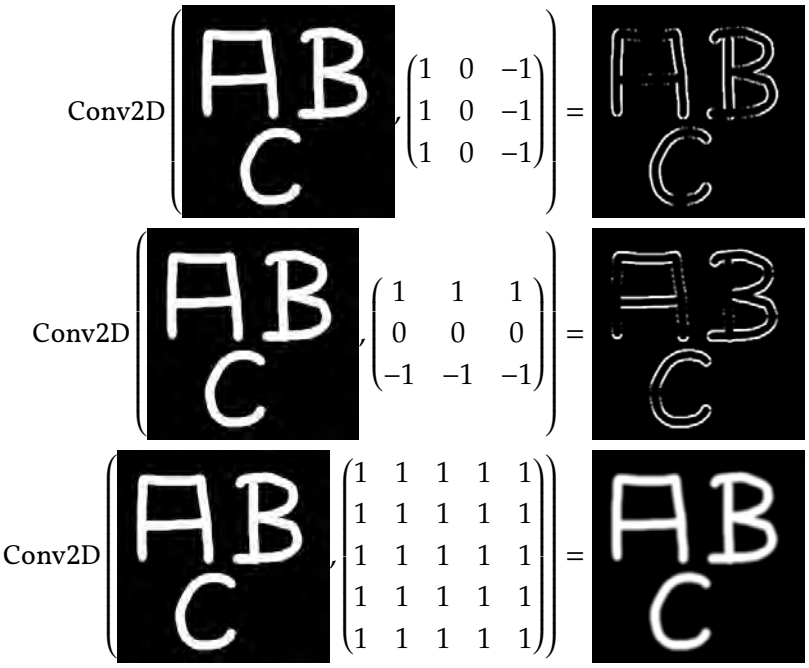


Abbildung 2.6.: Beispiele einer 2D Convolution auf einem Farbkanal eines Bildes. Grafiken erstellt mit dem Tool von [26].

Matrizen:

$$\text{innerProduct}\left(\begin{pmatrix} i_{11} & i_{12} \\ i_{21} & i_{22} \end{pmatrix}, \begin{pmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{pmatrix}\right) = f_{11} \cdot i_{11} + f_{12} \cdot i_{12} + f_{21} \cdot i_{21} + f_{22} \cdot i_{22}$$

TensorFlow implementiert den gesamten Vorgang in einer Operation mit dem Namen Conv2D. Mit dieser Operation werden zweidimensionale Eingabetensoren, also beispielsweise einzelne Farbkanäle von Bildern, gefiltert. Exemplarisch ist der Vorgang in Abbildung 2.6 dargestellt. Als erster Parameter der Conv2D-Funktion wird der Input übergeben, als zweiter Parameter der zu verwendende Filter. Dargestellt sind in der Abbildung die Erkennung von vertikalen und horizontalen Kanten sowie ein weiterer Filter, der das Bild unschärfer macht. In der Praxis sind die Filter trainierbare Parameter, die auch ganz anders aussehen können.

In CNNs gibt es noch zwei weitere wichtige Layer-Typen. Einerseits Dropout Layer, die Overfitting verhindern sollen [25], in dieser Arbeit aber nicht von Relevanz sind. Andererseits Flatten Layer, die einen n-dimensionalen Eingabetensor in einen eindimensionalen Tensor überführen.

$$\text{flatten}\left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\right) = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$$

Die tatsächliche Anordnung als Zeilen- oder Spaltenvektor und ob nach Zeilen- oder Spalten sortiert wird hängt von der Implementierung ab. Die Ausgabe des Flatten Layers wird in der Regel in Neuronalen Layern wie in KNNs weiterverarbeitet.

2.5. Weitere Begriffe

Begriff	Erklärung
Base64	Ein Datenformat, welches die Kodierung von Binärdaten in druckbaren Zeichen ermöglicht [13].
Casting	Die explizite oder implizite Umwandlung eines Datentyps in einen anderen Datentyp.
Eigen	Eine Softwarebibliothek für lineare Algebra, wird intern von TensorFlow verwendet.
Funktor	Ein Funktionsobjekt in der objektorientierten Programmierung.
JSON	Die JavaScript Object Notation, ein strukturiertes Datenformat [3].
Lazy Evaluation	Die Strategie, ein Ergebnis nicht sofort zu berechnen, sondern erst, wenn es benötigt wird.
WebSocket	Ein Kommunikationsprotokoll auf Basis von TCP, ursprünglich für Webbrowser gedacht [11].

Tabelle 2.1.: Weitere wichtige Definitionen

3. Donkey Car

Donkey Car ist eine Open Source Plattform für selbstfahrende Modellautos [6]. Ursprünglich gedacht war diese Platform für 1:16 RC-Autos in Verbindung mit einem Raspberry Pi und einer Kamera. Später wurde die Platform auch um andere Architekturen erweitert, etwa deutlich kleinere oder deutlich größere Modellautos (1:5 – 1:32), sowie auf andere Einplatinenrechner (ODROID, NVIDIA Jetson TX2) [8].

Für diese Arbeit wurde ein XciteRC Eagle Monster Truck [39] im 1:16-Maßstab auf die Donkey Car Plattform umgerüstet, da die offiziell vom Donkey Car Projekt unterstützten Modellautos in Europa nicht verfügbar waren. Entsprechend musste auch beim Umbau von der offiziellen Anleitung an einigen Stellen abgewichen werden. Zur Dokumentation entstand im Rahmen der Arbei auch eine Umbauanleitung für dieses Auto, diese befindet sich im Anhang unter Abschnitt A. Ein Foto des Autos ist in Abbildung 3.1 zu sehen. Die benötigten Materialien wurden von der Arbeitsgruppe Rechnerarchitektur zur Verfügung gestellt.

3.1. Architektur

Die Donkey Car Software ist in Python 3 geschrieben und modular aufgebaut, um die verschiedenen Eigenschaften und Funktionen der verfügbaren Modellautos auszunutzen. Dazu werden die Autos aus sogenannten Parts zusammengesetzt, die jeweils eine gewisse Funktion erfüllen und miteinander verbunden werden können. Technisch sind die Parts als verschiedene Python-Klassen realisiert, deren Instanzen über eine Graphenstruktur verbunden werden. Die wichtigsten Parts sind mit einer kurzen Beschreibung in Tabelle 3.1 zu finden. Die Graphen zum Generieren der Trainingsdaten und zum selbstständigen Fahren sind in den Abbildungen 3.2 und

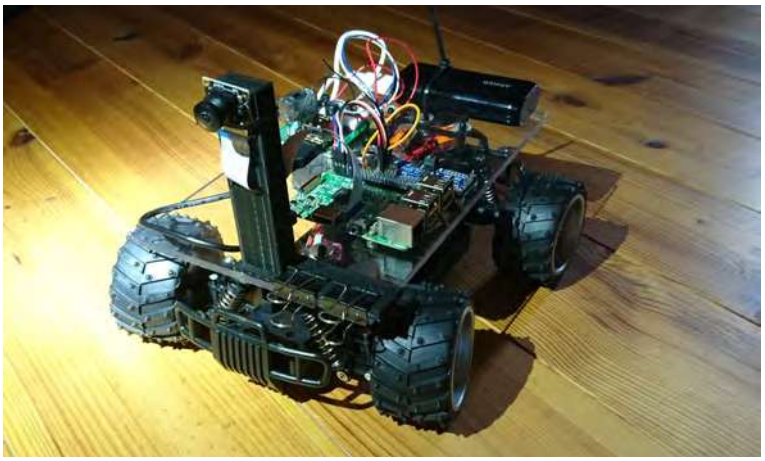


Abbildung 3.1.: Das für diese Arbeit umgebaute Donkey Car

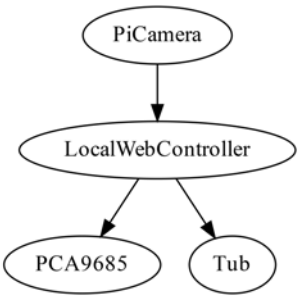


Abbildung 3.2.: Der Standard-Graph zum Generieren der Trainingsdaten

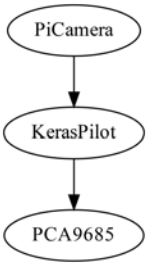


Abbildung 3.3.: Der Standard-Graph zum selbstständigen Fahren

3.3 zu sehen.

3.2. Neuronales Netz

Der `KerasPilot`-Part des Donkey Cars enthält Code zur Konstruktion eines Neuronalen Netzes. Die Netztopologie ist in Abbildung 3.4 dargestellt. Der Knoten `img_in` wird mit dem zu berechnenden Bild gefüllt. Dieses hat die Dimensionen (160, 120, 3), 160x120 Pixel mit 3 Farbkanälen (rot, grün, blau). Die Konfiguration der Convolutional Layer ist in Tabelle 3.2 zu sehen. `Dense_1` enthält 100 Neuronen mit linearer Aktivierungsfunktion und `Dense_2` 50 Neuronen mit linearer Aktivierungsfunktion. In den Dropout Layern werden jeweils 10 % der Einträge in den Tensoren auf 0 gesetzt. `angle_out` und `throttle_out` sind Layer mit jeweils einem Neuron und linearer Aktivierungsfunktion.

Insgesamt besitzt dieses Netz damit 266.628 trainierbare Parameter. Die Netztopologie basiert auf einer in einem von NVIDIA veröffentlichten Paper vorgeschlagenen Netztopologie [2] zum autonomen Fahren [7]. Das Netz besitzt zwei separate Ausgaben: `angle_out` (Lenkwinkel) und `throttle_out` (Beschleunigung). Im Donkey Car wird allerdings nur `angle_out` verwendet und die eigentliche Beschleunigung/Zielgeschwindigkeit vom Benutzer vorgegeben, da die Entwickler Probleme mit der Beschleunigungsausgabe haben. Für den normalen Anwendungsfall des Donkey Cars, das Fahren von Rennen, ist diese Vorgehensweise akzeptabel.

Diese Netzbeschreibung wird zunächst mit der High-Level-Bibliothek Keras durchgeführt. Keras setzt auf TensorFlow auf und erzeugt ein TensorFlow-Netz. Dies geschieht normalerweise transparent für den Nutzer der Bibliothek. Es ist jedoch möglich, über die Keras API auf das erstellte TensorFlow-Netz zuzugreifen und es zu exportieren. Das Netz kann dann außerhalb von Keras weiterverwendet werden, beispielsweise um es in dem Visualisierungstool Tensor-

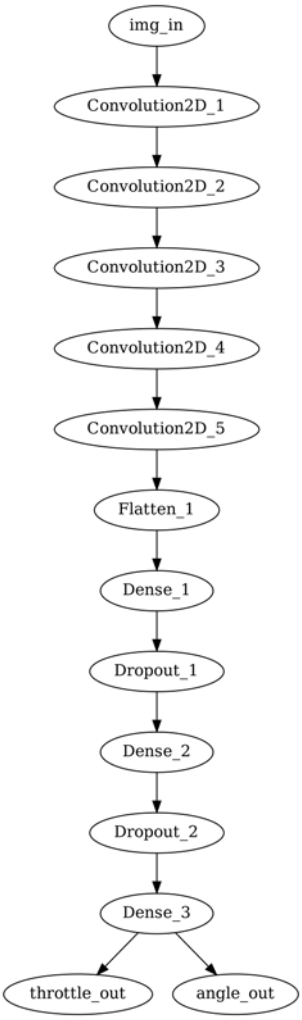
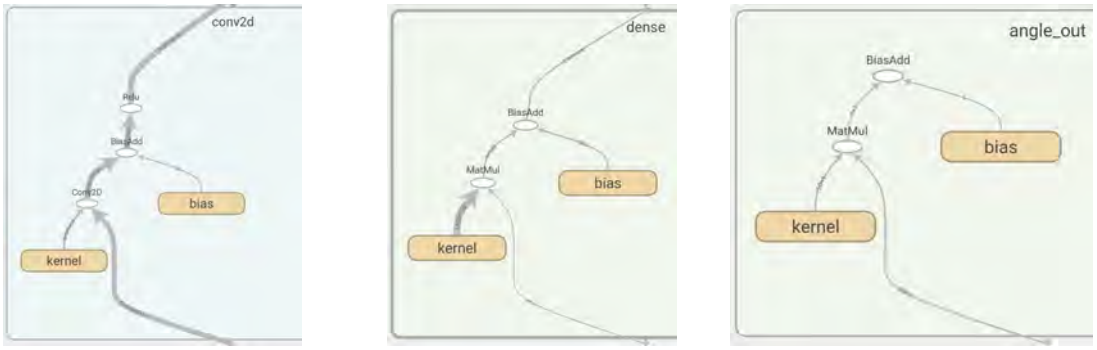


Abbildung 3.4.: Das neuronale Netz, mit dem Donkey Car standardmäßig fährt



(a) Der erste Convolutional Layer (b) Der erste Dense Layer (c) Der angle_out Dense Layer

Abbildung 3.5.: Ansichten aus TensorBoard auf die Inhalte der Layer des Donkey Car Netzwerkes. Dir grau umrandeten weißen Ovale stellen TensorFlow Operationen dar.

Board zu visualisieren.

Das erste Layer im Netz ist in Abbildung 3.5a dargestellt. Aus der Darstellung ist ersichtlich, dass Convolutional Layer in Keras aus mehreren TensorFlow Operationen zusammengesetzt sind. Die eigentliche Convolution ist eine eigene Operation, anschließend wird auf das Ergebnis ein Bias addiert und erst dann wird der Rectifier angewendet. Die Dense Layer sind in Abbildungen 3.5b und 3.5c dargestellt. Diese bestehen aus einer MatMul- (Matrixmultiplikation) und einer BiasAdd-Operation.

Die einzigen Operationen, in denen dieses Netz tatsächlich auf den Tensoren Rechenoperationen durchführt, sind Conv2D, MatMul und BiasAdd. Andere Operationen, wie Relu und Dropout, lesen die Tensoren nur aus und setzen einige Einträge auf 0, verändern aber ansonsten nicht den Inhalt. Auch der Flatten Layer ändert nur die Dimensionalität des Tensors. Daher reicht es zur vollständigen Rechnung mit Approximierung aus, die drei Operationen Conv2D, MatMul und BiasAdd mit Approximation zu implementieren.

3.3. Donkey Simulator

Die Donkey Car Plattform enthält einen Simulator. Dieser basiert auf dem sdsandbox-Projekt von Tawn Kramer [28], welches mit der Unity-Spieleentwicklungsumgebung realisiert wurde. Der Simulator kann Teststrecken zufällig generieren und vordefinierte Teststrecken laden. Ein Screenshot des Simulators befindet sich in Abbildung 3.6.

Der Donkey Simulator erlaubt das Aufzeichnen von Trainingsdaten für ein Donkey Car Netz. Ein mit diesen Daten trainiertes Netz kann mittels eines WebSockets an den Simulator angebunden werden und in diesem fahren. Dazu werden vom Simulator Kamerabilder aus der Perspektive des virtuellen Autos erzeugt und diese über den WebSocket an ein weiteres Programm verschickt, welches das Netz berechnet. Die vom Netz berechnete Ausgabe des Lenkwinkels wird anschließend als Antwort über den WebSocket zurückgesendet. Die Kodierung der WebSocket-Pakete erfolgt im JSON-Format, wobei das Kamerabild als Base64-kodierte Grafik übermittelt wird.

Wichtig in dem Simulator sind die Proportionen und Abmessungen des Autos und der Straße, sowie die verwendeten Einheiten. Der Simulator verwendet bezuglose Maßeinheiten. Durch

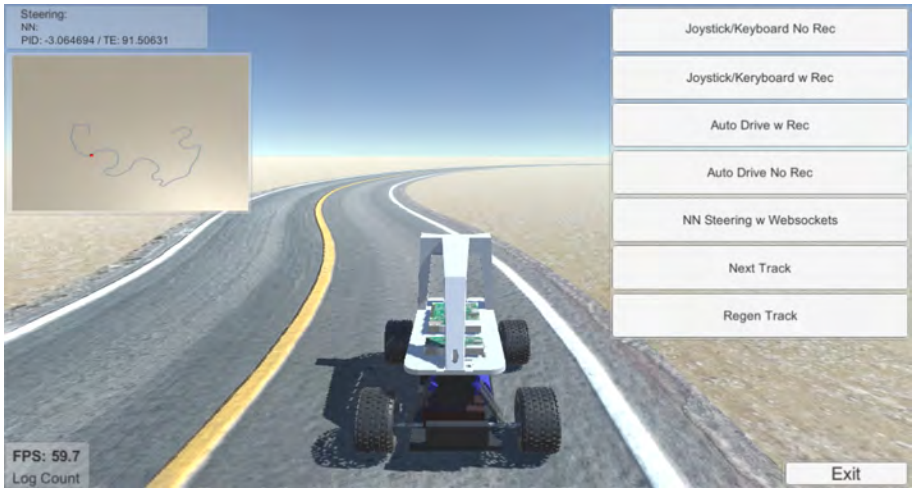


Abbildung 3.6.: Screenshot aus dem Donkey Car Simulator mit einer zufällig generierten Strecke.

Vermessung des 3D-Modells des Autos wurde festgestellt, dass das Auto im Simulator 2,2 Einheiten lang ist, was bei dem Standard-Donkey Car 285 mm entspricht. Pro Einheit ergeben sich damit etwa 130 mm. Die Straße ist 10 Einheiten breit, also 1300 mm.

Part	Beschreibung
`PCA9685`	Steuert den PCA9685-IC von NXP Semiconductors über I²C an. Dieser PWM-Chip steuert den Antrieb und die Lenkung des Autos.
`PiCamera`	Erzeugt Bilder mit der Kamera des Raspberry Pi und gibt diese an andere Parts weiter.
`KerasPilot`	Lädt/erstellt ein neuronales Netzwerk mit der Keras-Bibliothek, wertet die Eingaben aus und gibt die Ergebnisse an andere Parts weiter.
`Tub`	Speichert beliebige Eingabedaten in Dateien ab. Beispielsweise werden zum Training die Bilder der PiCamera und die Benutzereingabe abgespeichert, mit denen später das Netz trainiert wird.
`LocalWebController`	Hostet einen Webserver auf dem Raspberry Pi. Auf diesem wird eine Seite zur Verfügung gestellt, über die der Benutzer das Auto manuell fahren kann und bei Bedarf Trainingsdaten speichern kann.

Tabelle 3.1.: Übersicht über die wichtigsten Donkey Car Parts

Convolutional Layer	Anzahl der Filter	Größe der Filter	Strides	Pooling	Aktivierungsfunktion
Convolution2D_1	24	5×5	2	Nein	Rectifier
Convolution2D_2	32	5×5	2	Nein	Rectifier
Convolution2D_3	64	5×5	2	Nein	Rectifier
Convolution2D_4	64	3×3	2	Nein	Rectifier
Convolution2D_5	64	3×3	1	Nein	Rectifier

Tabelle 3.2.: Konfiguration der Convolutional Layer im Donkey Car CNN

4. Implementierung

Die Implementierung basiert auf TensorFlow 1.12 und Donkey Car 2.5.1. Die drei in dem beschriebenen Netz genannten Operationen wurden als „custom operation“ implementiert. Ein guter technischer Überblick über diese Vorgehensweise ist in der TensorFlow Dokumentation zu finden [30]. Um möglichst kompatibel zu den bereits in TensorFlow existierenden Operationen zu bleiben, wurden diese aus dem TensorFlow Quellcode in ein eigenes Projekt kopiert. Anschließend wurde an den Stellen, an denen die tatsächliche Berechnung stattfindet, die Approximation eingebaut. TensorFlow erlaubt es, zu einer Operation verschiedene Kernel[1] zu implementieren. Diese Funktionalität kann dazu genutzt werden, die Operation auf verschiedenartigen Berechnungsprozessoren auszuführen (CPU/GPU). Die Operationen wurden im Rahmen dieser Arbeit nur für CPUs implementiert.

Da TensorFlow viele der intern von den Operationen benötigte Funktionen nicht über eine öffentliche API bereitstellt, musste viel Code aus TensorFlow kopiert werden. Die Kompatibilität zu eventuellen zukünftigen Versionen der Operationen ist daher möglicherweise eingeschränkt.

Das Projekt mit den eigenen Operationen wurde mit speziellen Kompilerparametern in eine dynamische Programmbibliothek gebaut (unter Linux in `.so`-Dateien), welche dann dynamisch in TensorFlow eingebunden werden.

4.1. ApproxType

In Listing 4.1 wurde zunächst ein Alias für die verwendete Approximationsklasse erzeugt, da mit unterschiedlichen Approximationsgraden approximiert werden sollte. In Zeile 1 wird ein Alias für den ct_float-Typ namens `CtFloatType` erzeugt, in dem die Templateparameter auf die Konstanten `EXPONENT_BITS` und `MANTISSA_BITS` gesetzt werden. Diese werden von außen durch ein Build-Skript gesetzt. Zeile 2 war das Ergebnis des Versuchs, die AriCoFloat-Bibliothek der Arbeitsgruppe Rechnerarchitektur als Datentyp zu verwenden. Diese stellte sich jedoch performancetechnisch als ungünstiger heraus, daher wurde zur Auswertung nur ctfloat genutzt. In Zeile 4 wurde für die gewünschte Approximationsklasse der Alias `ApproxType` zugewiesen, was einen generischen Entwicklungsprozess über alle möglichen approximierten Floattypen ermöglichte.

```
1 typedef ct_float<EXPONENT_BITS, MANTISSA_BITS> CtFloatType;
2 typedef AriCoFloat AriCoType;
3
```

[1] Die Operation ist ein Interface, der Kernel die konkrete Implementierung

```
4 typedef CtFloatType ApproxType;
```

Listing 4.1: Auszug aus ApproxType.h

4.2. Eigen

TensorFlow selbst kann nicht ohne großen Aufwand um neue Datentypen für Tensoren erweitert werden. Intern verwendet TensorFlow die Eigen-Bibliothek für lineare Algebra. Eigen erlaubt im Gegensatz zu TensorFlow sehr einfach die Verwendung eigener Skalartypen und dokumentiert deren Einbau [9]. Für ctfloat reichte es, das Beispiel von Eigen für eigene Datentypen zu kopieren. Dies ist erwartungskonform, da sich der ct_float-Datentyp äußerlich vollständig kompatibel zu einem regulären float verhält. Ein Auszug aus dieser Implementierung ist in Listing 4.2 zu sehen. Relevant sind dort die Zeilen 1-4, in denen im Eigen-Namespace die erforderlichen Traits gesetzt wurden. Diese erben von den Traits des double-Datentyps, der bereits alle erforderlichen Eigenschaften besitzt.

```
1  namespace Eigen {
2      template<>
3      struct NumTraits<CtFloatType>
4              : NumTraits<double> // permits to get the epsilon, dummy_precision,
       lowest, highest functions
5      {
6          typedef CtFloatType Real;
7          CtFloatType NonInteger;
8          typedef CtFloatType Nested;
9          enum {
10             IsComplex = 0,
11             IsInteger = 0,
12             IsSigned = 1,
13             RequireInitialization = 1,
14             ReadCost = 1,
15             AddCost = 3,
16             MulCost = 3
17         };
18     };
19 }
```

Listing 4.2: Auszug aus ApproxType.cpp

4.3. Operationen

4.3.1. Allgemeines

Die eigenen Operationen und Kernel werden in einem C++-Projekt implementiert. Da die Mantissenbits in ctfloat Templateparameter sind, ist es nicht einfach, diese zur Laufzeit anzupassen. Daher wurde ein Skript geschrieben, welches das Projekt mit allen möglichen Mantissenbits

von 2 bis 20 in eine jeweils eigene einzelne Programmbibliothek baut. Dabei werden den Namen der Operationen die Exponenten- und die Mantissenbits angehängt. So heißt die MatMul-Operation, die mit 8 Exponentenbits und 6 Mantissenbits approximiert `ApproxMatMul86`. Dadurch wird die Namenskonvention von TensorFlow beibehalten. Technisch werden die unterschiedlichen Versionen erzeugt, indem das Skript zur Kompilezeit die Konstanten `EXPONENT_BITS` und `MANTISSA_BITS` setzt. Besagtes Skript ist in Listing 4.3 zu sehen. Das Setzen der Variablen erfolgt in Zeile 6. Zur Evaluation wurden alle Programmbibliotheken gleichzeitig in TensorFlow geladen und das Netz auf die jeweils gewünschte Operation abgeändert (Details siehe Abschnitt „Simclient“).

```
1 #!/usr/bin/env bash
2
3 for i in $(seq 2 20); do
4     NAME=build_8_$i
5     cd $NAME
6     cmake -DEXPONENT_BITS=8 -DMANTISSA_BITS=$i ..
7     make -j
8     cd ..
9 done
```

Listing 4.3: Gekürzter Auszug aus dem Buildskript

4.3.2. MatMul

Die MatMul-Operation multipliziert zwei zweidimensionale Eingabetensoren miteinander und bildet das Matrixprodukt. Die Originalimplementierung ruft einen Funktor auf, der wiederum mit Eigen eine Tensor contraction (Tensorverjüngung) durchführt. Eine Tensor contraction ist eine Form der Tensormultiplikation [21].

```
24 namespace tensorflow {
25 namespace functor {
26
27 // Helpers to define tensor<T> needed by MatMul op.
28 template <typename T>
29 struct MatMulTypes {
30   typedef Eigen::TensorMap<Eigen::Tensor<T, 2, Eigen::RowMajor>, Eigen::Aligned
       >
31       out_type;
32   typedef Eigen::TensorMap<Eigen::Tensor<const T, 2, Eigen::RowMajor>,
33                            Eigen::Aligned>
34       in_type;
35 };
36
37 template <typename Device, typename In0, typename In1, typename Out,
38           typename DimPair>
39 void MatMul(const Device& d, Out out, In0 in0, In1 in1,
40             const DimPair& dim_pair) {
41   out.device(d) = in0.contract(in1, dim_pair);
```

```
42 }
43
44 template <typename Device, typename T>
45 struct MatMulFunctor {
46   // Computes on device "d": out = in0 * in1, where * is matrix
47   // multiplication.
48   void operator()(
49       const Device& d, typename MatMulTypes<T>::out_type out,
50       typename MatMulTypes<T>::in_type in0,
51       typename MatMulTypes<T>::in_type in1,
52       const Eigen::array<Eigen::IndexPair<Eigen::DenseIndex>, 1>& dim_pair);
53 };
```

Listing 4.4: TensorFlow Code des MatMul-Funktors [35]

Der Aufruf der Berechnung durch Eigen erfolgt in Zeile 41 des Listings 4.4. `out` ist der Ausgabetensor. Da sich der Tensor prinzipiell auf verschiedenen Berechnungsprozessoren (CPU/GPU) befinden kann, muss die Zuweisung auf dem `device(d)` dieses Tensors ausgeführt werden. `contract` gibt eine `TensorContractionOp` zurück. An dieser Stelle ist die Lazy Evaluation von Eigen zu sehen, die tatsächliche Berechnung wird erst durchgeführt, wenn das Ergebnis benötigt wird, also sobald der `out`-Tensor für weitere Berechnungen wieder ausgelesen wird.

Die eigentliche Implementierung des Kernels für die MatMul-Operation beinhaltet jedoch noch wesentlich mehr. Dort werden verschiedene Spezialfälle behandelt und eine Typunterscheidung durchgeführt, die den Aufruf des Funktors umgehen können [34]. Im Zuge der Implementierung der Approximation wurde dies stark reduziert, da nur der float-Typ behandelt werden muss und um den Funktor in jedem Fall aufzurufen (s. `matmul.cpp`).

Die `Eigen::TensorMap<Eigen::Tensor<T, 2, Eigen::RowMajor>, Eigen::Aligned>` muss zunächst in einen äquivalenten Tensor umgewandelt werden, der anstatt `T` den `ApproxType` als gespeicherten Datentyp beinhaltet. Dazu wurde eine Hilfsfunktion geschrieben, die diese Umwandlung vornimmt.

```
1 template <typename Device, typename T>
2 Eigen::Tensor<ApproxType, 2, Eigen::RowMajor> convertMatrixToApprox(const
      Device &d, T input) {
3     Eigen::Tensor<ApproxType, 2, Eigen::RowMajor> ret;
4     ret.resize(input.dimensions());
5     ret.setZero();
6     ret.device(d) = input.template cast<ApproxType>();
7     return ret;
8 }
```

Listing 4.5: Auszug aus common.h

Die Funktion ist in Listing 4.5 zu sehen. In ihr wird zunächst in Zeile 3 der Rückgabetensor erzeugt. Dieser wird in Zeile 4 auf die Größe des Eingabetensors angepasst. Wichtig ist Zeile 5, dort wird dieser Tensor mit `setZero` auf Null gesetzt, was erforderlich ist, um den tatsächlichen Inhalt des Tensors zu initialisieren. Da `ApproxType` kein primitiver Datentyp sondern eine C++-Klasse ist, ist diese Initialisierung wichtig. Schließlich wird der Eingabetensor in Zeile 6 auf den

ApproxType gecastet und dieser Cast dem Rückgabetensor zugewiesen. Mit dieser Hilfsfunktion kann der Funktor der MatMul-Operation angepasst werden.

```
1  template<typename Device, typename In0, typename In1, typename Out,
2          typename DimPair>
3  void MatMul(const Device &d, Out out, In0 in0, In1 in1,
4              const DimPair &dim_pair) {
5      auto out_approx = convertMatrixToApprox(d, out);
6      auto in_approx_0 = convertMatrixToApprox(d, in0);
7      auto in_approx_1 = convertMatrixToApprox(d, in1);
8
9      out_approx.device(d) = in_approx_0.contract(in_approx_1, dim_pair);
10
11     out.device(d) = out_approx.template cast<float>();
12 }
```

Listing 4.6: Auszug aus matmul.cpp

Der angepasste Funktor ist in Listing 4.6 zu sehen. Die drei in die Funktion eingehenden Tensoren werden in Zeilen 5-7 mit der Hilfsfunktion in approximierte Tensoren umgewandelt. Anschließend wird in Zeile 9 die contract-Funktion analog zur vorherigen Funktion aufgerufen. Schließlich wird der Ausgabetensor in Zeile 9 wieder zurück in ein float gecastet. Dies ist zulässig, da die Umwandlung ohne Informationsverlust stattfindet.

4.3.3. BiasAdd

Die BiasAdd-Operation ähnelt der normalen Additionsoperation für Tensoren, die zwei Tensoren Element für Element addiert. Die Besonderheit liegt darin, dass bei dieser Operation ein Tensor immer eindimensional ist, also einen Vektor darstellt. Dieser wird mittels Broadcasting erweitert und auf den anderen Tensor aufaddiert. Auch diese Operation ruft einen Funktor auf, der die Addition durchführt.

```
27 template <typename Device, typename T, int Dims>
28 struct Bias {
29   // Add "bias" to "input", broadcasting it on all dimensions but the last one.
30   void operator()(const Device& d, typename TTypes<T, Dims>::ConstTensor input,
31                   typename TTypes<T>::ConstVec bias,
32                   typename TTypes<T, Dims>::Tensor output) {
33     if (input.size() >= INT_MAX) {
34       const int64_t bias_size = bias.dimension(0);
35       const int64_t rest_size = input.size() / bias_size;
36       Eigen::DSizes<int64_t, 1> one_d(input.size());
37       Eigen::DSizes<int64_t, 1> bcast(rest_size);
38       output.reshape(one_d).device(d) =
39           input.reshape(one_d) + bias.broadcast(bcast).reshape(one_d);
40     } else {
41       const int bias_size = bias.dimension(0);
42       const int rest_size = input.size() / bias_size;
43       Eigen::DSizes<int, 1> one_d(input.size());
```

```
44        Eigen::DSizes<int, 1> bcast(rest_size);
45        To32Bit(output).reshape(one_d).device(d) =
46            To32Bit(input).reshape(one_d) +
47            To32Bit(bias).broadcast(bcast).reshape(one_d);
48      }
49    }
50  };
```

Listing 4.7: TensorFlow Code des BiasAdd-Funktors [32]

In der Zeile 33 des Listings 4.7 führt dieser Funktor eine Fallunterscheidung für sehr große Tensoren durch. Da diese in dem betrachteten Netz nicht vorkommen wurde nur der andere Fall ab Zeile 41 implementiert.

```
1  template<typename Device, typename T, int Dims>
2  struct Bias {
3      // Add "bias" to "input", broadcasting it on all dimensions but the last
       one.
4      void operator()(const Device &d, typename TTypes<T, Dims>::ConstTensor
       input,
5                      typename TTypes<T>::ConstVec bias,
6                      typename TTypes<T, Dims>::Tensor output) {
7
8          Eigen::Tensor<ApproxType, Dims, Eigen::RowMajor> input_approx;
9          input_approx.resize(input.dimensions());
10         input_approx.setZero();
11         input_approx.device(d) = input.template cast<ApproxType>();
12
13         auto bias_approx = convertVecToApprox(d, bias);
14
15         Eigen::Tensor<ApproxType, Dims, Eigen::RowMajor> output_approx;
16         output_approx.resize(output.dimensions());
17         output_approx.setZero();
18         output_approx.device(d) = output.template cast<ApproxType>();
19
20         Eigen::Tensor<T, Dims, Eigen::RowMajor> output_tensor;
21         output_tensor.resize(output.dimensions());
22         output_tensor.setZero();
23
24         if (input_approx.size() >= INT_MAX) {
25             std::cerr << "Warning: Not approximating large tensor!\n";
26             // [...]
27         } else {
28             const int bias_size = bias_approx.dimension(0);
29             const int rest_size = input_approx.size() / bias_size;
30             Eigen::DSizes<int, 1> one_d(input_approx.size());
31             Eigen::DSizes<int, 1> bcast(rest_size);
32             output_approx.reshape(one_d).device(d) = (input_approx.reshape(
       one_d) + bias_approx.broadcast(bcast).reshape(one_d));
```

```
33              output_tensor.resize(output_approx.dimensions());
34              output_tensor.setZero();
35              output_tensor.reshape(one_d).device(d) = output_approx.reshape(
    one_d).template cast<T>();
36          }
37          output.device(d) = output_tensor;
38      }
39  };
```

Listing 4.8: Auszug aus biasadd.cpp

Der approxoimierende Funktor ist in Listing 4.8 zu sehen. Die Umwandlung in approximierte Input- und Output Tensoren in den Zeilen 8-22 findet an dieser Stelle nicht mit einer Hilfsfunktion statt, da der Templateparameter `Dims` nicht unmittelbar an die Hilfsfunktion übergeben werden kann, ohne eine weitere Templateklasse zu erzeugen. Die Lösung, die Umwandlung ohne Hilfsfunktion durchzuführen, stellt daher einen akzeptablen Kompromiss dar. Ansonsten ist die Vorgehensweise analog zur Implementierung der MatMul-Operation. Der Aufruf von `To32Bit` auf den Tensoren in der Originalfunktion (Listing 4.7, Zeilen 45-47) hat nur den Index der Tensoren auf 32-Bit-Werte geändert und ist an dieser Stelle überflüssig.

4.3.4. Conv2D

Die Conv2D-Operation konnte aufgrund eines nicht näher identifizierten Bugs nicht analog zu den beiden anderen Operationen implementiert werden. Der Bug sorgt dafür, dass der Inhalt des Output-Tensors der Operation nicht-deterministisch wird und nicht mehr dem gewünschten Ergebnis entspricht.

Die Behelfslösung bestand darin, anstatt die gesamte Berechnung mit den approximierten Datentypen durchzuführen, die Input- und Filter-Tensoren zunächst zu approximieren und anschließend wieder in floats umzuwandeln. Dabei ergibt sich bei Conv2D, dass beim inneren Produkt jeweils nicht nur die Multiplikationen stattfinden, sondern auch die Ergebnisse aufsummiert werden (siehe Abschnitt 2.4). Bei dieser Berechnung entsteht mit hoher Wahrscheinlichkeit ein float mit höherer Genauigkeit als bei approximierter Auswertung. Diese Schwäche in der Implementierung wurde bewusst in Kauf genommen, um überhaupt eine approximierende Conv2D-Operation erreichen zu können.

```
54  template <typename Device, typename Input, typename Filter, typename Output>
55  void SpatialConvolutionFunc(const Device& d, Output output, Input input,
56                              Filter filter, int row_stride, int col_stride,
57                              int row_dilation, int col_dilation,
58                              const Eigen::PaddingType& padding) {
59    // Need to swap row/col when calling Eigen.
60    output.device(d) =
61        Eigen::SpatialConvolution(input, filter, col_stride, row_stride, padding,
62                                  col_dilation, row_dilation);
63  }
64
65  template <typename Device, typename T>
66  struct SpatialConvolution {
```

```
67    void operator()(const Device& d, typename TTypes<T, 4>::Tensor output,
68                    typename TTypes<T, 4>::ConstTensor input,
69                    typename TTypes<T, 4>::ConstTensor filter, int row_stride,
70                    int col_stride, int row_dilation, int col_dilation,
71                    const Eigen::PaddingType& padding) {
72      SpatialConvolutionFunc(d, output, input, filter, row_stride, col_stride,
73                             row_dilation, col_dilation, padding);
74    }
75  };
```

Listing 4.9: TensorFlow Code des Conv2D-Funktors [33]

Der Originalcode ist in Listing 4.9 dargestellt. Der Funktor `SpatialConvolution` in Zeile 66 ruft nur die Funktion `SpatialConvolutionFunc` in Zeile 55 auf.

Für die behelfsmäßige Approximation wurde wieder eine Hilfsfunktion geschrieben, die einen normalen Tensor in einen approximierten Tensor umwandelt und anschließend die Umwandlung rückgängig macht.

```
1  template <typename Device, typename T>
2  Eigen::Tensor<float, 4, Eigen::RowMajor> approximateTensor(const Device &d, T &
       input) {
3      Eigen::Tensor<ApproxType, 4, Eigen::RowMajor> temp;
4      temp.resize(input.dimensions());
5      temp.setZero();
6      temp.device(d) = input.template cast<ApproxType>();
7      Eigen::Tensor<float, 4, Eigen::RowMajor> ret;
8      ret.resize(input.dimensions());
9      ret.setZero();
10     ret.device(d) = temp.template cast<float>();
11     return ret;
12 }
```

Listing 4.10: Auszug aus common.h

Die Hilfsfunktion ist in Listing 4.10 zu sehen. Analog zu Listing 4.5 konvertiert sie einen Tensor in den Zeilen 3-6 zunächst in einen Tensor mit approximierten Datentyp. Anschließend wird der Tensor mit der gleichen Methode wieder in einen float-Tensor umgewandelt. Diese Hilfsfunktion wurde dann in den Aufruf des Funktors eingebaut (Listing 4.11).

```
1  template<typename Device, typename T>
2  struct SpatialConvolution {
3      void operator()(const Device &d, typename TTypes<T, 4>::Tensor output,
4                      typename TTypes<T, 4>::ConstTensor input,
5                      typename TTypes<T, 4>::ConstTensor filter, int row_stride,
6                      int col_stride, int row_dilation, int col_dilation,
7                      const Eigen::PaddingType &padding) {
8
9          auto input_approx = approximateTensor(d, input);
10         auto filter_approx = approximateTensor(d, filter);
11
```

```
12          SpatialConvolutionFunc(d, output, input_approx, filter_approx,
        row_stride, col_stride, row_dilation, col_dilation, padding);
13      }
14  };
```

Listing 4.11: Auszug aus conv2d.cpp

Zu beachten ist hierbei, dass der Output-Tensor nicht approximiert wird. Dies ist nicht notwendig, da in jedem Fall der Output einer Conv2D-Operation den Input einer approximierenden Operation ergibt, in der der Input approximiert wird.

4.4. Donkey Simulator

Im Rahmen dieser Arbeit wurde der Donkey Simulator wurde so erweitert, dass er die Abweichung des Autos vom optimalen Pfad in der Mitte der rechten Fahrspur ausgibt. Unity verwendet für 3D-Koordinaten ein linkshändiges Koordinatensystem [29], die Y-Koordinate gibt also die Höhe über dem Boden an.

```
101         public float GetTrackDistance(Vector3 carPosition) {
102                 if(iActiveSpan >= nodes.Count - 3)
103                         return 1000.0f;
104
105                 PathNode a = nodes[iActiveSpan];
106                 PathNode b = nodes[iActiveSpan + 1];
107
108                 carPosition.y = a.pos.y;
109                 LineSeg3d pathSeg = new LineSeg3d(ref a.pos, ref b.pos);
110                 Vector3 errVec = pathSeg.ClosestVectorTo(ref carPosition);
111                 return errVec.magnitude;
112         }
```

Listing 4.12: Funktion zur Rückgabe der aktuellen Pfadabweichung in „Assets/Scripts/CarPath.cs“ (gekürzt)

Die Funktion zur Ermittlung der Pfadabweichung ist in Listing 4.12 zu sehen. Der Pfad besteht aus einer Liste von Punkten, die über Straßensegmente miteinander verbunden sind. In den Zeilen 105 und 106 werden die beiden Punkte ermittelt, zwischen denen sich das Auto gerade befindet. Die Y-Komponente der Position des Autos wird in Zeile 108 ignoriert, da sich der Mittelpunkt des Fahrzeugs durch die Fahrbewegungen leicht auf dieser Achse bewegt, dies aber in der Berechnung der Pfadabweichung nicht berücksichtigt werden soll. Anschließend wird in den Zeilen 109-110 ein Liniensegment zwischen den beiden ermittelten Punkten gezogen und der minimale Abstand des Autos von diesem Lininensegment gebildet. Die Ausgabe dieser Funktion wurde als zusätzliches Feld in dem JSON-Paket, welches über den in Abschnitt 3.3 beschriebenen WebSocket versendet wird, eingefügt.

4.5. Simclient

Um die approximierten Operationen im Donkey Car Simulator evaluieren zu können, wurde ein Client geschrieben, der einen WebSocket Server für den Simulator bereitstellt. Eine Ansicht des Benutzerinterfaces des Simclients ist in Abbildung 4.1 dargestellt. Der Simclient wurde in Python 3 in Verbindung mit PyQt5 für die GUI, NumPy, Keras und TensorFlow für die Neuronalen Netze und Pygame für die Bildverarbeitung implementiert. Er stellt verschiedene Funktionen bereit, die bei der Evaluation der approximierten Operationen helfen. So können in ihm die Anzahl der zu verwendenden Mantissenbits eingestellt werden, die approximierten Operationen einzeln ein- und ausgeschaltet werden sowie ein halbautomatischer Test mit mehreren Durchläufen derselben Konfiguration durchgeführt werden. Die Halbautomatik ergibt sich, da nach jedem Testlauf das Auto im Simulator manuell wieder auf den Startpunkt der Strecke zurückgesetzt werden muss. Zudem kann die Approximation ganz abgeschaltet werden, um Referenzwerte zu bilden.

Der Simclient ist außerdem in der Lage, Testläufe aufzuzeichnen und in Form von gnuplot-Dateien abzuspeichern, die anschließend in Grafiken umgewandelt werden können. Dabei wird zu jedem Simulationsschritt die Abweichung des simulierten Autos von dem Pfad in der Mitte der rechten Fahrspur aufgezeichnet. Ein weiteres Python-Skript kann über die Pfadabweichung dann den Durchschnitt bilden sowie integrieren und erzeugt daraus eine weitere Grafik. Beispiele für diese Grafiken sind in den Abbildungen 4.2 und 4.3 zu sehen. Die erste Grafik zeigt einen Plot der Abweichung des Autos vom Pfad sowie die Geschwindigkeit und die Zielgeschwindigkeit. Die zweite Grafik zeigt die durchschnittliche Pfadabweichung für alle Fahrten eines Testlaufs. Die Basislinie stellt die durchschnittliche Pfadabweichung für Fahrten ohne Approximation dar.

Der wichtigste Codeausschnitt aus dem Simclient ist in Listing 4.13 zu sehen. Dort werden die Operationen aus dem TensorFlow Graphen durch die approximierten Operationen ersetzt. In den Zeilen 1-3 wird zunächst die Graphdefinition aus einer Datei ausgelesen. In Zeile 5 wird über alle Knoten des Graphen iteriert. Falls eine approximierbare Operation gefunden wird und diese ersetzt werden soll, wird der Name der Operation ersetzt (Zeilen 6-11). In den Zeilen 13-17 wird dann der Graph instanziiert und der Ein- und Ausgabetensor aus dem Graphen extrahiert.

```
1  graph_def = graph_pb2.GraphDef()
2  with open(file, "rb") as f:
3      graph_def.ParseFromString(f.read())
4  graph_nodes = graph_def.node
5  for node in graph_nodes:
6      if node.op == "Conv2D" and self.approximateConv2D:
7          node.op = "ApproxConv2D{}{}".format(exponent, mantissa)
8      if node.op == "BiasAdd" and self.approximateBiasAdd:
9          node.op = "ApproxBiasAdd{}{}".format(exponent, mantissa)
10     if node.op == "MatMul" and self.approximateMatMul:
11         node.op = "ApproxMatMul{}{}".format(exponent, mantissa)
12
13 name = "".join(random.choices(string.ascii_uppercase + string.digits, k=16))
14 import_graph_def(graph_def, name=name)
```

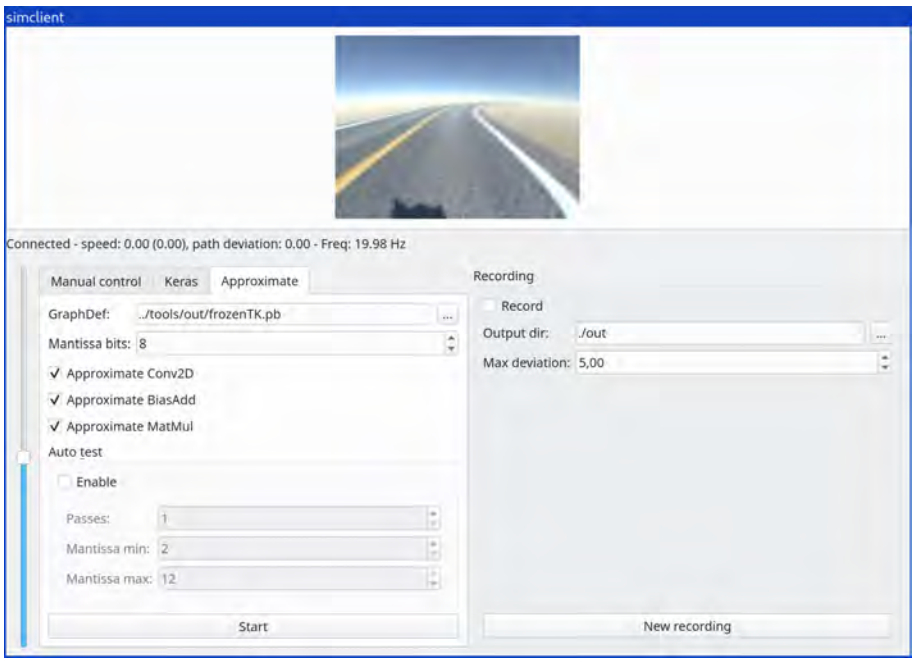


Abbildung 4.1.: Der Simclient im approximierten Fahrmodus (Kamerabild im Simclient vergrößert)

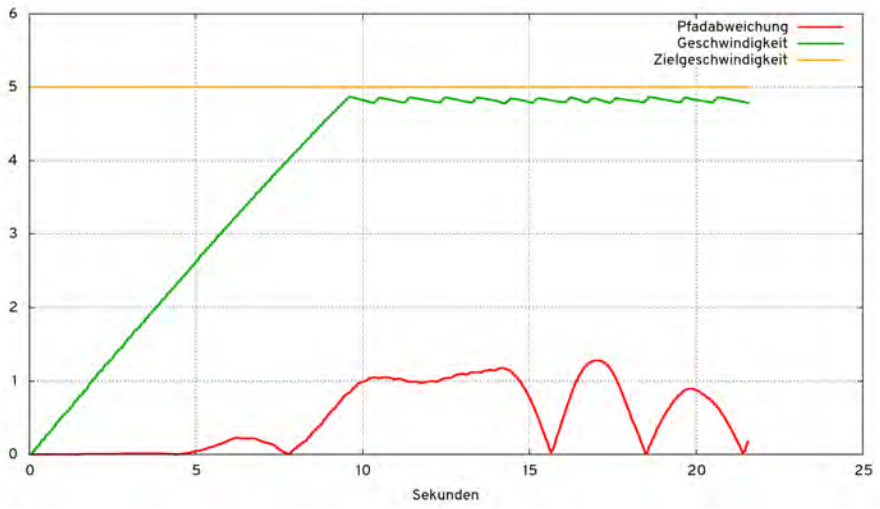


Abbildung 4.2.: Beispiele für die mit dem Simclient erzeugbaren Grafiken für einen einzelnen Durchlauf

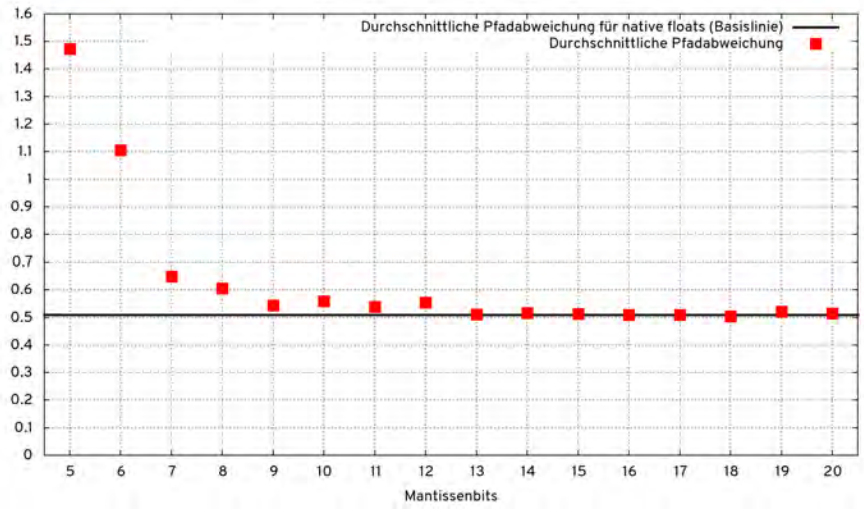


Abbildung 4.3.: Beispiele für die mit dem Simclient erzeugbaren Grafiken für eine Reihe von Durchläufen mit einer unterschiedlichen Anzahl an Mantissenbits

```
15 g = tf.get_default_graph()
16 self.approxImgIn = g.get_tensor_by_name(name + "/img_in:0")
17 self.approxOutput = g.get_tensor_by_name(name + "/angle_out/BiasAdd:0")
```

Listing 4.13: Auszug aus simclient.py

5. Evaluation

5.1. Tests mit dem Simulator

Da die approximierten Berechnungen auf dem Raspberry Pi des umgebauten Modellautos nicht in akzeptabler Geschwindigkeit druchgeführt werden können, wurde die Auswertung im Simulator durchgeführt.

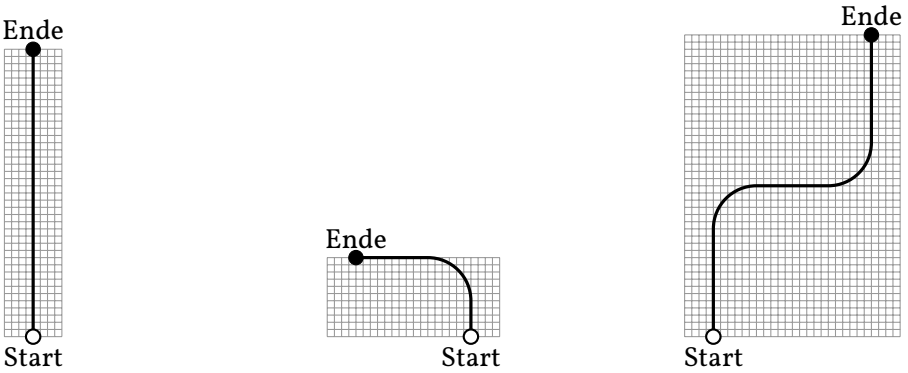
5.1.1. Methodik

Zur Evaluation wurden drei Teststrecken mit aufsteigender Komplexität konstruiert, deren Straßenführung in Abbildung 5.1 zu sehen sind. Teststrecke 1 ist eine einfache gerade Strecke. Auf Teststrecke 2 befindet sich eine einzelne Linkskurve zwischen zwei geraden Abschnitten. Teststrecke 3 enthält ein längeres gerades Stück, eine Rechtskurve, ein weiteres gerades Stück, eine Linkskurve und wieder ein gerades Stück. Die geraden Streckenabschnitte wurden eingebaut um das Ein- und Ausfahrverhalten des Autos in den Kurven beobachten zu können. Das Auto fuhr auf den Strecken jeweils vom Start- zum Endpunkt. Ab einer Pfadabweichung von 5 (65 cm) galt eine Fahrt als nicht bestanden und der Test wurde abgebrochen. Der Wert von 5 wurde gewählt, da die Fahrbahn 10 Einheiten breit ist und bei einer Abweichung von 5 das Auto entweder nach links auf der falschen Fahrspur fährt oder nach rechts ganz von der Fahrbahn abkommt.

Mit Unterstützung der automatisierten Testfunktion des Simclients wurde jede Strecke im Simulator mit verschiedenen Kombinationen von Fahr- und Approximationsparametern getestet. Auf jeder Teststrecke wurde mit zwei unterschiedlichen Zielgeschwindigkeiten (niedrig 2, hoch 5) und Anzahl an Mantissenbits (2-20) getestet. Der Test jeder Kombination wurde dabei zwei Mal durchgeführt und der Durchschnitt gebildet, da das System aus Simulator und Simclient nicht echtzeitfähig ist und somit Unterschiede auch zwischen Testläufen mit gleichen Parametern zu erwarten sind. Das Auto im Simulator fährt während der Berechnung der Ausgabe des Netzes weiter. Die Frequenz, mit der das Netz ausgewertet wurde lag in allen Fällen bei etwa 20 Hz, da dies standardmäßig vom Donkey Simulator als maximale Frequenz festgelegt ist und auch die approximierte Netzausführung auf der verwendeten Hardware mit dieser Frequenz durchgeführt werden konnte.

Die Auswertung wurde auf einem Linux-System mit einem AMD Ryzen Threadripper 1950X 16-Core Prozessor mit 3,4-4 GHz und 32 GB RAM durchgeführt. Die CPU-Auslastung betrug während der Tests je nach Parameterkombination maximal 30%, sodass nicht von auslastungsbedingten Verschlechterungen der Netzperformance durch Verzögerungen bei der Berechnung ausgegangen werden kann.

Zum Training des Netzes wurden zunächst 30.322 Datensätze mit dem Donkey Simulator von zufällig generierten Strecken erzeugt. Dabei wurde die Geschwindigkeit zwischen 2 und 5 variiert. Die Daten wurden anschließend in ein Trainingsset mit 80 % und ein Testset mit 20 % der



(a) Strecke 1 (b) Strecke 2 (c) Strecke 3

Abbildung 5.1.: Das Layout der Teststrecken, auf denen gefahren wurde. Der Maßstab ist für alle Strecken identisch, 1 Kästchen entspricht 1 Simulatoreinheit (etwa 13 cm)

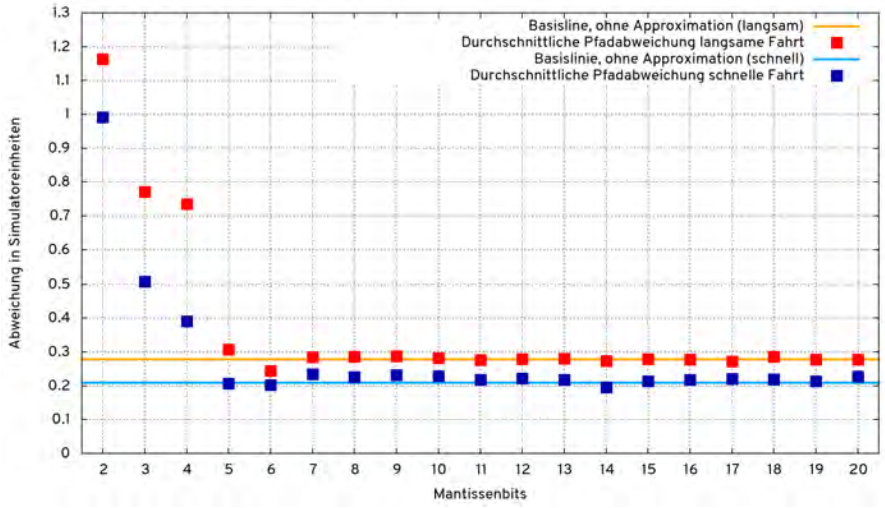


Abbildung 5.2.: Die durchschnittliche Pfadabweichung auf Teststrecke 1 bei beiden untersuchten Geschwindigkeiten

Daten aufgeteilt und ein Netz mit der Standardfunktionalität der Donkey Car Plattform trainiert.

Aus Gründen der Übersichtlichkeit werden die Ergebnisse in den folgenden Abschnitten nur zusammenfassend dargestellt, die vollständigen Tabellen mit den Ergebnissen der Auswertungen befinden sich im Anhang in Abschnitt B.

5.1.2. Teststrecke 1

Auf Teststrecke 1 wurde das Ziel mit allen Parameterkombinationen erreicht. In Abbildung 5.2 ist die durchschnittliche Pfadabweichung für alle betrachteten Approximationsgrade zu sehen. Bei genauerer Betrachtung der Ergebnisse in Abbildung 5.3 fällt jedoch auf, dass die Approximation mit 2 Mantissenbits die Fahrbahn bei längeren Teststrecken verlassen würde. Dies ist in der Abbildung beispielhaft für die langsame Fahrt zu sehen, tritt aber auch bei der schnel-

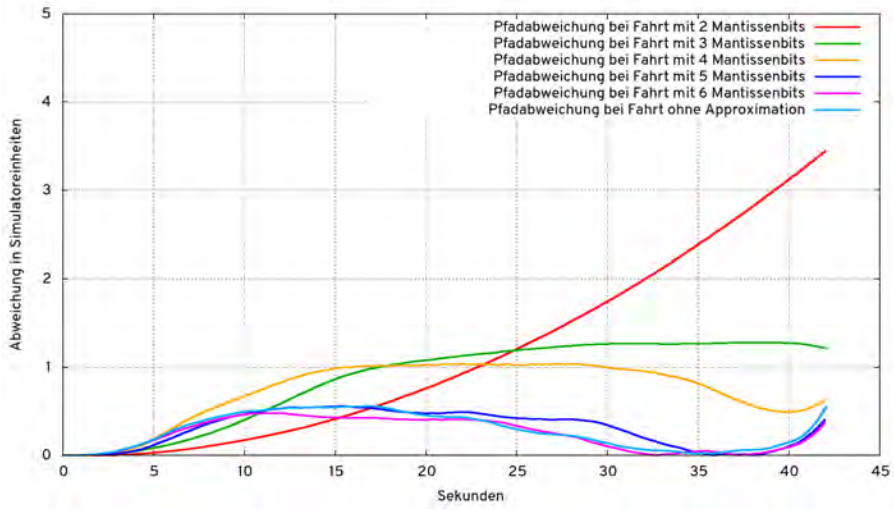


Abbildung 5.3.: Die Pfadabweichung auf Teststrecke 1 bei niedriger Geschwindigkeit mit unterschiedlichen Approximationsgraden

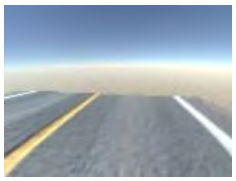


Abbildung 5.4.: Das Ende einer Teststrecke aus der Kameraperspektive des virtuellen Autos

len Fahrt auf. Das trainierte Netz besitzt auf gerader Strecke eine leichte Tendenz, nach links zu fahren, gleicht dies jedoch bereits bei kleiner Abweichung vom Pfad durch Gegenlenken wieder aus. Bei einer geringeren Anzahl an Mantissenbits (3-4) erfolgt dieses Gegenlenken verspätet und mit geringerer Intensität. Am Ende der Strecke (etwa ab Sekunde 38) nimmt die Pfadabweichung bei fast allen Testfahrten wieder zu. Der Grund hierfür ist, dass die Straße aus dem Kamerabild verschwindet (Abbildung 5.4), obwohl das Ziel noch nicht erreicht ist und das Netz ohne Straße stark dazu tendiert, nach links zu lenken. Im Fall mit 4 Mantissenbits verhält sich das Auto zu diesem Zeitpunkt anders, da es im Vergleich zu den anderen Fällen am weitesten vom Pfad entfernt ist und womöglich noch am Nachlenken ist.

Die Ergebnisse für die Fahrt mit hoher Geschwindigkeit sind diesen Ergebnissen sehr ähnlich. Auf gerader Strecke scheint die Geschwindigkeit nicht relevant zu sein. Das leicht bessere Gesamtergebnis bei höherer Geschwindigkeit könnte dadurch zustande kommen, dass für die Fahrt insgesamt weniger Netzberechnungen stattfinden und sich der Fehler somit nicht so stark aufaddiert. Der Unterschied von weniger als 0,1 Simulatoreinheiten bei Approximationen mit mehr als 7 Mantissenbits ist jedoch vernachlässigbar gering.

5.1.3. Teststrecke 2

Auf der zweiten Teststrecke wurde das Ziel nur bei Approximation mit 6 oder mehr Mantissenbits erreicht. Die durchschnittliche Pfadabweichung ist in Abbildung 5.5 zu sehen. Bei weniger als 9 Mantissenbits verschlechtert sich die Pfadabweichung exponentiell. Insgesamt ist die

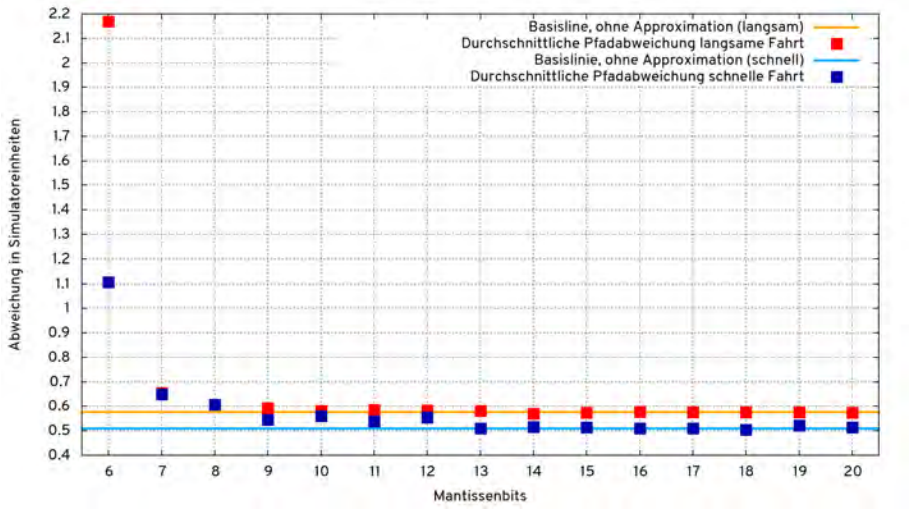


Abbildung 5.5.: Die durchschnittliche Pfadabweichung auf Teststrecke 2 bei beiden untersuchten Geschwindigkeiten

Pfadabweichung bei der Fahrt mit höherer Geschwindigkeit etwas geringer, aber auch hier ist der Unterschied kleiner als 0,1 Simulatoreinheiten. In Abbildung 5.6 ist erkennbar, wieso die Pfadabweichung ab 13 Mantissenbits für die höhere Geschwindigkeit etwas besser ist. In dem mit a) markierten Kreis beginnt das Ende der Kurve. Dort zeigt sich, dass bei der Fahrt mit 13 Mantissenbits das Auto besser nachlenkt. Der Vergleich der Fahrt mit 13 Mantissenbits mit der Fahrt ohne Approximation zeigt aber, dass dies ein zufälliges Ergebnis war und die Fahrt ohne Approximation an der Stelle wieder der Fahrt mit 12 Mantissenbits ähnelt.

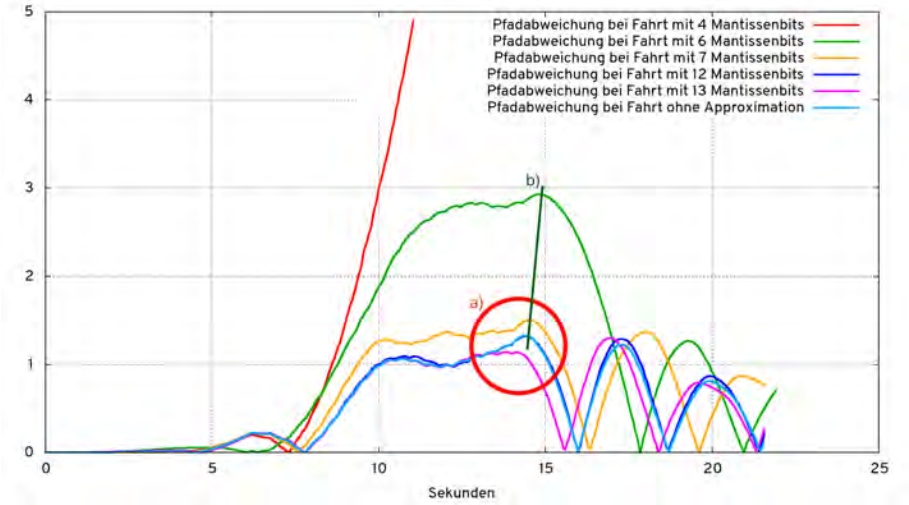


Abbildung 5.6.: Die Pfadabweichung auf Teststrecke 2 bei hoher Geschwindigkeit mit unterschiedlichen Approximationsgraden

In der Abbildung ist weiterhin gut erkennbar, dass während der Kurvenfahrt die Pfadabweichungen mit unterschiedlichen Approximationen einander im Verlauf recht ähnlich sind und nur die Skalierungen der Kurven voneinander abweichen. Besonders gut ist das an den Schnittpunkten der mit b) markierten geraden Strecke mit den Graphen zu sehen. Jeder Schnittpunkt

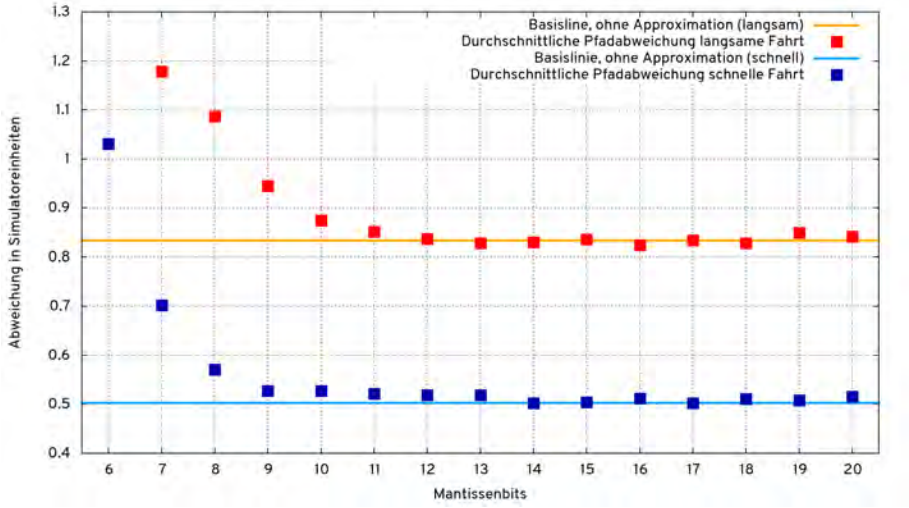


Abbildung 5.7.: Die durchschnittliche Pfadabweichung auf Teststrecke 3 bei beiden untersuchten Geschwindigkeiten

stellt dort für den jeweiligen Graphen ein Maximum dar. Dies legt die Vermutung nahe, dass durch zu starke Approximation das Lenksignal stark gedämpft werden könnte.

5.1.4. Teststrecke 3

Die Ergebnisse auf der dritten Teststrecke zeigen den Einfluss der Approximation auf die durchschnittliche Pfadabweichung (Abbildung 5.7) noch deutlicher als auf der zweiten Teststrecke. Auf dieser Strecke wurde das Ziel bei der Approximation mit 6 Mantissenbits nur bei der Fahrt mit hoher Geschwindigkeit erreicht. Bei weniger als 6 Mantissenbits kam das Auto in allen Fällen bereits in der ersten Kurve von der Fahrbahn ab. In der Abbildung ist eine abnehmende Genauigkeit ab etwa 11 Mantissenbits deutlich zu erkennen, im Falle der langsamen Fahrt ist sie jedoch nicht mehr exponentiell sondern annähernd linear. Auf der Teststrecke ist ab der Approximation mit 14 Mantissenbits keine weitere Verbesserung der Netzperformance zu beobachten.

5.1.5. Allgemeine Beobachtungen

Die Ergebnisse zeigen, dass Approximate Computing mit Mantissenbitreduktion für den Anwendungsfall selbstfahrender Modellautos geeignet ist. In allen Testfällen erzielten Approximationen mit mehr als 14 Mantissenbits keine weiteren Verbesserungen der Performance. In allen Fällen führte die Approximation ab 10 Mantissenbits aufwärts zu durchschnittlichen Pfadabweichungen von maximal 0,1 Simulatoreinheiten beim Vergleich mit der Fahrt ohne Approximation.

Auf allen Teststrecken war ein Unterschied zwischen den Ergebnissen bei langsamer und bei schneller Fahrt erkennbar. Dieser war jedoch auf den ersten beiden Teststrecken mit weniger als 0,1 Simulatoreinheiten sehr gering. Auf der dritten Teststrecke war der Unterschied mit 0,3 Simulatoreinheiten wesentlich größer. Da dies auch bei der Fahrt ohne Approximation auftritt, ist davon auszugehen, dass für das trainierte Netz diese spezielle Kombination von Teststrecke

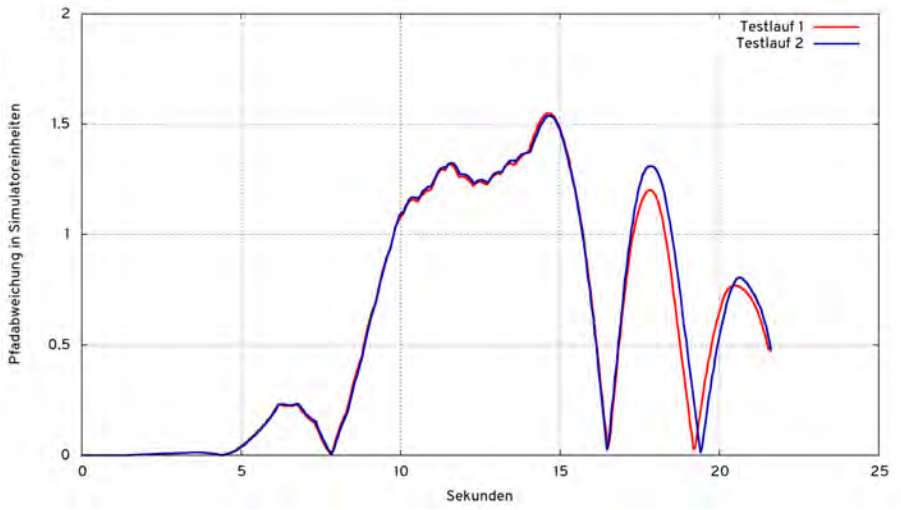


Abbildung 5.8.: Beide Testläufe auf der zweiten Teststrecke mit 8 Mantissenbits bei hoher Geschwindigkeit

(a) Bild 1 (b) Bild 2 (c) Bild 3 (d) Bild 4

Abbildung 5.9.: Die vier für die Einzelbildanalyse verwendeten Eingabebilder

und Geschwindigkeit ungünstig ist.

Beim Auswerten der Ergebnisse fiel weiterhin auf, dass sich die Testläufe mit gleichen Parametern in den meisten Fällen nicht stark unterscheiden, obwohl der Versuchsaufbau wie in der Methodik erläutert zu unterschiedlichen Ergebnissen führen kann. Ein Beispiel, in dem dies besonders gut zutrifft, ist in Abbildung 5.8 zu sehen. Dort ist die durchschnittliche Pfadabweichung der beiden Testläufe auf der zweiten Strecke bei hoher Geschwindigkeit mit 8 Mantissenbits abgebildet. Die beiden Graphen verlaufen für einen Großteil der Strecke nahezu identisch, nur am Ende der Strecke sind leichte Abweichungen zu erkennen.

Die Übertragbarkeit der Ergebnisse auf große selbstfahrende Autos ist nicht ohne Weiteres durchführbar, da die Ergebnisse unter Laborbedingungen im Simulator entstanden sind und zur Steuerung eines selbstfahrenden Autos wesentlich mehr als die Ermittlung des optimalen Lenkwinkels auf einer bestimmten Strecke erforderlich ist. Bei der Verarbeitung von Kameradaten sind weitere Probleme etwa die Erkennung von Verkehrszeichen [24] und von Hindernissen [27] auf der Strecke, die auch einen Einfluss auf den Lenkwinkel haben.

5.2. Einzelbildanalyse

Zusätzlich zu den Tests im Simulator wurde eine Einzelbildanalyse bezüglich des Verhaltens der approximierten Netzausgabe beim Hinzufügen von Bildrauschen durchgeführt. Dazu wurden vier Eingabebilder gewählt, welche verschiedene Sitationen des Autos darstellen können. Die-

(a) Ohne Rauschen (b) Rauschgrad 2 (c) Rauschgrad 5 (d) Rauschgrad 10

Abbildung 5.10.: Beispiel für ein Eingabebild mit unterschiedlichen Rauschgraden

se sind in Abbildung 5.9 dargestellt. Bild 1 und 2 stammen aus dem Simulator und zeigen eine Fahrsituation auf gerader Strecke sowie in einer Rechtskurve, in der sich das Auto bereits fast auf der Mittellinie befindet. Bild 3 ist komplett schwarz und Bild 4 komplett weiß. Die einfarbigen Bilder wurden gewählt, da einerseits bei sehr dunklen Bildern die Kamera des Raspberry Pi starkes Rauschen erzeugt und andererseits die Kamera bei hohem Lichteinfall übersteuert und ein weißes Bild liefert. Keines der gewählten Bilder wurde während des Trainings des Netzes verwendet.

Die Eingabebilder wurden mit Gaußschem Rauschen unterschiedlicher Stärke zwischen 0 und 10 in 0,01er-Schritten überlagert. Dieser Vorgang wurde mit der ImageMagick-Software [17] durchgeführt. Das Rauschen selbst war dabei für jedes Bild einer Serie identisch und wurde nur mit unterschiedlichen Intensitäten [18] überlagert. In Abbildung 5.10 sind einige Beispiele für eins der Eingabebilder mit unterschiedlichen Rauschgraden abgebildet.

```
1 #!/usr/bin/env bash
2
3 export LC_ALL=C
4 for i in $(seq 0 0.01 10); do
5     OCL_ICD_VENDORS=mesa convert -seed 1000 -attenuate $i input_1.png +noise
      gaussian noise/out_${i}.png
6 done
```

Listing 5.1: Das Skript zur Generierung der verrauschten Bilder

Das Bash-Skript zur Erzeugung der verrauschten Bilder ist in Listing 5.1 aufgeführt. In Zeile 3 wird zunächst die Ausgabesprache aller im Folgenden aufgerufenen Programme auf einen einheitlichen Wert geändert, da es ansonsten bei der Generierung der Fließkommazahlen zu Problemen mit dem Dezimalpunkt kommt. In Zeile 4 beginnt die Schleife, die die Variable `i` hochzählt. Die Addition des Rauschens erfolgt in Zeile 5. Dort wird das `convert`-Programm von ImageMagick aufgerufen, welches dem Eingabebild dann Rauschen mit dem Grad in `i` hinzufügt und das Ausgabebild in einem Ausgabeordner abspeichert. Für die Bilderreihen wurden unterschiedliche Seeds gewählt.

In das Netz, welches auch für die Tests mit dem Simulator verwendet wurde, wurden anschließend alle Bilder eingegeben und die Netzausgabe berechnet. Ein Ausschnitt aus dem Ergebnis dieses Vorgehens ist in Abbildung 5.11 zu sehen. Dort ist für das erste Bild der Einzelbildanalyse die Netzausgabe zu einigen Approximationsgraden aufgezeichnet. Für die Berechnungen mit 2 und 4 Mantissenbits, die sich in den Simulatortests als zu stark approximiert erwiesen haben, ist gut zu erkennen, dass die Auflösung der möglichen Ausgabewerte stark abgenommen hat.

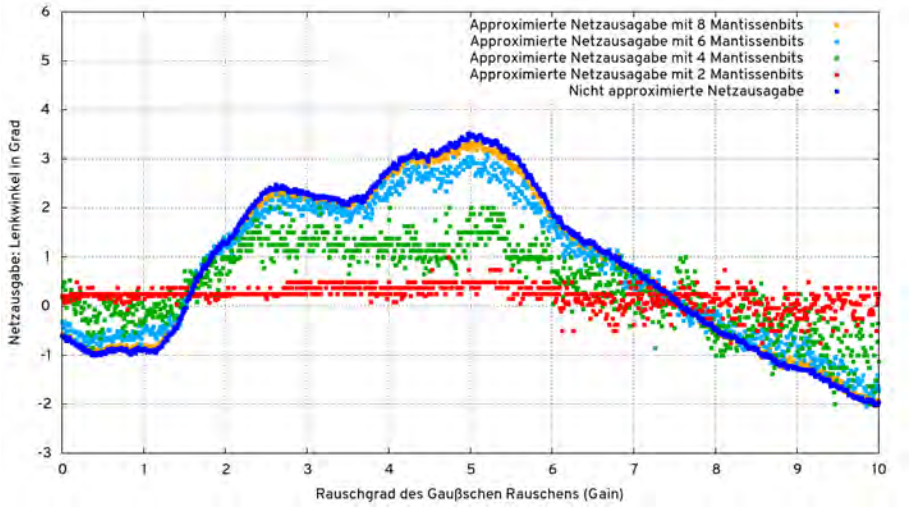


Abbildung 5.11.: Die Netzausgaben für Eingabebild 1

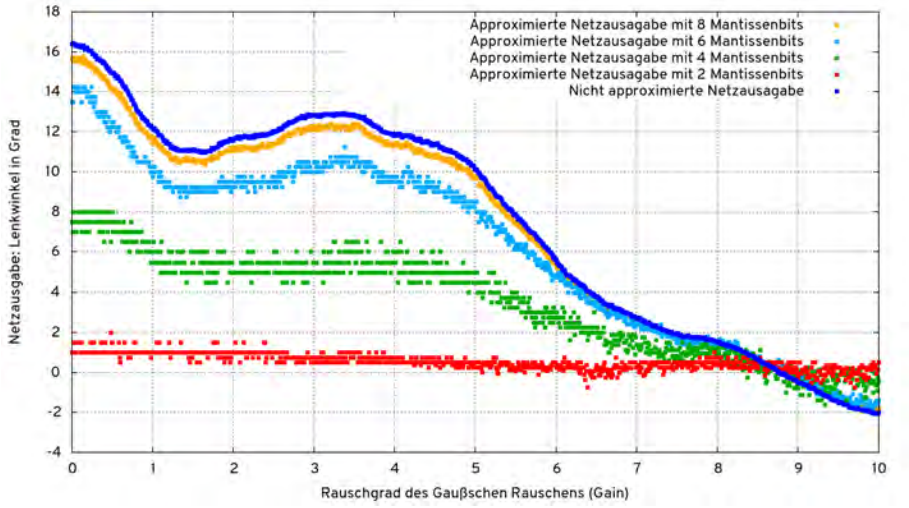


Abbildung 5.12.: Die Netzausgaben für Eingabebild 2

Für Approximationen mit mehr Mantissenbits nähert sich die Netzausgabe immer weiter der nicht-approximierten Netzausgabe an und die Abweichungen benachbarter Werte nimmt ab. Dieselben Beobachtungen sind in der Auswertung mit dem zweiten Bild zu machen (Abbildung 5.12).

Die Ergebnisse für die Bilder 3 und 4 sind in den Abbildungen 5.13 und 5.14 zu sehen. Die Beobachtungen der vorherigen Bilder sind auch auf diese Bilder, mit denen das Netz nicht trainiert wurde, übertragbar. Zudem lässt sich aus den Abbildungen ableiten, dass sich durch die approximierte Berechnung das Netz auch bei unerwarteter Eingabe ähnlich zur exakten Berechnung verhält.

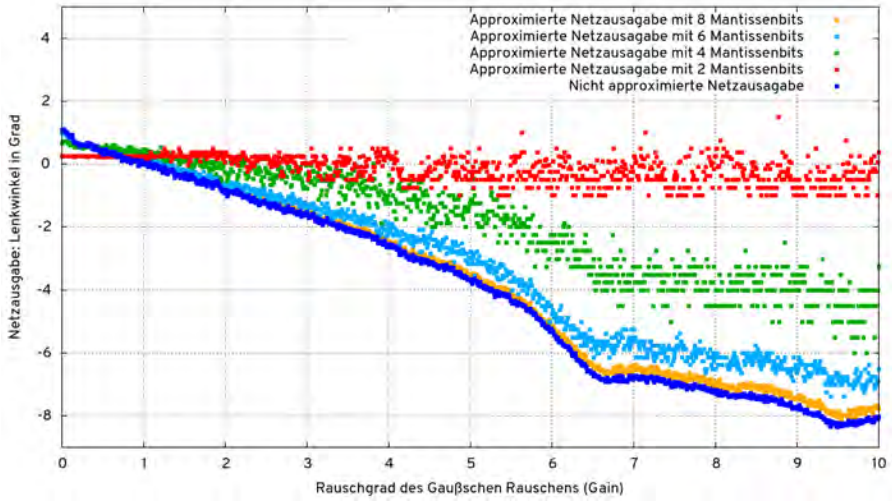


Abbildung 5.13.: Die Netzausgaben für Eingabebild 3

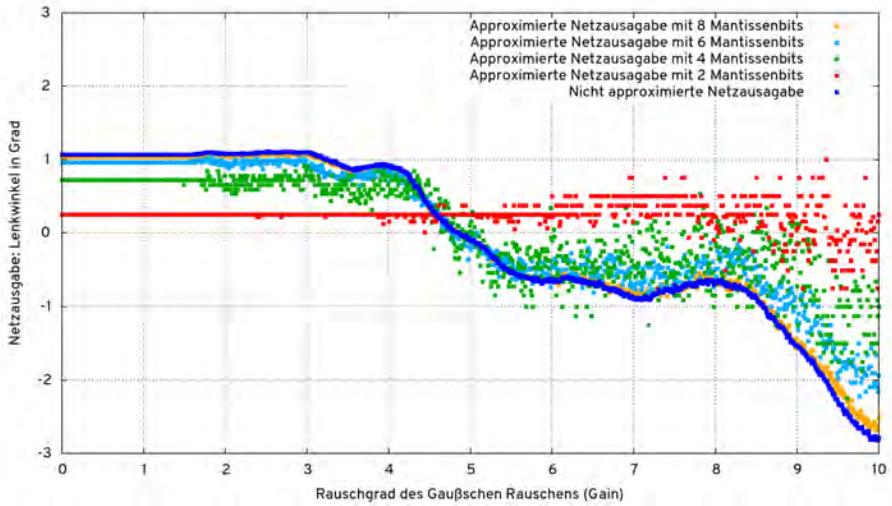


Abbildung 5.14.: Die Netzausgaben für Eingabebild 4

6. Schlussbemerkungen

6.1. Zusammenfassung

In dieser Abschlussarbeit wurde approximierte Berechnung in drei TensorFlow-Operationen implementiert und anschließend deren Performance bei der Lenkung eines virtuellen Modellautos bei verschiedenen Approximationsgraden evaluiert. Dies fand auf Basis der Donkey Car Plattform statt. Im Rahmen der Abschlussarbeit entstand zudem ein Tool, welches die approximierte Berechnung eines CNNs durchführt und an den Donkey Simulator angebunden werden kann.

Mithilfe der entwickelten Implementierung ist es möglich, auch weitere KNNs in TensorFlow mit einer unterschiedlichen Anzahl an Exponenten- und Mantissenbits in den genannten Operationen approximiert zu berechnen.

Die aus der Arbeit hervorgehenden Ergebnisse zeigen, dass Approximate Computing für die Berechnungen von CNNs geeignet ist und viele Vorteile besitzen. Je nach Anwendungsfall gibt es Approximationsgrade, ab denen sich das Ergebnis nicht mehr signifikant verbessert.

6.2. Ausblick

Auf Basis dieser Arbeit ergeben sich weitere Forschungsfragen. Die Übertragbarkeit der Ergebnisse auf große selbstfahrende Autos ist ein wichtiger Punkt. Dabei ist die Ermittlung eines akzeptablen Approximationsgrades auch in komplexeren Fahrsituationen relevant. Denkbar wäre beispielsweise eine Umschaltung des Approximationsgrades je nach Fahrsituation.

In dem betrachteten Netz erfolgt die Ausgabe des Lenkwinkels kontinuierlich, kann also jeden Wert annehmen. In vielen Anwendungsfällen ist es günstiger, den Ausgabebereich zu diskretisieren und mittels One-Hot Encoding einen Wert auszuwählen. Ein Vergleich dieser beiden Ansätze bei approximierter Berechnung wäre auch denkbar.

Ein wichtiger Punkt ist die Approximation bereits im Training durchzuführen und die daraus resultierenden Auswirkungen zu untersuchen. Eventuell eignet sich Approximate Computing mit weniger Bits als weitere Methode zur Reduktion von Overfitting.

6.3. Danksagungen

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Anfertigung der Bachelorarbeit direkt oder indirekt unterstützt haben. Besonderer Dank gilt der Arbeitsgruppe Rechnerarchitektur im Fachbereich 3, speziell Prof. Dr. Rolf Drechsler, der diese Arbeit vermittelt hat und der sich bereit erklärt hat, als Erstprüfer aufzutreten, Dr. Felix Putze, der die Aufgabe des Zweitprüfers wahrgenommen hat, und Saman Fröhlich, der die Arbeit direkt betreut hat.

Zudem danke ich der INRIA, dem nationalen französischen Institut für Informatik und angewandte Mathematik, speziell den Personen Romain Mercier und Olivier Sentieys, für die Bereitstellung der `ct_float`-Softwarebibliothek.

Besonderer Dank gilt natürlich auch allen Autoren, auf deren Arbeiten ich aufbauen durfte (siehe Literaturverzeichnis) sowie den Autoren der freien Software, ohne die diese Arbeit nicht möglich gewesen wäre. Beispielhaft wären da folgende Projekte zu nennen: Donkey Car, Linux, Git, OpenSCAD, KiCad EDA, XeLaTeX, GCC, Python, Keras, TensorFlow, gnuplot, ImageMagick und die Vollkorn-, Roboto-, Asana Math- und Fantasque Sans-Schriftfamilien.

Abschließend danke ich meiner Familie und ganz besonders meinen Eltern, die mich bei meinem gesamten Studium unterstützt haben.

Literatur

[1] Martín Abadi u. a. TensorFlow: A system for large-scale machine learning. In: arXiv e-prints, arXiv:1605.08695 (2016-05). URL: https://arxiv.org/abs/1605.08695.

[2] Mariusz Bojarski u. a. End to End Learning for Self-Driving Cars. In: arXiv e-prints, arXiv:1604.07316 (2016-04). URL: https://arxiv.org/abs/1604.07316.

[3] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. STD 90. 2017-12. URL: https://tools.ietf.org/html/rfc8259.

[4] Xiaogang Chen u. a. Computer Vision - ACCV 2014 Workshops : Singapore, Singapore, November 1-2, 2014, Revised Selected Papers, Part I : Pedestrian Detection with Deep Convolutional Neural Network. In: Lecture Notes in Computer Science 9008 (2015), S. 354–365. ISSN: 9783319166278. DOI: 10.1007/978-3-319-16628-5.

[5] ctfloat Bibliothek. 2018. URL: https://gitlab.inria.fr/sentieys/ctfloat (besucht am 12. 02. 2019).

[6] Donkey Car - Home. 2018. URL: http://www.donkeycar.com/ (besucht am 17. 10. 2018).

[7] Donkey Car Slack: Angaben über das verwendete Netz (Registrierung erforderlich). 2019. URL: https://donkeycar.slack.com/archives/C9Y06JXV3/p1540212514000100 (besucht am 02. 02. 2019).

[8] Donkey Car Slack: Angaben über verwendete Rechner (Registrierung erforderlich). 2018. URL: https://donkeycar.slack.com/archives/C487786DC/p1529540990000073 (besucht am 21. 07. 2018).

[9] Eigen: Using custom scalar types. 2019-01. URL: http://eigen.tuxfamily.org/dox/TopicCustomizing_CustomScalar.html (besucht am 12. 02. 2019).

[10] Example: SWAN wave model. 2009. URL: http://www.texample.net/tikz/examples/swan-wave-model/ (besucht am 12. 02. 2019).

[11] I. Fette und A. Melnikov. The WebSocket Protocol. RFC 6455. 2011-12. URL: https://tools.ietf.org/html/rfc6455.

[12] IEEE Standard for Floating-Point Arithmetic. In: IEEE Std 754-2008 (2008-08), S. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[13] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648. 2006-10. URL: https://tools.ietf.org/html/rfc4648.

[14] Keras Documentation. 2019. URL: https://keras.io/ (besucht am 10. 02. 2019).

[15] Alex Krizhevsky, Ilya Sutskever und Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, S. 2012.

[16] Rudolf Kruse u. a. Computational Intelligence: eine methodische Einführung in künstliche neuronale Netze, evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze. 2., überarbeitete und erweiterte Auflage. Computational Intelligence. Wiesbaden: Springer Vieweg, 2015. ISBN: 978-3-658-10903-5. DOI: 10.1007/978-3-658-10904-2.

[17] ImageMagick Studio LCC. ImageMagick Website. 2019. URL: http://www.imagemagick.org/ (besucht am 18. 02. 2019).

[18] ImageMagick Studio LCC. ImageMagick: Attenuate option. 2019. URL: http://www.imagemagick.org/script/command-line-options.php#attenuate (besucht am 18. 02. 2019).

[19] Yann LeCun u. a. Backpropagation applied to handwritten zip code recognition. In: Neural Computation 1 (1989), S. 541–551. URL: http://yann.lecun.org/exdb/publis/pdf/lecun-89e.pdf.

[20] Yann LeCun u. a. Gradient-Based Learning Applied to Document Recognition. In: Proceedings of the IEEE 86 (1998-12), S. 2278–2324. DOI: 10.1109/5.726791.

[21] W. Ma u. a. Acceleration of Streamed Tensor Contraction Expressions on GPGPU-Based Clusters. In: 2010 IEEE International Conference on Cluster Computing. 2010-09, S. 207–216. DOI: 10.1109/CLUSTER.2010.26.

[22] Chris J. Maddison u. a. Move Evaluation in Go Using Deep Convolutional Neural Networks. In: arXiv e-prints, arXiv:1412.6564 (2014-12). URL: https://arxiv.org/abs/1412.6564.

[23] Tomas Mikolov u. a. Efficient Estimation of Word Representations in Vector Space. In: arXiv e-prints, arXiv:1301.3781 (2013-01). URL: https://arxiv.org/abs/1301.3781.

[24] Andreas Møgelmose, Mohan M. Trivedi und Thomas B. Moeslund. Traffic Sign Detection and Analysis: Recent Studies and Emerging Trends. In: In 15th IEEE International Conference on Intelligent Transportation Systems. 2012.

[25] Srivastava Nitish u. a. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In: Journal of Machine Learning Research 15 (2014), S. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[26] python-conv2d: 2D image convolution example in Python. 2016-11. URL: https://github.com/sunsided/python-conv2d/ (besucht am 05. 12. 2018).

[27] Sebastian Ramos u. a. Detecting Unexpected Obstacles for Self-Driving Cars: Fusing Deep Learning and Geometric Modeling. In: arXiv e-prints, arXiv:1612.06573 (2016-12). URL: https://arxiv.org/abs/1612.06573.

[28] sdsandbox: donkey Branch. 2019-01. URL: https://github.com/tawnkramer/sdsandbox/tree/donkey (besucht am 12. 02. 2019).

[29] Unity Technologies. Unity Manual: Transformations. 2019. URL: https://docs.unity3d.com/Manual/Transforms.html (besucht am 18. 02. 2019).

[30] TensorFlow: Adding a New Op. 2019. URL: https://www.tensorflow.org/guide/extend/op (besucht am 12. 02. 2019).

[31] TensorFlow: Broadcasting semantics. 2019. URL: https://www.tensorflow.org/xla/broadcasting (besucht am 16. 01. 2019).

[32] tensorflow/bias_op.h at r1.12. 2018. URL: https://github.com/tensorflow/tensorflow/blob/r1.12/tensorflow/core/kernels/bias_op.h (besucht am 12. 02. 2019).

[33] tensorflow/conv_2d.h at r1.12. 2018. URL: https://github.com/tensorflow/tensorflow/blob/r1.12/tensorflow/core/kernels/conv_2d.h (besucht am 12. 02. 2019).

[34] tensorflow/matmul_op.cc at r1.12. 2018. URL: https://github.com/tensorflow/tensorflow/blob/r1.12/tensorflow/core/kernels/matmul_op.cc (besucht am 12. 02. 2019).

[35] tensorflow/matmul_op.h at r1.12. 2018. URL: https://github.com/tensorflow/tensorflow/blob/r1.12/tensorflow/core/kernels/matmul_op.h (besucht am 12. 02. 2019).

[36] Thingiverse: Donkey Self-Racing Car Referenzdesign. 2018-03. URL: https://www.thingiverse.com/thing:2260575 (besucht am 17. 10. 2018).

[37] J. Y. F. Tong, D. Nagle und R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems 8.3 (2000-06), S. 273–286. ISSN: 1063-8210. DOI: 10.1109/92.845894.

[38] C. Torres-Huitzil und B. Girau. Fault and Error Tolerance in Neural Networks: A Review. In: IEEE Access 5 (2017), S. 17322–17341. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2742698.

[39] XciteRC Eagle Brushed 1:16 RC Modellauto Elektro Monstertruck Heckantrieb 100 RtR 2,4 GHz. 2018. URL: https://www.conrad.de/de/xciterc-eagle-brushed-116-rc-modellauto-elektro-monstertruck-heckantrieb-100-rtr-24-ghz-1298994.html (besucht am 17. 11. 2018).

A. Umbauanleitung Donkey Car

A.1. Zusätzlich benötigte Materialien

- Ein ESC (Fahrtenregler, Motorsteuerung) für brushed-Motoren mit Steuerung über ein Standard Servo-Signal
- Ein Servo für die Lenkung
- Ein 3D-Ausdruck der Kamerahalterung, Modell aus Abschnitt A.7 einfach mit OpenSCAD öffnen, auf Render klicken und das STL exportieren
- Eine Befestigungsplatte für den Raspberry Pi, den Servocontroller, die Kamerahalterung und die Stromversorgung
- Metallklammern zum Festklemmen der Kamerahalterung

Falls zusätzlich noch der Akku durch zwei 18650-Zellen ausgetauscht werden soll (optional):

- 18650-Akkuhalter
- 2 18650 Akkus, Empfehlung: Nicht Li-Po, sondern $LiFePo_4$, da sicherer
- Ladegerät für 18650 $LiFePo_4$ Akkus
- LCD Voltmeter zur Überwachung
- Sicherungen, zum Beispiel einen Sicherungshalter für 12 V-Autosicherungen und eine 5 A-Sicherung

Immer nützlich:

- WAGO Kabelklemmen der Serie 221 (2er und 3er)
- Wiederverwendbare und normale Kabelbinder
- Doppelseitiges Klebeband

A.1.1. Wichtige Hinweise

Vor eigenen Experimenten sollten diese wichtigen Hinweise beachtet werden:

1. Niemals den Servo alleine an die Servosteuerung anschließen! Nur zusammen mit dem ESC. Der ESC erzeugt intern die korrekte Spannungsversorgung für den Servo und gibt sie über sein eigenes Servokabel zurück. Falls man nur den Servo alleine anschließt zieht dieser seinen ganzen Strom aus dem Raspberry Pi, wofür dieser nicht ausgelegt ist.

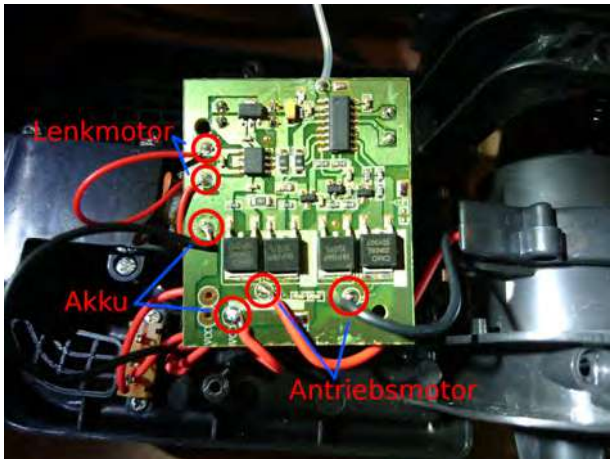


Abbildung A.1.: Die originale Motorsteuerung

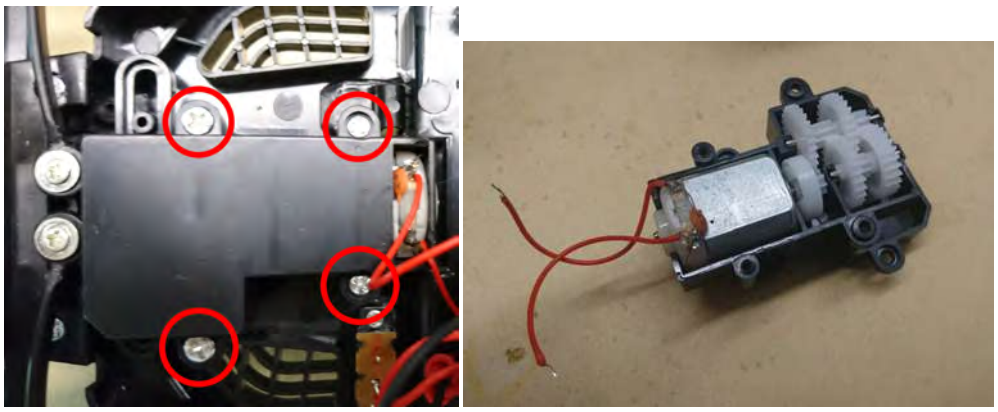


Abbildung A.2.: Die Lenkungssteuerung

2. Empfohlen werden Lithium-Eisenphosphat-Akkus ($LiFePo_4$), da diese deutlich sicherer sind als Lithium-Ionen-Akkus. Eine Spannungsüberwachung ist empfohlen, um die Akkus nicht tiefzuentladen. Eine Kurzschlusssicherung ist auch sehr sinnvoll.

A.2. Umbau

A.2.1. Auto

A.2.1.1. Fahrtelektrik

1. Die Verkleidung abnehmen. Die Verkleidung für die Original-Motorsteuerung abschrauben.
2. Alle Kabel von der Original-Motorsteuerung entlöten (siehe Abbildung A.1).
3. Die Original-Lenksteuerung abschrauben (siehe Abbildung A.2).
4. Das Auto sollte nun leer aussehen (siehe Abbildung A.3).

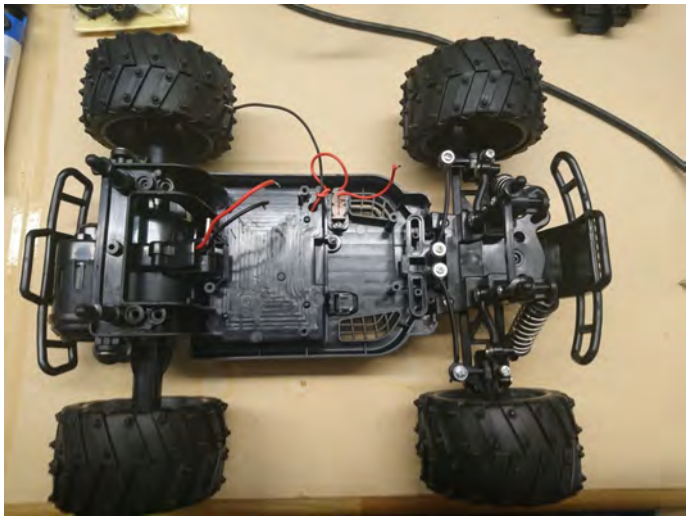


Abbildung A.3.: Das Auto ohne Steuerung

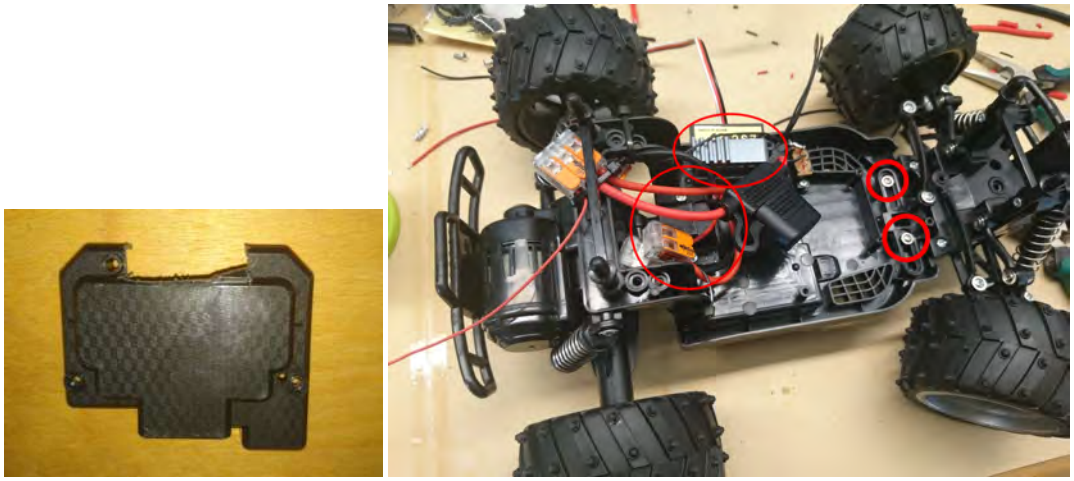


Abbildung A.4.: Abdeckung und ESC-Einbau

5. Von der Originalabdeckung ein Teil absägen und wieder festschrauben, sodass der Antriebsmotor fixiert ist. Den ESC an eine sinnvolle Stelle kleben und die beiden dicken Kabel farbrichtig mit dem Motor verbinden (Kabelklemmen, ansonsten verlöten und isolieren). Zusätzlich vorne an die angegebene Stelle Schrauben drehen. Es eignen sich die Schrauben, mit denen das Auto im Karton befestigt war (siehe Abbildung A.4).

6. Den Servo erst einbauen wenn die Position bestimmt ist! Also erst wenn das Auto fast fertig ist und die Software schon auf dem Raspberry Pi läuft.

7. Den Stromkreis für die Fahrelektronik zusammenbauen: Wenn nötig den Original-Batteriestecker entfernen. Mit dem Schalter die Masse schalten, von dort

   a) In den ESC und von dort zum Pluspol

   b) In das Voltmeter falls benötigt und zum Pluspol

   c) Ein weiteres Kabel für die Masseverbindung zum RasPi, wenn man direkt stecken

Abbildung A.5.: Stoßdämpfer, links original und rechts gekürzt

will einen Standard DuPont-Stecker crimpen

(Details siehe Schaltplan in Abschnitt A.6)

A.2.1.2. Stoßdämpfer vorne

Die vorderen Stoßdämpfer waren mit der schweren Befestigungsplatte überfordert. Daher wurden sie ausgebaut und mit Kabelbindern der verkürzt, was sie effektiv härter gemacht hat (siehe A.5). Der erste Versuch, die Federn selbst auszutauschen hat nicht funktioniert, da die Stoßdämpfer nicht zerstörungsfrei auseinanderzubauen waren.

A.2.1.3. Befestigungsplatte

Auf der Befestigungsplatte werden der Raspberry Pi, der Servo-Controller, die Kamera und die Powerbank montiert. Die originalen, vom Donkey Car Projekt verwendeten 3D-druckbaren Teile des Referenzdesigns [36] wurden nicht verwendet, da diese nicht mit dem verwendeten Modellauto kompatibel waren und der Druck selbst technisch schwierig war.

1. Die Platte vorbereiten: Komplett designen und ausdrucken oder Löcher in eine bereits vorhandene Platte bohren.
2. Die Kamerahalterung drucken
3. Die einzelnen Komponenten festschrauben. Empfehlung: Die langen Schrauben durchstecken, 2 Muttern darauf, durchstecken und mit einer weiteren Mutter festmachen. Die Komponenten korrekt verbinden (siehe auch Schaltplan in Abschnitt A.6).
4. Die Platte auf das Auto aufsetzen und die beiden Stecker für Masse und ESC einstecken.

A.3. Inbetriebnahme und letzte Schritte beim Zusammenbau

1. Den Raspberry Pi im Folgenden noch **nicht einschalten**. Den Servo an Port 1 des Servocontrollers anschließen. Den Schalter an der Unterseite des Autos betätigen. Der ESC

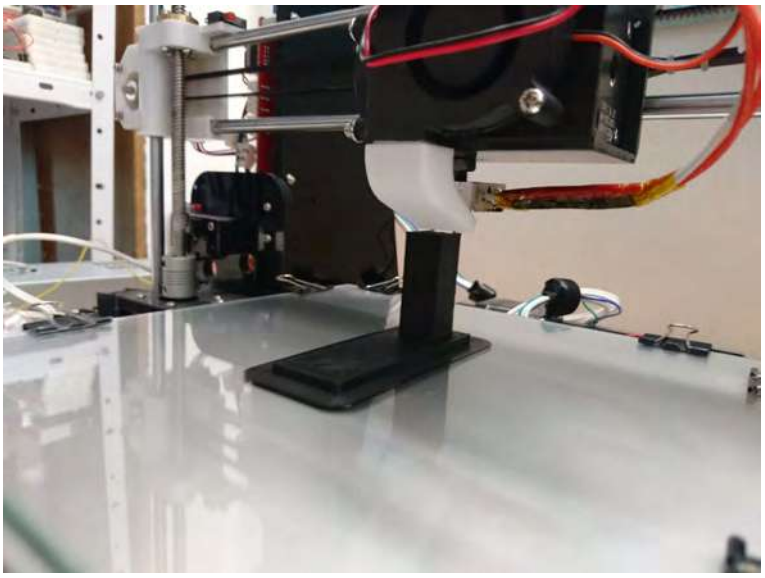


Abbildung A.6.: 3D Drucker beim Druck des ersten Prototyps der Kamerahalterung

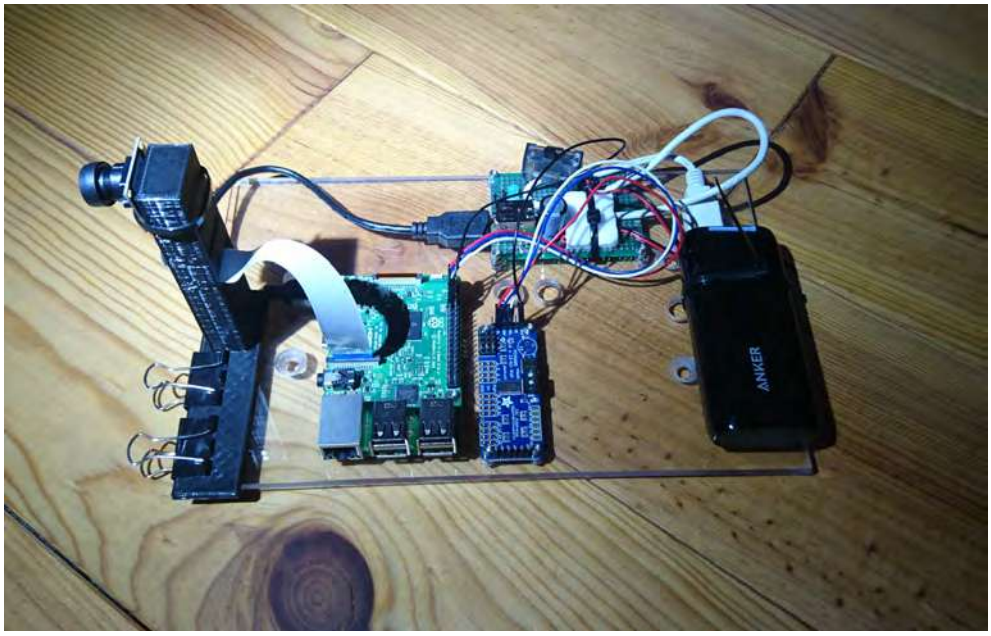


Abbildung A.7.: Die Besfestigungsplatte mit den einzelnen Komponenten

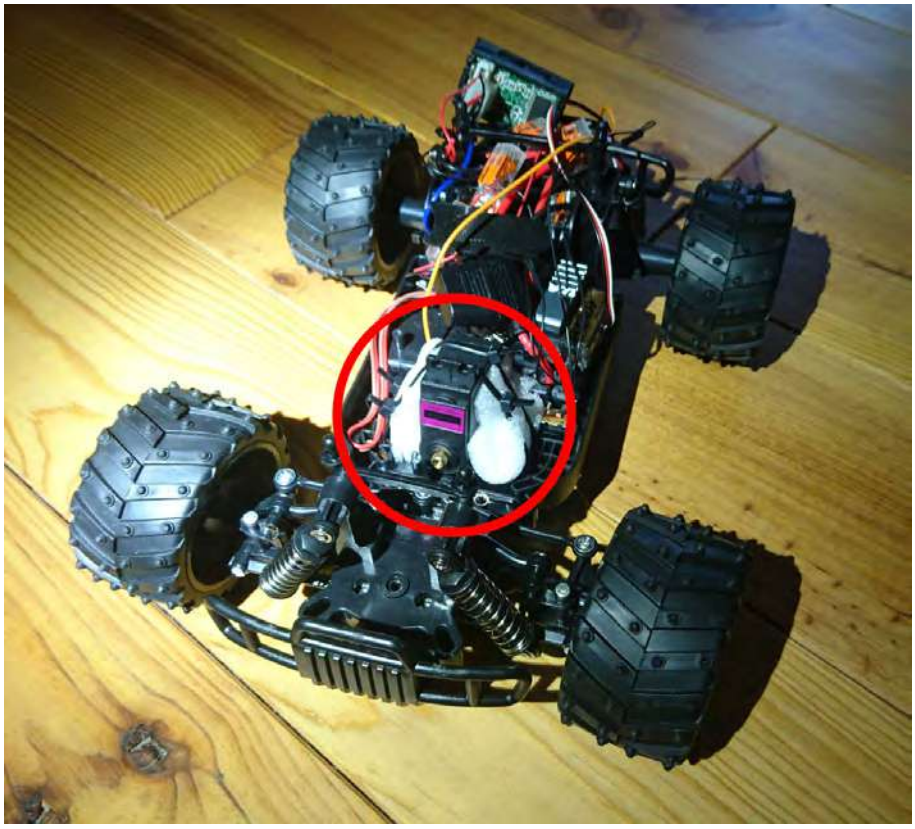


Abbildung A.8.: Die Befestigung des Servos

sollte piepsen und der Servo sollte sich ein bisschen drehen und dann anhalten. Wieder ausschalten und den Vorgang wiederholen, bis sich der Servo nicht mehr dreht.

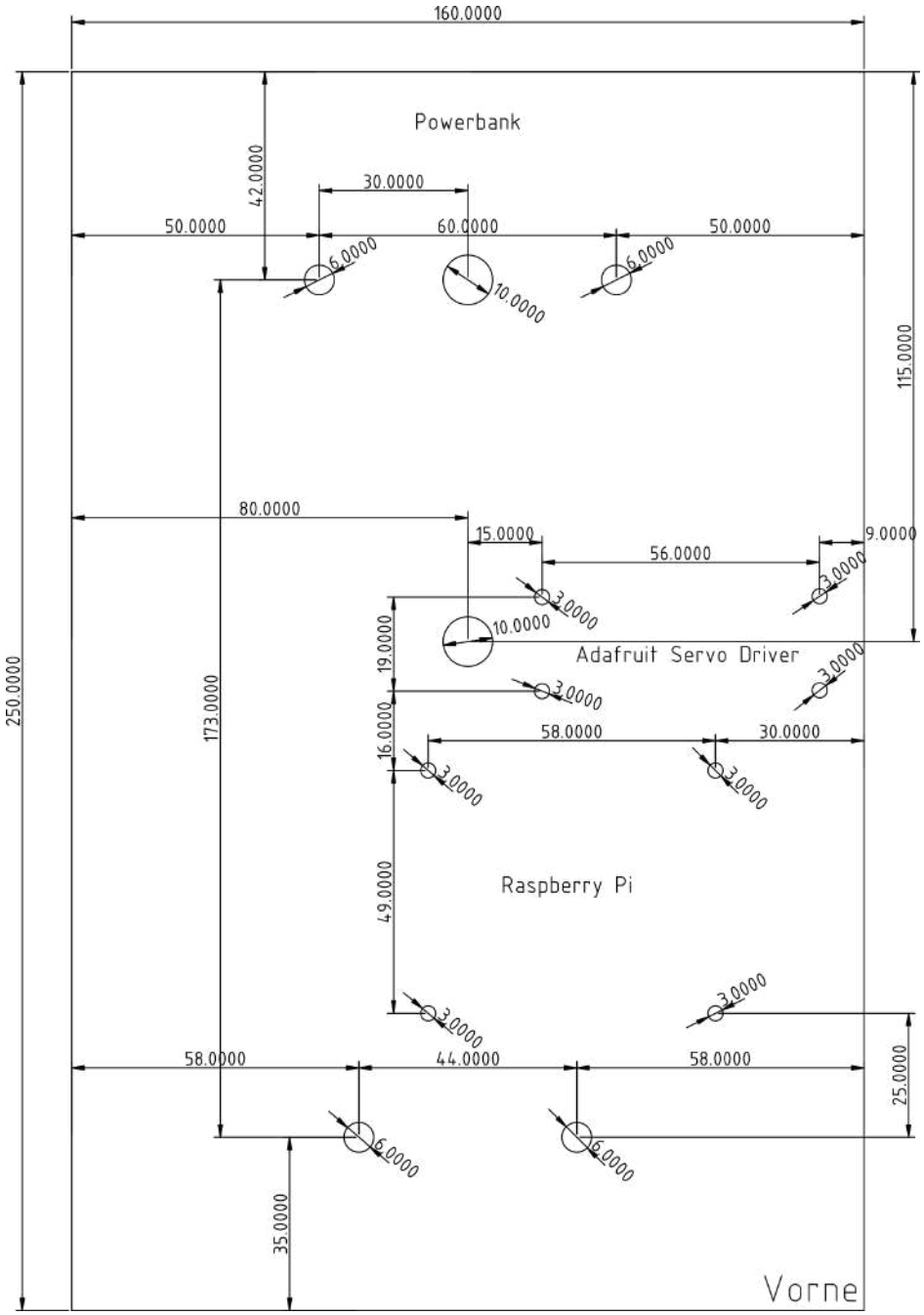
2. An den Servo den zweiarmigen-Adapter mit der schwarzen Schraube anschrauben, und zwar so, dass das Auto ganz nach rechts lenken würde. Den Servo mit Kabelbindern montieren. Bei Bedarf mit Füllmaterial fixieren. Siehe Abbildung A.8.

3. Nun kann der Raspberry Pi eingeschaltet werden. Der Anleitung folgen und **Wichtig:** die `config.py` anpassen. `STEERING_LEFT_PWM = 530` und `STEERING_RIGHT_PWM = 660`.

A.4. Besonderheiten an dem umgebauten Auto

Auf dem im Rahmen dieser Arbeit umgebauten Auto wurde zusätzlich eine Platine zur Stromverteilung eingebaut. Auf dieser befindet sich auch ein Atmega ATtiny85-Mikrocontroller mit Programmierport (ICSP) und einigen explizit herausgeführten Pins, da diese während der Testphase zum Experimentieren benötigt wurden. Die ganze Platine kann bei einem Nachbau ignoriert werden. Im Schaltplan (Abschnitt A.6) wurde sie dennoch zur Dokumentation und der Vollständigkeit halber aufgeführt.

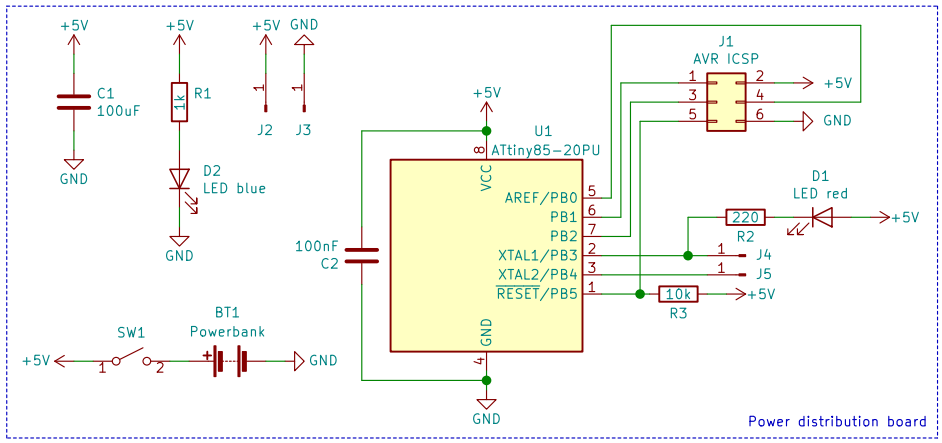
A.5. Maßzeichnung Befestigungsplatte

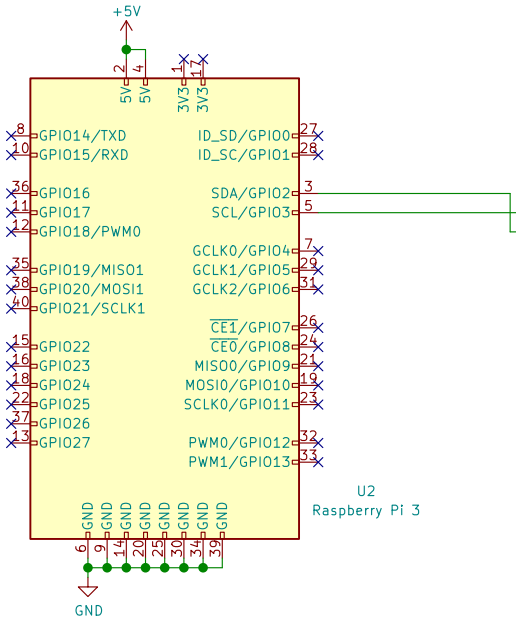


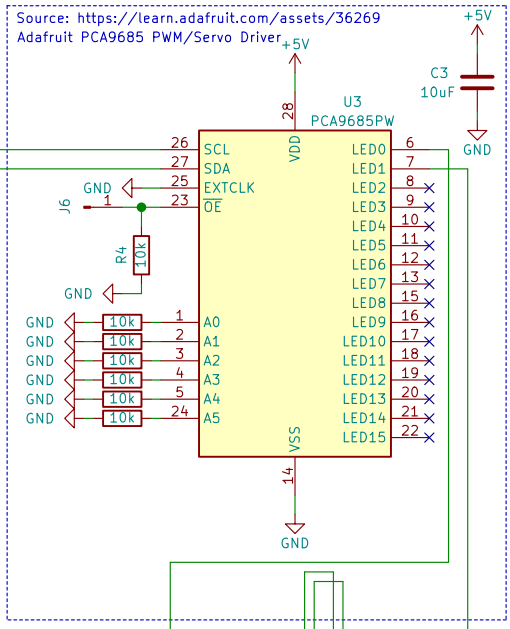
Alle Angaben in Millimetern (mm)

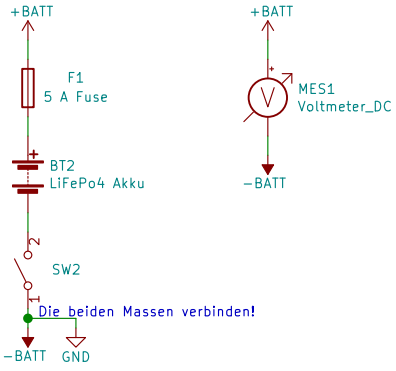
Die Positionen der Löcher und Komponenten wurden so gewählt, dass die Verlegung der Kabel möglichst sinnvoll erfolgen konnte.

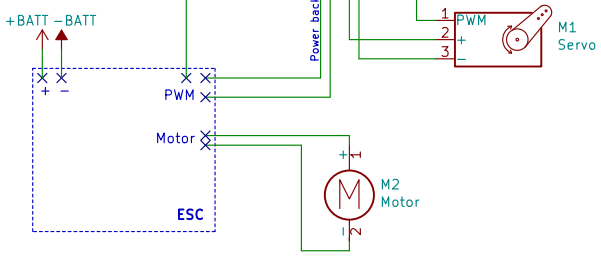
A.6 Schaltplan











Universität Bremen **Christian Friedrich Coors**		
Sheet: / File: circuit.sch		
Title: Donkey Car		
Size: A4	Date:	**Rev: 1.0**
KiCad E.D.A. kicad 5.0.1		Id: 1/1

A.7. Kamerahalter OpenSCAD-Quellcode

```
// Donkey Car camera tower

union() {
    cube([25,90,3]);
    difference() {
        translate([5,62.5,3]) cube([15,25,100]);
        translate([0,65,80]) cube([25,20,4]);
        translate([0,65,65]) cube([25,20,4]);
        translate([0,65,50]) cube([25,20,4]);
    }
}
```

B. Vollständige Ergebnisse der Testläufe

Die drei angegebenen Werte stehen jeweils für Durchlauf 1/Durchlauf 2/Durchschnitt.

Mantissenbits	Ziel erreicht	Integral über Abweichung vom Pfad	Durchschnitt der Abweichung vom Pfad
2	Ja/Ja/-	47.60/50.73/49.17	1.12/1.21/1.16
3	Ja/Ja/-	36.42/28.43/32.43	0.87/0.67/0.77
4	Ja/Ja/-	30.83/30.95/30.89	0.74/0.73/0.73
5	Ja/Ja/-	12.66/12.99/12.82	0.30/0.31/0.31
6	Ja/Ja/-	10.16/10.39/10.27	0.24/0.25/0.24
7	Ja/Ja/-	11.73/12.12/11.92	0.28/0.29/0.28
8	Ja/Ja/-	11.98/11.97/11.97	0.29/0.29/0.29
9	Ja/Ja/-	12.18/11.95/12.06	0.29/0.28/0.29
10	Ja/Ja/-	11.84/11.72/11.78	0.28/0.28/0.28
11	Ja/Ja/-	11.63/11.59/11.61	0.28/0.28/0.28
12	Ja/Ja/-	11.60/11.83/11.71	0.28/0.28/0.28
13	Ja/Ja/-	11.70/11.87/11.78	0.28/0.28/0.28
14	Ja/Ja/-	11.66/11.32/11.49	0.28/0.27/0.27
15	Ja/Ja/-	11.83/11.54/11.68	0.28/0.28/0.28
16	Ja/Ja/-	11.47/11.84/11.66	0.27/0.28/0.28
17	Ja/Ja/-	11.67/11.24/11.45	0.28/0.27/0.27
18	Ja/Ja/-	12.26/11.68/11.97	0.29/0.28/0.29
19	Ja/Ja/-	11.48/11.80/11.64	0.28/0.28/0.28
20	Ja/Ja/-	11.61/11.80/11.70	0.28/0.28/0.28
Ohne Approximation	Ja/Ja/-	11.69/11.76/11.73	0.28/0.28/0.28

Tabelle B.1.: Ergebnisse des Testlaufs der ersten Teststrecke bei langsamer Fahrt

Mantissenbits	Ziel erreicht	Integral über Abweichung vom Pfad	Durchschnitt der Abweichung vom Pfad
2	Ja/Ja/-	20.25/19.63/19.94	1.01/0.98/0.99
3	Ja/Ja/-	10.16/10.04/10.10	0.51/0.51/0.51
4	Ja/Ja/-	7.75/7.83/7.79	0.39/0.39/0.39
5	Ja/Ja/-	4.47/3.70/4.08	0.23/0.19/0.21
6	Ja/Ja/-	4.04/4.03/4.04	0.20/0.20/0.20
7	Ja/Ja/-	4.78/4.50/4.64	0.24/0.23/0.23
8	Ja/Ja/-	4.60/4.40/4.50	0.23/0.22/0.23
9	Ja/Ja/-	4.90/4.37/4.63	0.25/0.22/0.23
10	Ja/Ja/-	4.40/4.67/4.53	0.22/0.23/0.23
11	Ja/Ja/-	4.29/4.45/4.37	0.22/0.22/0.22
12	Ja/Ja/-	4.65/4.20/4.43	0.23/0.21/0.22
13	Ja/Ja/-	4.48/4.14/4.31	0.23/0.21/0.22
14	Ja/Ja/-	3.78/4.02/3.90	0.19/0.20/0.20
15	Ja/Ja/-	4.23/4.28/4.25	0.21/0.21/0.21
16	Ja/Ja/-	4.46/4.17/4.32	0.23/0.21/0.22
17	Ja/Ja/-	4.23/4.52/4.37	0.21/0.23/0.22
18	Ja/Ja/-	4.41/4.39/4.40	0.22/0.22/0.22
19	Ja/Ja/-	4.34/4.20/4.27	0.22/0.21/0.21
20	Ja/Ja/-	4.37/4.69/4.53	0.22/0.24/0.23
Ohne Approximation	Ja/Ja/-	4.24/4.13/4.19	0.21/0.21/0.21

Tabelle B.2.: Ergebnisse des Testlaufs der ersten Teststrecke bei schneller Fahrt

Mantissenbits	Ziel erreicht	Integral über Abweichung vom Pfad	Durchschnitt der Abweichung vom Pfad
2	Nein/Nein/-	-/-/-	-/-/-
3	Nein/Nein/-	-/-/-	-/-/-
4	Nein/Nein/-	-/-/-	-/-/-
5	Nein/Nein/-	-/-/-	-/-/-
6	Ja/Ja/-	101.50/101.08/101.29	2.18/2.16/2.17
7	Ja/Ja/-	30.07/30.05/30.06	0.66/0.65/0.66
8	Ja/Ja/-	27.79/28.03/27.91	0.60/0.61/0.61
9	Ja/Ja/-	27.06/27.06/27.06	0.59/0.59/0.59
10	Ja/Ja/-	26.80/26.53/26.66	0.59/0.58/0.58
11	Ja/Ja/-	26.91/26.52/26.72	0.59/0.58/0.58
12	Ja/Ja/-	26.96/26.42/26.69	0.59/0.58/0.58
13	Ja/Ja/-	26.67/26.48/26.57	0.59/0.58/0.58
14	Ja/Ja/-	26.04/26.11/26.07	0.57/0.57/0.57
15	Ja/Ja/-	26.05/26.22/26.13	0.57/0.57/0.57
16	Ja/Ja/-	26.13/26.55/26.34	0.57/0.58/0.58
17	Ja/Ja/-	26.26/26.54/26.40	0.57/0.58/0.58
18	Ja/Ja/-	26.64/26.01/26.33	0.58/0.57/0.58
19	Ja/Ja/-	26.58/26.02/26.30	0.58/0.57/0.57
20	Ja/Ja/-	25.94/26.63/26.29	0.57/0.58/0.57
Ohne Approximation	Ja/Ja/-	26.53/26.27/26.40	0.58/0.58/0.58

Tabelle B.3.: Ergebnisse des Testlaufs der zweiten Teststrecke bei langsamer Fahrt

Mantissenbits	Ziel erreicht	Integral über Abweichung vom Pfad	Durchschnitt der Abweichung vom Pfad
2	Nein/Nein/-	-/-/-	-/-/-
3	Nein/Nein/-	-/-/-	-/-/-
4	Nein/Nein/-	-/-/-	-/-/-
5	Nein/Nein/-	-/-/-	-/-/-
6	Ja/Ja/-	24.16/24.08/24.12	1.11/1.10/1.11
7	Ja/Ja/-	13.85/13.97/13.91	0.65/0.65/0.65
8	Ja/Ja/-	12.88/13.16/13.02	0.60/0.62/0.61
9	Ja/Ja/-	11.79/11.73/11.76	0.55/0.54/0.54
10	Ja/Ja/-	11.67/12.42/12.05	0.54/0.58/0.56
11	Ja/Ja/-	12.06/11.32/11.69	0.56/0.52/0.54
12	Ja/Ja/-	11.41/12.39/11.90	0.53/0.58/0.55
13	Ja/Ja/-	10.75/11.10/10.93	0.50/0.52/0.51
14	Ja/Ja/-	11.20/10.96/11.08	0.52/0.51/0.52
15	Ja/Ja/-	11.10/10.97/11.03	0.52/0.51/0.51
16	Ja/Ja/-	10.88/11.06/10.97	0.51/0.51/0.51
17	Ja/Ja/-	10.93/10.98/10.95	0.51/0.51/0.51
18	Ja/Ja/-	10.63/11.02/10.83	0.50/0.51/0.51
19	Ja/Ja/-	10.75/11.63/11.19	0.50/0.54/0.52
20	Ja/Ja/-	11.03/11.11/11.07	0.51/0.52/0.51
Ohne Approximation	Ja/Ja/-	11.08/10.69/10.88	0.52/0.50/0.51

Tabelle B.4.: Ergebnisse des Testlaufs der zweiten Teststrecke bei schneller Fahrt

Mantissenbits	Ziel erreicht	Integral über Abweichung vom Pfad	Durchschnitt der Abweichung vom Pfad
2	Nein/Nein/-	-/-/-	-/-/-
3	Nein/Nein/-	-/-/-	-/-/-
4	Nein/Nein/-	-/-/-	-/-/-
5	Nein/Nein/-	-/-/-	-/-/-
6	Nein/Nein/-	-/-/-	-/-/-
7	Ja/Ja/-	97.85/99.03/98.44	1.17/1.19/1.18
8	Ja/Ja/-	83.98/96.49/90.23	1.01/1.16/1.09
9	Ja/Ja/-	82.59/74.84/78.72	0.99/0.90/0.94
10	Ja/Ja/-	74.03/70.33/72.18	0.89/0.85/0.87
11	Ja/Ja/-	70.60/70.46/70.53	0.85/0.85/0.85
12	Ja/Ja/-	70.25/68.14/69.19	0.85/0.83/0.84
13	Ja/Ja/-	69.29/67.69/68.49	0.84/0.82/0.83
14	Ja/Ja/-	68.54/68.73/68.63	0.83/0.83/0.83
15	Ja/Ja/-	69.41/68.70/69.05	0.84/0.83/0.84
16	Ja/Ja/-	67.45/68.91/68.18	0.81/0.84/0.82
17	Ja/Ja/-	69.41/68.25/68.83	0.84/0.82/0.83
18	Ja/Ja/-	69.18/67.66/68.42	0.84/0.82/0.83
19	Ja/Ja/-	71.04/69.57/70.31	0.86/0.84/0.85
20	Ja/Ja/-	69.13/70.33/69.73	0.84/0.85/0.84
Ohne Approximation	Ja/Ja/-	68.55/69.82/69.18	0.82/0.84/0.83

Tabelle B.5.: Ergebnisse des Testlaufs der dritten Teststrecke bei langsamer Fahrt

Mantissenbits	Ziel erreicht	Integral über Abweichung vom Pfad	Durchschnitt der Abweichung vom Pfad
2	Nein/Nein/-	-/-/-	-/-/-
3	Nein/Nein/-	-/-/-	-/-/-
4	Nein/Nein/-	-/-/-	-/-/-
5	Nein/Nein/-	-/-/-	-/-/-
6	Ja/Ja/-	36.32/37.52/36.92	1.02/1.04/1.03
7	Ja/Ja/-	24.02/25.53/24.77	0.68/0.72/0.70
8	Ja/Ja/-	20.50/19.84/20.17	0.58/0.56/0.57
9	Ja/Ja/-	18.40/18.70/18.55	0.53/0.53/0.53
10	Ja/Ja/-	18.27/18.68/18.47	0.52/0.53/0.53
11	Ja/Ja/-	18.48/18.10/18.29	0.53/0.52/0.52
12	Ja/Ja/-	17.96/18.47/18.21	0.51/0.52/0.52
13	Ja/Ja/-	17.79/18.61/18.20	0.51/0.53/0.52
14	Ja/Ja/-	17.58/17.84/17.71	0.50/0.51/0.50
15	Ja/Ja/-	17.73/17.71/17.72	0.51/0.50/0.50
16	Ja/Ja/-	18.08/18.01/18.04	0.51/0.51/0.51
17	Ja/Ja/-	17.93/17.65/17.79	0.50/0.50/0.50
18	Ja/Ja/-	17.69/18.19/17.94	0.50/0.52/0.51
19	Ja/Ja/-	18.05/17.57/17.81	0.51/0.50/0.51
20	Ja/Ja/-	17.60/18.65/18.13	0.50/0.53/0.52
Ohne Approximation	Ja/Ja/-	18.06/17.37/17.71	0.52/0.49/0.50

Tabelle B.6.: Ergebnisse des Testlaufs der dritten Teststrecke bei schneller Fahrt

C. Inhalt des beiliegenden Datenträgers

float_precision/	Programm und Skripte zur Erstellung der Abbildung 2.2
float_test/	Programm zur Erstellung von Tabelle 1.1
model/	Das zur Evaluation verwendete trainierte Netz im Keras- und TensorFlow-Format
noise/	Die Programme und Skripte, die zur Einzelbildanalyse mit dem Rauschen verwendet wurden
operations/	Das Projekt mit den approximierten TensorFlow Operationen
sdsim_patched/	Das um Pfadabweichung und die Teststrecken erweiterte Unity-Projekt des Donkey Simulators
simclient/	Der Simclient inklusive Plotting-Skripten
umbau/	Quelle des Kamerahalters, Maßzeichnung der Befestigungsplatte im DXF-Format und Quelldateien des Schaltplans
thesis.pdf	Die Bachelorarbeit im PDF-Format