Autonome Spurhaltung durch Maschinelles Lernen

Daniel Riege

Department Computer Science, HAW Hamburg,
Berliner Tor 7, 20099 Hamburg

Abstract. Eine autonome Spurhaltung, mit Hilfe eines Machine Learning Ansatzes, wird in dieser Projektausarbeitung beschrieben. Das Ziel war es ein ferngesteuertes Auto, im Maßstab 1:10, innerhalb einer Spur zu halten, ohne dabei Bildverarbeitungs Algorithmen zu verwenden. Dafür wurde ein convolutional neural net (CNN), welches auf NVIDIAs PilotNet basiert, trainiert. Dieses ist in der Lage einzelne Bildpunkte auf einen Lenkwinkel, für das Auto, abzubilden. Als Testplatform wurde dafür ein Tamiya RC car verwendet. Die Rechenleistung, für die Inferenzen des CNN, kommen von einem Raspberry Pi4 in Kombination mit einem Google Coral Stick. Mit nur einer handvoll gelabelten on-board Bildern von einer Fahrbahn, war das CNN in der Lage den richtigen Lenkwinkel vorherzusagen, um das Auto auf der Spur zu halten. Selbst unbekannte Straßenzüge oder nicht vorhandene Fahrbahnmarkierungen waren kein Problem.

Keywords: Spurhaltung · Maschinelles Lernen · autonomes Fahren · CNN · 1:10 Modellauto

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Ein autonomes Fahrzeug besteht aus mehreren wichtigen Komponenten. Eines davon ist es, das Fahrzeug in der Spur zu halten. Für einen menschlichen Fahrer ist es trivial ein Auto auf der Fahrbahn zu halten. Aber für einen Computer ist es nicht so einfach. Viele Algorithmen für die Spurhaltung basieren auf Bibliotheken wie OpenCV um die Fahrbahn zu erkennen und dann den Pfad planen, um den richtigen Lenkeinschlag zu berechnen. Dieser Ansatz läuft sehr stabil und die berechneten Lenkwinkel sind sehr genau, jedoch kommt dieser Ansatz nicht ohne Probleme. Straßen besitzen nicht immer sichtbare Fahrbahnmarkierungen, vor allem in Städten sind diese oft abgefahren. In Wohngebieten bilden oft parkende Autos am Straßenrand die Fahrbahnbegrenzung. Um diesen Problemen entgegenzuwirken, testete NVIDIA ein Faltungsnetzwerk (CNN), um den Lenkwinkel vorherzusagen. Als Input dient dabei lediglich eine Frontkamera. Diese Idee scheint vielversprechend, da das Auto vom Menschen lernt und dabei nicht auf sichtbare Straßenmarkierungen angewiesen ist [5]. Um das Auto zu trainieren sollen selbst aufgenommene Daten dienen. Dabei sollen von verschiedenen Positionen auf der Strecke Bilder aufgenommen werden, von der Frontkamera, und diese mit dem aktuellen Lenkwinkel abgespeichert werden. Der Lenkwinkel kommt durch das Fernsteuerung über den Rundkurs zu Stande. Der Kurs soll verschiedene Kurven beinhalten. Als Fahrbahnmarkierung soll Straßenkreide dienen.
Die folgenden Kapitel beschäftigen sich mit dem Aufbau eines solchen autonomen Fahrzeugs im Maßstab 1:10. Der Fokus liegt dabei auf der Implementierung und Optimierung des PilotNet Netzwerkes in einem eingebettetem System.

2 Hardware

2.1 Allgemeiner Aufbau

Das Tamiya Modellauto enthält bereits alle Komponenten, die es brauch, damit das Fahrzeug sich fortbewegen kann. Um das Modellauto allerdings mit einem Raspberry Pi anzusteuern, benötigt es einer anderen Verkabelung und zusätzlicher Komponenten.

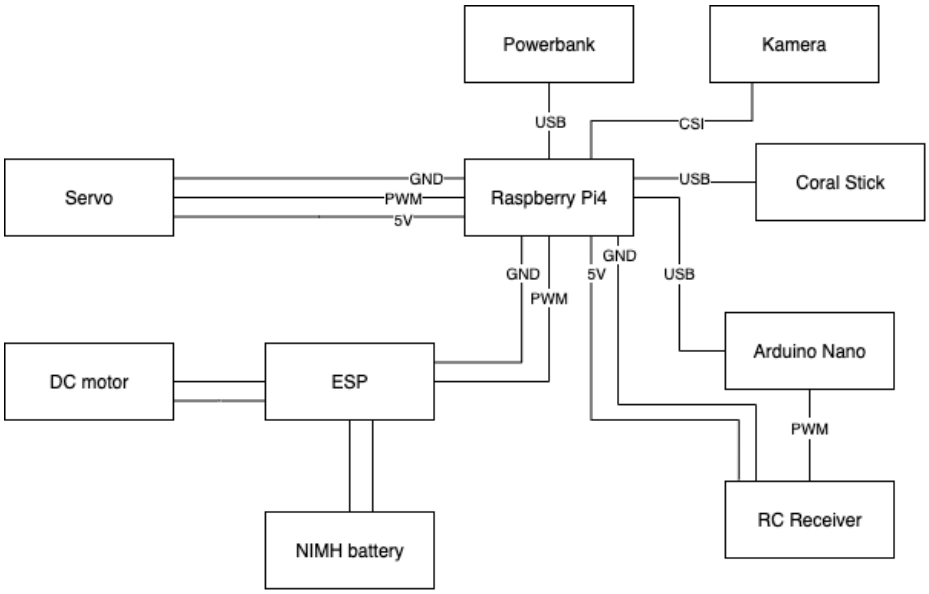

Fig. 2.1. Schaltplan für das modifizierte Modellauto

Abbildung 2.1 zeigt den Schaltplan des umgebauten Modellautos. Die Stromversorgung für den Motor wird von einem 7.2V Nickel-Metallhydrid Akkupacks gewährleistet. Da der Motor bei Vollgas extrem viel Strom zieht, ist es sinnvoll die restlichen Komponenten wie den Raspberry Pi oder Coral Stick über eine externe Stromversorgung zu versorgen. Daher ist an des Raspberry eine USB Powerbank angeschlossen, die einen maximalen Strom von 3A abgeben kann, den der Raspberry Pi 4 laut Datenblatt auch braucht [6]. Die an den Pi angeschlossenen Komponenten, wie der Coral Stick, Lenkservo oder Arduino Nano werden vom Pi mit Strom versorgt. Nur der Fahrtenregler (ESP) bezieht seinen Strom vom Akkupack, da dieser den Bürstenmotor steuert.

2.2 PWM Messungen am Empfänger

Obwohl das Fahrzeug am Ende autonom fahren soll, wird der Antriebsmotor dennoch von außen, über eine Fernbedienung, gesteuert. Dadurch muss der RC

Receiver erhalten bleiben. Dieser steuert nun nicht den Lenkservo und Fahrtenregler direkt an, sondern ist ebenfalls an den Raspberry Pi angeschlossen, da dieser nun die zentrale Steuereinheit bildet. Normalerweise kommunizieren einzelne Komponenten im Modellbau über eine Pulsweiten-Modulation (PWM). Mit Hilfe von PWM kann ein digitaler Output ein analoges Signal simulieren, indem die Länge des HIGH Pulses relativ zur Periodendauer variiert. Im Modellbau werden damit allerdings auf digitaler Ebene Servos und Fahrtenregler gesteuert. Dabei gilt die Länge des HIGH Pulses als Wert für z.B. den Servowinkel [7]. In der Regel operieren diese PWM Signale mit 50Hz.

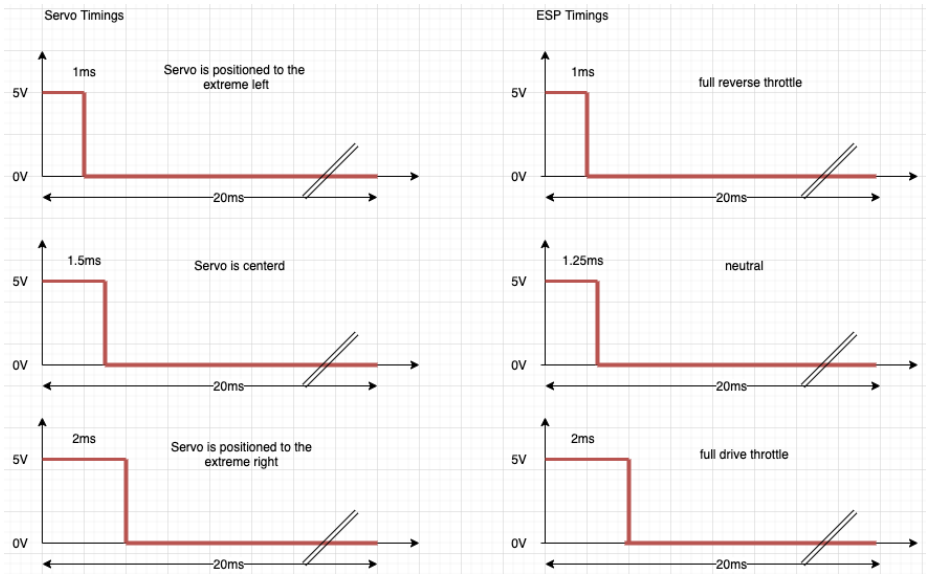

Fig. 2.2. PWM Timings für den Lenkservo und Fahrtenregler

Abbildung 2.2 zeigt die Timings der PWM Signale, die vom RC Empfänger ausgesendet werden. Da der Empfänger sowohl den Servo also auch Motor ansteuert, besitzt der Empfänger zwei Kanäle. Um diese Signale nun zu messen, gibt es verschiedene Verfahren, wie z.B. das PWM Signal mit Hilfe eines analog-digital-converter (ADC) abzutasten. Da dafür der ADC aber genau eingestellt werden muss, um über die komplette Periodendauer hinweg zu messen und keine Momentaufnahme zu machen, wird in diesem Projekt ein anderes Verfahren verwendet. Es wird die Zeit zwischen steigender und fallender Flanke gemessen. Dafür wird an einen GPIO ein Interrupt auf steigender und fallender Flanke angehängt. Bei steigender Flanke wird der aktuelle Zeitstempel gespeichert. Bei fallender Flanke wird die Differenz aus aktuellen und gespeichertem Zeitstempel gebildet, um so die Länge des HIGH Pulses in Nanosekunden zu erhalten. Da die PWM Signale mit 50Hz arbeiten und nur der Bereich von 5-10% von Bedeutung

ist, bewegt sich die Zeitmessung in einem Bereich von 1000-2000ns. Der erste Ansatz war es, diese Zeitmessung auf dem Raspberry Pi direkt durchzuführen. Auf diesem Pi sollte nachher allerdings auch noch die komplette Inferenz des Faltungsnetzes laufen. So stand die Frage im Raum, ob diese CPU Auslastung eine Auswirkung auf die Zeitmessung haben könnte. Daraufhin wurde ein Versuch durchgeführt, bei dem ein PWM Signal von 2ms am Raspberry Pi gemessen werden sollte. Einmal ohne einen großen Prozess im Hintergrund und einmal mit einem Tensorflow Beispielprogramm, welches Gegenstände durch die Kamera klassifiziert. Um noch Mikroprozessor mit Mikrocontroller vergleichen zu können, bei einer so zeitkritischen Messung, wurde dieses Signal ebenfalls an einem Arduino Nano gemessen. Das 2ms PWM Signal wurde von einem Arduino Uno generiert.

Periode	Raspberry normal	Raspberry vollast	Arduino Nano
1	2037	2045	2060
2	2070	1938	2060
3	2044	2066	2060
4	2090	18154	2061
5	2088	9415	2061
6	2042	2017	2060
7	2024	1972	2060
8	2036	2467	2059

Table 2.1. Zeitmessungen eines 2ms PWM Signals

(Alle Angaben in ns)

Wie man in Tabelle 2.1 sehen kann, schwanken die Werte am Raspberry Pi stärker als am Arduino Nano. Wenn der Raspberry nun noch eine rechenaufwändige Anwendung im Hintergrund laufen hat, so sind die Messwerte unbrauchbar, wenn man sich nochmal die Timings anschaut in denen der Servo und Fahrtenregler operieren. So viel die Entscheidung, einen Arduino Nano zu verwenden, um die PWM Signale zu messen. Diese gemessenen Signale werden dann als digitale Integer über die serielle Schnittstelle, hier über USB, an den Raspberry übertragen.

```
1  void prepareAndSend() {
2    pwm_valueCh2 = new_timeCh2 - prev_timeCh2;
3    pwm_valueCh1 = new_timeCh1 - prev_timeCh1;
4
5    pwm_valueCh2 = frequency_correction(pwm_valueCh2);
6    pwm_valueCh2 = trim(2, pwm_valueCh2);
7    pwm_valueCh1 = frequency_correction(pwm_valueCh1);
8    pwm_valueCh1 = trim(1, pwm_valueCh1);
9    uint16_t payloadCh1 = pwm_valueCh1 & PWMMASK;
10   Serial.println(payloadCh1);
```

```
11    uint16_t payloadCh2 = pwm_valueCh2 & PWM_MASK;
12    payloadCh2 = payloadCh2 | CHANNEL_MASK;
13    Serial.println(payloadCh2);
14  }
```

Code 2.1. UART Übertragung der Kanalwerte

Für die Übertragung sind nur Zeile 9-13 in Code 2.1 wichtig. Davor findet die Berechnung der Zeitdifferenz und Korrekturen statt. Diese Korrekturen werden noch näher erläutert. Für das Senden der Daten wird die println Funktion der Arduino Library verwendet. Um am Raspberry nun herauszufinden, welcher Wert zum Servo und welcher zum Fahrtenregler gehört, ist der MSB einer einzelnen Zeile der Channel. Dabei gilt für Channel 1 (Servo) MSB 0 und Channel 2 (Fahrtenregler) MSB 1. Die Funktion prepareAndSend() wird aufgerufen, nachdem am Kanal 2 die Leitung auf 0 gezogen wird.

```
1  void risingCh2() {
2    attachInterrupt(1, fallingCh2, FALLING);
3    prev_timeCh2 = micros();
4  }
5
6  void fallingCh2() {
7    attachInterrupt(1, risingCh2, RISING);
8    new_timeCh2 = micros();
9    // this will be done here due to timing
10   // falling edge ch2 is the last interrupt within a 50Hz period
11   prepareAndSend();
12 }
13
14 void risingCh1() {
15   attachInterrupt(0, fallingCh1, FALLING);
16   prev_timeCh1 = micros();
17 }
18
19 void fallingCh1() {
20   attachInterrupt(0, risingCh1, RISING);
21   new_timeCh1 = micros();
22 }
```

Code 2.2. Externe Interrupts für zwei PWM Kanäle

Dies ist so möglich, da die Funkverbindung zwischen Sender und Empfänger auf nur einer Funkfrequenz läuft und die Daten der zwei Kanäle nacheinander gesendet werden muss. Dieses Verhalten sieht man auch, wenn man die PWM Signale misst. Es wird zuerst die Leitung für Kanal 1 auf HIGH und dann wieder auf LOW gezogen und erst dann fängt Kanal 2 an. Deshalb arbeiten die Modellbau Komponenten wahrscheinlich auch im Bereich von 1-2ms, damit innerhalb eines 20ms Zyklus mehrere Kanäle übertragen werden können.

2.3 Probleme mit Empfänger

Beim Testen mit dem realen Empfänger ist vor allem eines aufgefallen. Die Werte waren nicht wie erwartet, wenn man die Fernsteuerung verwendet hat. Auf Nullstellung bei der Lenkung würde man einen Wert von 1.5ms erwarten (s.o.). Doch es wurden 1.42ms gemessen. Dies könnte an der PWM Messung liegen, jedoch wenn man den Servo direkt an den Empfänger anschließt (ohne Arduino oder Raspberry dazwischen), hat man das selbe Verhalten feststellen können. Die Reifen waren nach links eingeschlagen, so weit, dass man das selbst mit der Trimmung an der Fernsteuerung nicht korrigieren konnte. Nach zwei weiteren Messungen ergaben sich folgende Werte.

Tatsächlich	Gemessen	Differenz
2000	1830	170
1500	1420	80
1000	980	20

Table 2.2. PWM Messungen am Empfänger

(Alle Angaben in ns)

Schaut man sich Tabelle 2.2 an, sieht man dass die Differenzen nicht gleich sind. Es ist also kein Trimmungs bzw. Messproblem durch Offset. Daher wurde einmal die Periode gemessen zwischen steigernder und steigernder Flanke auf Kanal 1. Dabei ergab sich eine Periodendauer von 18352ns = 54.4899738 Hz. Aus irgendeinem Grund ist die PWM Frequenz am Empfänger verstellt, die man auch nicht wieder auf 50Hz stellen kann. So muss die Frequenz durch Software angepasst werden, indem man die gemessenen Werte umrechnet.

$$f(t) = \frac{t * 54.4899738Hz}{50Hz} \tag{1}$$

t ist dabei die gemessene PWM Zeit. Diese Rechnung wird auch in der oben (Code 2.1 Zeile 5 u. 7) angesprochenen Korrektur durchgeführt. Nach dieser Frequenzkorrektur wird pro Kanal noch eine Trimmung durchgeführt, die normalerweise an der Fernsteuerung vorgenommen wird, hier aber durch Software. Dabei werden jediglich kleine Verstellungen am Servo korrigiert.

3 Software

3.1 allgemeiner Aufbau

In diesem Kapitel geht es primär um die Software Architektur am Raspberry Pi. Der gesamte Projektcode für den Raspberry ist in Python geschrieben, da dies für den Machine Learning Kontext sinnvoll erschien.

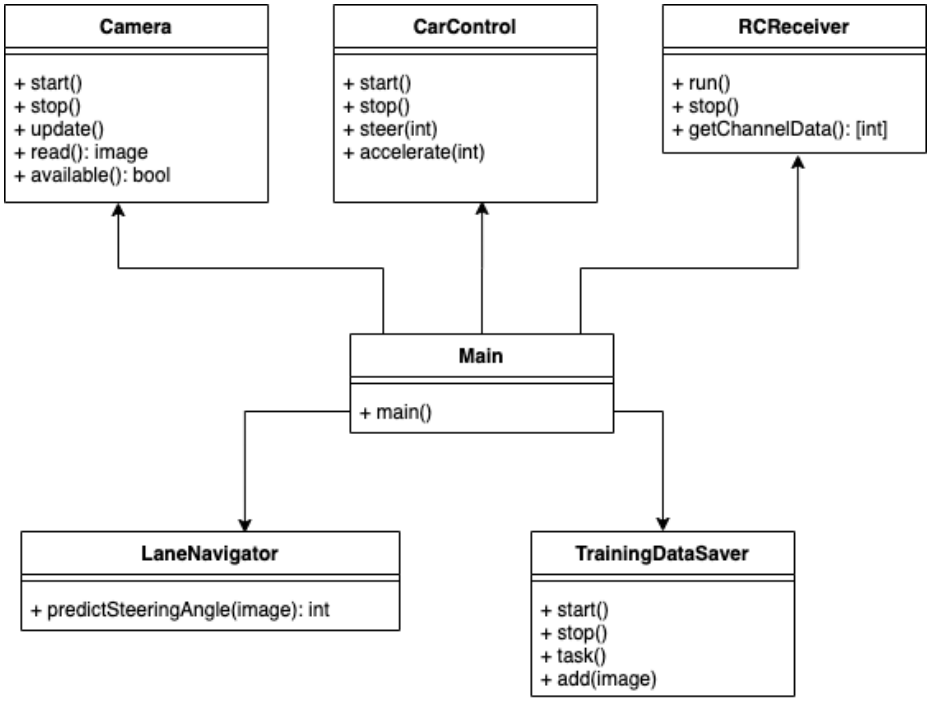

Fig. 3.1. Klassendiagramm des Python Projektes

Abbildung 3.1 zeigt das Klassendiagramms und damit den Aufbau des gesamtes Projektes für den Raspberry Pi. Da das Auto vom menschlichen Verhalten lernen soll, gibt es zwei Fahrmodi, den man beim Start des Programms auswählen kann. Einmal den autonomes Modus, bei dem die Lenkung voll autonom, anhand des trainierten Modells, die Gassteuerung aber noch über die Fernsteuerung passiert. Der andere Modus ist der Trainings Modus, bei dem das Auto komplett manuell über die Fernsteuerung gesteuert wird und nur die Trainingsdaten abspeichert.
Der Programmablauf soll sich am EVA Prinzip orientieren, sprich ein wiederkehrender Ablauf von Eingabe-Verarbeitung-Ausgabe. In der Main soll es eine Schleife geben, die sich das aktuelle Bild von der Kamera holt und im Trainingsmodus

sich den Input des Empfängers besorgt (Eingabe). Die Verarbeitung ist im autonomen Betrieb die Inferenz des CNNs und im Training die Speicherung von Trainingsdaten. Die Ausgabe ist die Ansteuerung der Hardware, also des Lenkservos und Fahrtenreglers für den Motor. Dieser Ablauf wird in folgender Abbildung nochmals verdeutlicht.

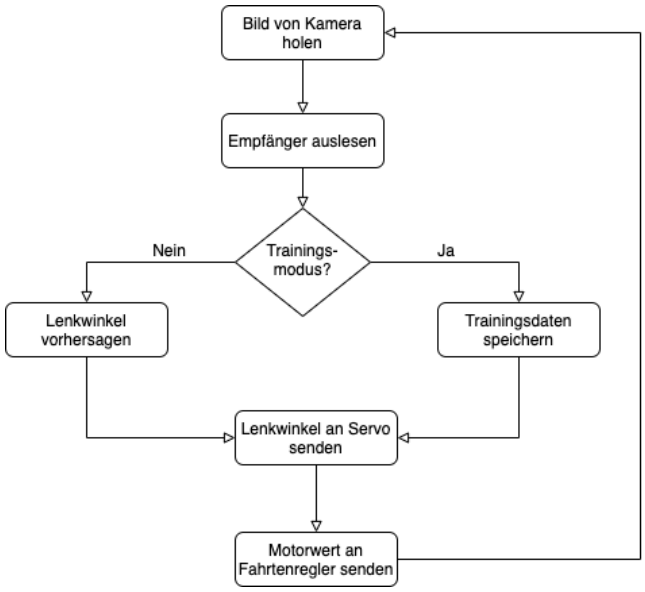

Fig. 3.2. Ablauf des Main Programms nach EVA Prinzip

Da es sich um ein zeitkritisches System handelt, muss ein solcher Schleifendurchlauf innerhalb einer bestimmten Zeit passieren. Angestrebt werden für dieses Projekt 30Hz. Die Eingabe, Verarbeitung und Ausgabe dürfen maximal also nicht 33ms überschreiten. Dies erfordert auf Grund der leistungsbeschränkten Hardware einiger Optimierungen, die in den folgenden Abschnitten näher erläutert werden.

3.2 Kamera

Die Kamera ist der einzige Sensor, der in diesem Projekt verwendet wird. Da die Kamera über einen speziellen Anschluss am Pi angeschlossen wird [6], wurde diese nicht im Hardware Kapitel erwähnt. Viel interessanter hingegen ist die Ansteuerung der Kamera auf Softwareebene. Es gibt verschiedene Arten den Kamera-Feed auszulesen, da es verschiedene Bibliotheken gibt. Der erste Ansatz war es die Picamera Library zu verwenden, um kontinuierlich Bilder zu machen. Mittels OpenCV sollten diese Bilder dann als Preview angezeigt werden.

```
1 # initialize the camera and grab a reference to the raw camera capture
```

```
2  camera = PiCamera()
3  camera.resolution = (640, 480)
4  camera.framerate = 30
5  rawCapture = PiRGBArray(camera, size=(640, 480))
6
7  # capture frames from the camera
8  for frame in camera.capture_continuous(rawCapture,
9                                          format="bgr",
10                                         use_video_port=True):
11     image = frame.array
12
13     # show the frame
14     cv2.imshow("Frame", image)
15     ...
```

Code 3.1. Zugriff auf Kamera Feed mit Preview über Picamera

Da die Main-Schleife mit 30Hz laufen soll, muss auch die Kamera ihr Bild alle 33ms aktualisieren. In Zeile 4 wird die Framerate auf 30 FPS eingestellt, jedoch ist damit noch nicht gewährleistet, dass auch das Laden des Kamerabildes schnell genug geschieht. Nach einer Zeitmessung des in Code 3.1 gezeigten Codes, ergab sich eine ungefähre Periodenzeit von 94ms, was 10Hz entsprechen würde. Das ist für die oben gezeigten Anforderungen viel zu langsam. Es gibt allerdings noch eine zweite Methode um auf den Kamera-Feed zuzugreifen. Die Bibliothek OpenCV bietet auch eine Möglichkeit auf eine Raspberry Pi Kamera zuzugreifen.

```
1  stream = cv2.VideoCapture(0)
2  stream.set(cv2.CAP_PROP_FRAME_WIDTH,640)
3  stream.set(cv2.CAP_PROP_FRAME_HEIGHT,480)
4  stream.set(cv2.CAP_PROP_FPS, 30)
5
6  while True:
7      ret, img = stream.read()
8      cv2.imshow("Frame", img)
```

Code 3.2. Zugriff auf Kamera Feed mit Preview über OpenCV

Im Prinzip wird das selbe gemacht wie in Code 3.1, jedoch rein durch OpenCV. Das Bild wird in dieser Methode über das read(), welches blockierend ist, in Zeile 7 geladen. Nach einer Zeitmessung mit diesem Code, hat sich eine Periodenzeit von ca. 33ms ergeben, was bedeutet, dass das Laden des Bildes so schnell geht, dass die 30 FPS der Kamera eingehalten werden können. Jetzt sollen aber innerhalb dieser 33ms noch andere Berechnungen durchgeführt werden, die ebenfalls zeitspielig ist. Zur Demonstration wurde daraufhin innerhalb der While Schleife nach Zeile 8 noch ein delay von 20ms eingebaut, welches z.B. die Inferenz des CNN simulieren soll. Auch hier ergab sich eine Periodenzeit von 33ms. Nach einer weiteren Erhöhung des delays auf 21ms, wurden manche Frames übersprungen wodurch die Periodenzeit zwischen 33ms und 66ms gesprungen ist. Man kann also sagen, dass das Laden des Kamerabildes in etwa 13ms braucht. So bleiben noch 20ms

für restliche Berechnungen in der Main Schleife.
Wie bereits kurz angeschnitten ist die read Methode, aus Zeile 7 Code 3.2, blockierend. Das heißt diese Methode wartet, bis das Bild geladen und dekodiert wurde. Da dieser Code auf dem main Thread läuft geht viel Zeit verloren. Die weitere Idee war es nun, dieses Laden des Kamera-Feeds in einen anderen Thread auszulagern. Das Ziel ist es dabei mehr Zeit in der Main-Schleife zu gewinnen, als die getesteten 20ms. Die Main-Schleife muss aber trotzdem noch das Bild bekommen. So soll der main Thread und der Thread für die Kamera auf einen gemeinsamen Speicher zugreifen. Dieser gemeinsame Speicher wird durch die Standard Queue in Python realisiert. Das besondere bei dieser ist, dass die get() Methode blockierend ist. Also wenn keine zu verarbeitende Bilder in der Queue liegen, würde der main Thread warten, bis wieder ein Bild drin liegt. Dadurch kann sich die CPU ausruhen, wenn sie alle Bilder abgearbeitet hat, die sich eventuell gestaut haben durch zu lange Rechenzeiten bei vorherigen Bildern. So ergibt sich folgender Code für den Kamera-Feed.

```
1       def update(self):
2           while True:
3               if self.stopped:
4                   print("[Camera] thread stopped.")
5                   return
6               # ensure the queue has room in it
7               if not self.Q.full():
8               # read the next frame from the file
9                   (grabbed, frame) = self.stream.read()
10                  if not grabbed:
11                      continue
12                  self.Q.put(frame)
```

Code 3.3. Laden des Kamera-Feed über OpenCV in eine Queue

Diese update Methode läuft dabei auf einem eigenem Thread in der Klasse Kamera. Die Initialisierung des Streams ist dabei die gleiche wie in Code 3.2.

```
1       def read(self):
2           # return next frame in the queue
3           return self.Q.get()
```

Code 3.4. Laden des ersten Frames aus der Queue

Die read Methode der Kamera Klasse wird dann aus der Main-Schleife aufgerufen und gibt ersten Frame aus der Queue zurück. Im idealen Fall liegt immer nur ein Frame in der Queue. Wenn allerdings die Main-Schleife für einen Durchlauf länger braucht als 30ms, so füllt sich diese Queue weiter auf. So kann kein Frame verloren gehen, was gerade beim Sammeln von Trainingsdaten wichtig ist. Ist die Main-Schleife schneller als 30ms, so ist die read Methode dieser Klasse blockierend, da die get Methode der Queue Library blockiert, bis ein neuer Frame drin liegt.

3.3 RCReceiver

Wie Bereits im Kapitel über die Hardware angesprochen, werden die PWM Signale des Empfängers mit einem Arduino Nano gemessen und über USB an den Raspberry gesendet. Dadurch muss diese Python Klasse nicht viel machen, außer die übertragenen Daten vom Arduino entgegenzunehmen. Mit Hilfe der Bibliothek Serial [4] können serial ports geöffnet und ausgelesen werden. Da auf dem verwendeten Raspberry eine Linux Distribution läuft, ist diese serielle Schnittstelle als Datei im /dev Verzeichnis repräsentiert. Durch einen Scan des /dev Verzeichnisses lässt sich der Pfad zum Arduino schnell finden. Wie auch die Kameraklasse, läuft das Auslesen der Datei in dieser Klasse auf einem seperaten Thread.

```
1  try:
2      self.serial.open()
3      print("[RCReceiver] serial port opened. Listening started.")
4  except:
5      print("[RCReceiver] error opening serial port.")
6      return
7  while self.running:
8      if(self.serial.in_waiting >1):
9          try:
10             line = int(self.serial.readline().decode("utf-8"))
11             ch = (line & CH_MASK) >> 15
12             pwm = line & PWM_MASK
13             if pwm in self.validRange:
14                 self.channelData[ch] = pwm
15                 if self.callback:
16                     self.callback(ch, pwm)
17         except ValueError:
18             print("[RCReceiver] warning: ValueError received")
19             continue
```

Code 3.5. Auslesen des Devicefiles für Empfängerwerte

Als erstes muss die Devicedatei geöffnet werden. Sollte die Datei nicht gefunden werden, weil z.B. die Verbindung getrennt wurde, so soll die Funktion abbrechen. In Zeile 8 wird dann gewartet, bis der Arduino einen neuen Wert gesendet hat (vgl. oben, die Kanäle werden einzeln gesendet). in Zeile 8 wird dieser dann ausgelesen und richtig decodiert. Mit Hilfe einer Maske wird dann überprüft wie das MSB gesetzt ist, also ob es sich um Kanal 1 oder Kanal 2 handelt. Der PWM Wert wird dann an jeweilige Stelle in einen Zwischenspeicher geschrieben. Dieser Zwischenspeicher beinhaltet immer nur den aktuellen Wert. Wie bei der Kamera, wird dieser Zwischenspeicher aus der Main-Schleife über eine extra Methode ausgelesen.

```
1      def getChannelData(self):
```

```
2         return self.channelData
```

Code 3.6. Auslesen des Zwischenspeichers für Kanalwerte

Was in Code 3.5 allerdings noch auffällt, dass nach dem speichern der Kanalwerte eine Callback function aufgerufen wird. Das hat den Hintergrund dass der in Abbildung 3.2 gezeigter Ablauf der Main Methode nicht ganz so übernommen wurde. Nämlich wird der Kanalwert für den Motor direkt weitergeleitet, bevor jegliche Verarbeitung des Kamerabildes stattfindet. Dieses Weiterleiten findet auch nicht in der Main-Schleife im main Thread statt, sondern im Thread der RcReceiver-Klasse. Für den Fall, dass eine Berechnung in der Main-Schleife viel zu lange braucht oder sich etwas dort aufhängt, soll das Fahrzeug trotzdem noch manövrierfähig sein. Es hat also einen Sicherheitskritischen Aspekt. Am Sichersten wäre es wahrscheinlich diesen Teil komplett auf einen anderen Prozess auszulagern, innerhalb dieses Projekts wurde dies aber nur auf einen externen Thread gemacht. Zusammengefasst wird also innerhalb des aufgerufenen Callbacks direkt der Motor angesprochen und im Trainingsmodus ebenfalls die Lenkung.

3.4 TrainingDataSaver

Damit man das Auto trainieren kann, braucht es Trainingsdaten. Wie bereits kurz in der Einleitung angesprochen, bestehen diese aus Bildern mit den dazugehörigen Lenkwinkeln, die durch das menschliche Steuern des Fahrzeuges zu Stande kommen. Die Idee dabei ist es, den Lenkwinkel mit in den Dateinamen des jeweiligen Bildes zu schreiben. So ist es im Nachhinein einfacher den Lenkwinkel für ein Bild auszulesen. Da es sich hier um ein Modellauto handelt, wird der Lenkwinkel nicht in Grad angegeben, sondern als der PWM Wert, den der Empfänger ausliest. Dieser liegt bekanntlich zwischen 1000-2000, wobei 1500 Nullstellung, also geradeaus bedeutet. Um das Bild im laufenden Betrieb abzuspeichern, wird die imwrite() Methode von OpenVC verwendet.

```
1 import cv2
2 cv2.imwrite("./training_data/%s/%05d_%04d_%04d.jpg" %
3                                   (name,loopRun,ch1,  ch2),frame)
```

Code 3.7. Abspeichern eines gelabelten Bildes

In Code 3.7 wird dieses Abspeichern dargestellt. Um die Daten später besser zuordnen zu können, werden alle Trainingsdaten in ein Verzeichnis /training_data gelegt. Innerhalb dieses Verzeichnisses gibt es weitere Unterverzeichnisse, die nach dem jeweiligen Testlauf benannt sind z.B. Strasse_Rundkurs_1. Im Dateinamen des Bildes sind drei Informationen hinterlegt. Die erste ist der fünfstellige loopRun. Dieses ist nur ein Zähler, welcher bei jedem Bild um eins erhöht wird. Dadurch soll man am Ende die Bilder sortieren können, sodass diese in der richtigen Reihenfolge gespeichert werden. Als nächstes hat man die beiden Kanalwerte. Der erste ist der PWM Wert für den Lenkservo, sprich der Lenkwinkel. Der zweite ist der PWM Wert für den Fahrtenregler. Dieser wurde auch mit abgespeichert, für

den Fall, dass man bei späteren Tests die Automatisierung der Motorsteuerung inkludieren will. Der zweite und letzte Parameter, der der imwrite Funktion übergeben wird, ist der Frame an sich, welchen man durch die Kamera erhält. Das Abspeichern scheint trivial, jedoch geschieht es in Echtzeit. Deshalb muss man bedenken, dass dieser Vorgang eine gewisse Zeit nicht überschreiten darf. Nach einigen Testläufen wurde festgestellt, dass langsam der in Kapitel 3.2 angesprochene Frame Buffer (die Queue für die Bilder) vollläuft. Das Speichern braucht bei manchen Bildern also länger als 30ms. Messungen haben ergeben, dass manche Vorgänge teilweise 120ms brauchten. Woran das liegen könnte, ist mir unbekannt, jedoch muss man dieses Problem lösen. Eine Idee war es, einen Thread Pool zu bauen. Also mehrere Threads zu starten, welche sich eine Aufgabe aus einer Queue holen. In diesem Falle würde die Queue mit einem Tupel aus einem Bild und Label Informationen, wie in Code 3.7 dargestellt, gefüllt werden. Wenn ein Thread nun seine vorherige Aufgabe erledigt hat, würde er sich ein neues Tupel aus der Queue holen und dieses, wie bereits gezeigt, abspeichern. Hier wird wieder die Python Standard Queue verwendet, die von Haus aus bereits Thread-Safe ist. Die Threadmethode würde dann wie folgt aussehen.

```
1 def task(self):
2     print("[TrainingDataSaver] new thread started.")
3     while not self.stopped:
4         (frame, loopRun, ch1, ch2) = self.Q.get()
5         cv2.imwrite("./training_data/%s/%05d_%04d_%04d.jpg" %
6             (self.name,loopRun,ch1, ch2),frame)
7     print("[TrainingDataSaver] one thread stopped.")
```

Code 3.8. Threadmethode um das Bildspeichern in einen Pool auszulagern

Die Idee dahinter ist es, dass wenn ein Thread beim Speichern eines Bildes hängt. Die cv2.imwrite Methode also z.B. 60ms dauert, dass es immer noch genügend andere Threads gibt, die die Bilder abspeichern können, die als nächsten von der Kamera geliefert werden. So ist gewährleistet, dass kein Datensatz verloren geht. Um diesen Thread-Pool aufzusetzen werden lediglich mehrere Threads erstellt, denen die task Methode aus Code 3.8 übergeben wird.

```
1 def start(self):
2     print("[TrainingDataSaver] starting threads: ",self.threads)
3     for _ in range(0, self.threads):
4         t = Thread(target=self.task)
5         t.start()
6         self.threadList.append(t)
```

Code 3.9. Starten mehrerer Threads für den Thread-Pool

Die Variable self.threads aus Zeile 3 gibt die Anzahl der Threads vor. Standardmäßig wurden in diesem Projekt 7 Threads verwendet. Die erstellten und gestarteten Threads werden zum Schluss, in Zeile 6, in eine Liste geschrieben, um später noch auf diese zugreifen zu können, wenn diese beendet werden sollen.

3.5 LaneNavigator

Zu dieser Klasse gibt es nicht viel zu berichten, da sie sich auf das trainierte Machine Learning Model stützt, welches im nächsten Kapitel erläutert wird. Grundsätzlich besitzt diese Klasse nur eine Methode, predictSteeringAngle (vgl. Abbildung 3.1). Diese Methode bekommt das von der Kamera ausgelesene Bild übergeben, und liefert einen PWM Wert zurück, welcher direkt an den Lenkservo weiter übergeben werden kann. Die Bezeichnung Black Box trifft es in diesem Fall also ziemlich genau. Erwähnenswert wäre noch, dass diese Methode normal auf dem main Thread läuft.

3.6 CarControl

Die letzte Softwarekomponente die noch fehlt, ist die Ansteuerung der Hardware. Mit dieser Hardware ist der Lenkservo und der Fahrtenregler gemeint. Da im Hardware Kapitel bereits erklärt wurde, wie PWM bei diesen Komponenten funktioniert und was die Timings sind, wird in diesem Abschnitt nicht mehr näher drauf eingegangen. Nun soll aber ein PWM Signal generiert werden, anstatt eines zu messen. Um das Signal zu generieren, gibt es am Raspberry Pi extra GPIOs, die PWM geeignet sind. Mittels der RPi.GPIO [1] Library, werden die Pins in dieser Klasse gesteuert. Vorerst müssen die GPIOs allerdings richtig konfiguriert werden.

```
1 GPIO.setmode(GPIO.BCM)
2 GPIO.setup(self.steering_pin, GPIO.OUT)
3 GPIO.setup(self.throttle_pin, GPIO.OUT)
4
5 self.steering = GPIO.PWM(self.steering_pin, self.pwm_frequency)
6 self.throttle = GPIO.PWM(self.throttle_pin, self.pwm_frequency)
```

Code 3.10. Konfigurieren der GPIOs für die PWM Signal Generierung

In Zeile 1-3 werden erstmal die Pins, an denen der Lenkservo und Fahrtenregler angeschlossen ist, als Ausgang konfiguriert. In diesem konkreten Fall ist steering_pin=18 und throttle_pin=12, da dies PWM Pins sind. In Zeile 5-6 werden diese Pins dann als PWM konfiguriert. Die pwm_frequency entspricht dabei 50Hz. Mit self.steering.start() wird dann angefangen ein kontinuierliches PWM Signal zu erzeugen. Um nun die Pulsweite anzugeben, wird die Methode ChangeDutyCycle verwendet.

```
1 def steer(self, angle):
2     self.steering_value = self.__relativePWM(angle)
3     if 5.0 <= self.steering_value <= 10.0:
4         self.steering_value += self.steering_trim
5         self.steering.ChangeDutyCycle(self.steering_value)
6 def accelerate(self, throttle):
7     self.throttle_value = self.__relativePWM(throttle)
8     if 5.0 <= self.throttle_value <= 10.0:
```

```
9           self.throttle_value += self.throttle_trim
10          self.throttle.ChangeDutyCycle(self.throttle_value)
```

Code 3.11. Senden des Lenk und Motorwertes

Um die Hardwarekomponenten aus dem Main-Programm audzurufen gibt es zwei Methoden. steer und accelerate, sprich lenken und beschleunigen. Beim Lenken, wird der angle als PWM Wert zwischen 1000-2000 angegeben. Da die GPIO Library allerdings die PWM Werte als relative Werte haben will, die Methode sie aber als absolute Zeitwerte bekommt, müssen diese vorerst umgerechnet werden. Das macht die Methode relativePWM in Zeile 2 und 7 in Code 3.11.

```
1       def __relativePWM(self, pwm_value):
2           return pwm_value*self.pwm_frequency/10000.0
```

Code 3.12. Umrechnung der absoluten in relative PWM Werte

Nach der Umrechnung in relative Werte wird in Code 3.11 überprüft, ob die Eingabewerte im richtigen Intervall liegen (vgl. Zeile 3 & 8). Ist dies der Fall, so wird noch eine eventuelle Trimmung hinzugefügt, die man durch Testfahrten schätzen kann. Zum Schluss wird dann die ChangeDutyCycle Methode aufgerufen, bei der der relative PWM Wert angegeben wird. Dieser Wert wird solange gesendet, bis der Duty Cycle mit einem neuen Wert geändert wird. Da PWM Werte mit 50Hz laufen und das Main-Programm mit 30Hz angepeilt wird, bleibt ein Werte sowieso für meist nur eine ganze Periode bestehen.

4 Machine Learning Model

4.1 Architektur des CNN

In diesem Kapitel geht es um die Architektur des Faltungsnetzes, welches dem Auto ermöglichen soll, eigenständig zu entscheiden welcher Lenkwinkel für die aktuelle Situation richtig ist. Als Eingang dient dabei ein Bild der Frontkamera. In diesem sollen bestimmte Merkmale extrahiert werden, um eventuell die Fahrbahnmarkierungen zu erkennen. Der Output des Netzes soll der Lenkwinkel in Form eines PWM Wertes sein. Es handelt sich hierbei also um eine Regression anstatt einer Klassifikation, da ein kontinuierlicher Wert vorhergesagt werden soll. Wie in der Einleitung erwähnt, wird in diesem Projekt das von NVIDIA getestete CNN verwendet.

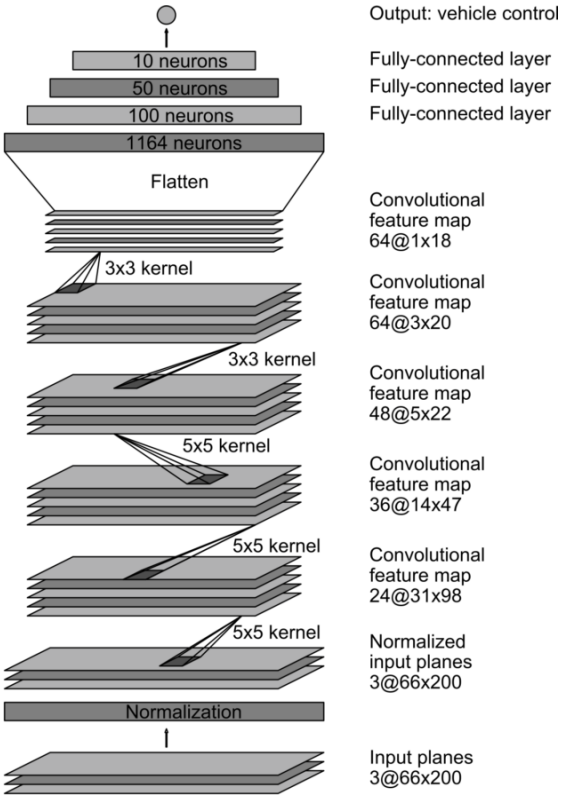

Fig. 4.1. CNN Architektur des von NVIDIA entwickelten Neuronalen Netzes
Quelle: https://developer.nvidia.com/blog/deep-learning-self-driving-cars/l

Das gezeigte Netz verfügt insgesamt über 9 Layer. 5 davon sind Faltungslayer und 3 sind Dense Layer. Die Faltungslayer sind dafür konzipiert eine feature extrac-

tion durchzuführen, als gewisse Merkmale im Bild zu finden. Die darauf folgenden Dense Layer sind der Controller Part, welche aus den erzeugten Feature maps den Lenkwinkel erzeugen. Da sich das Netz allerdings selber antrainiert, kann es klare Trennung zwischen Controller und Merkmalserkennung nicht gemacht werden, sondern nur angedeutet werden.

```
1  def nvidia_model():
2      model = Sequential(name='Nvidia_Model')
3
4      # Convolution Layers
5      model.add(Conv2D(24, (5, 5), strides=(2, 2),
6          input_shape=(66, 200, 3), activation='relu'))
7      model.add(Conv2D(36, (5, 5), strides=(2, 2), activation='relu'))
8      model.add(Conv2D(48, (5, 5), strides=(2, 2), activation='relu'))
9      model.add(Conv2D(64, (3, 3), activation='relu'))
10     model.add(Conv2D(64, (3, 3), activation='relu'))
11
12     # Fully Connected Layers
13     model.add(Flatten())
14     model.add(Dense(100, activation='relu'))
15     model.add(Dense(50, activation='relu'))
16     model.add(Dense(10, activation='relu'))
17
18     model.add(Dense(1))
19
20     optimizer = Adam(lr=1e-3) # lr is learning rate
21     model.compile(loss='mse', optimizer=optimizer)
22
23     return model
```

Code 4.1. Implementierung des CNN in Keras

Um das gesamte Netzwerk zu implementieren und zu trainieren, wird in diesem Projekt Keras verwendet. Code 4.1 zeigt den Aufbau in Keras. Als Aktivierungsfunktion wird Relu verwendet. Obwohl eine Funktion wie Elu eventuell besser geeignet wäre, da sie das dying Relu Problem verhindert, allerdings soll das Modell in einem späteren Schritt noch optimiert werden. Dafür soll das Modell in einen Tensorflow lite umgewandelt werden, wo es allerdings nur die Relu Funktion bisher gibt. Als Fehlerfunktion wird in diesem Netz der mean squared error verwendet, da es sich um ein Regressionsproblem handelt, und der mse dort besser für geeignet ist. Zudem wird noch der Adam optimizer verwendet, um die Backpropagation zu verbessern.

Bevor ein Bild allerdings ins Netz gegeben wird, muss es vorerst Normalisiert werden. Bei NVIDIA ist die Normalisierung im Netz abgebildet, in diesem Projekt wird sie allerdings extern in einer Vorverarbeitung gemacht. Diese Vorverarbeitung ist auch deshalb nötig, da das CNN ein Eingangsbild von 200x66 erwartet, die PiCamera allerdings ein Bild von 640x480 zurück gibt. Ebenfalls empfiehlt

NVIDIA das YUV Farbmodell zu verwenden, anstatt das Standard übliche RGB. Diese Umwandlung wird ebenfalls in der Vorverarbeitung stattfinden.

```
1 def __preprocess(self,image):
2     height, _, _ = image.shape
3     hheight = int(height/2)
4     image = image[-hheight:,:,:]
5     image = cv2.cvtColor(image, cv2.COLOR_RGB2YUV)
6     image = cv2.resize(image, (200,66))
7     image = image / 255 # normalizing
8     return image
```

Code 4.2. Vorverarbeitung, wie sie in der LaneNavigator Klasse vorkommt

In Zeile 2-5 wird das Bild von 640x480 Pixeln auf 640x240 Pixeln reduziert, damit das Verhältnis zu den 200x66 ungefähr passt. Da die Kamera auf die Fahrbahn zeigt und die obere Hälfte unwichtig ist für das Netz, wird die obere Hälfte weggeschnitten. Abbildung 4.2 macht dies deutlich.

Fig. 4.2. Bild der Fahrbahn von der Frontkamera des Fahrzeuges (640x480)

In Zeile 6 von Code 4.2, wird das Farbmodell mit Hilfe der OpenCV Methode umgerechnet. Anschließend wird das Bild auf 200x66 verkleinert und normalisiert. Die Farbwerte befinden sich also im Bereich von 0..1 anstatt im Bereich von 0..255. Ebenfalls werden die Labelwerte, sprich die Lenkwerte auf den selben Bereich normalisiert. So gibt das Neuronale Netz ebenfalls Werte von 0..1 raus, diese kann man dann aber wieder in den PWM Wertebereich von 1000-2000 umrechnen.

4.2 Black Dot Versuch

Bevor es direkt an die autonome Spurhaltung geht, soll erst einmal die komplette Funktionsweise der Machine Learning Architektur zusammen mit der Software

und Hardware getestet werden. Der erste Versuch ist es, dass die Lenkung einem schwarzen Punkt auf Papier folgenden kann. Dabei wird ein schwarzer Punkt auf ein Blatt Papier gedruckt und dieses Blatt Papier wird während, sich das Auto im Trainingsmodus befinden, vor die Kamera gehalten und nach links und rechts bewegt. Parallel wird die Lenkung entsprechend der Bewegung des Papiers geändert. So soll das Auto im nächsten Schritt lernen, diesem schwarzen Punkt selbstständig und präzise zu folgen. Nachdem 4570 Bilder aufgenommen wurden, wurde dieses Netz auf einem Google Cloud Computing Rechner mit Tesla Grafikkarten trainiert. Dabei wurde der Trainingssatz unterteilt in 80% Trainingsdaten und 20% Testdaten.

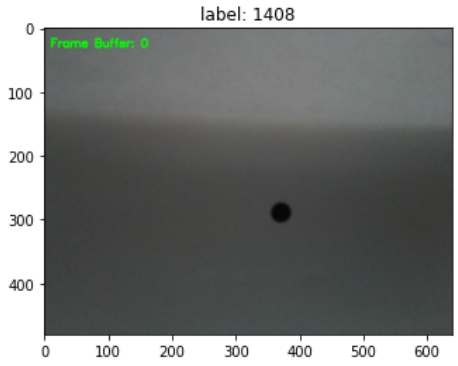

Fig. 4.3. Beispiel Bild aus dem Black Dot Trainingssatzes

Der erwartete Output für Abbildung 4.3 ist also 1408. Nachdem das Machine Learning Model in 70 Epochen trainiert wurde, ergab sich folgende Fehlerkurve.

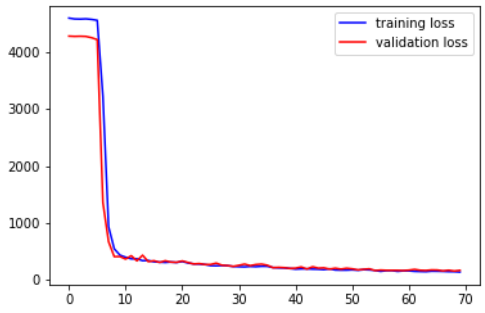

Fig. 4.4. Fehlerkurve nach 70 Epochen des Black Dot Trainingssatzes

Die Accuracy war zu diesem Zeitpunkt bei 93%, was ausreichend ist. Allerdings bezog sich dies auf die Testdaten, die bereits im Training verwendet wurden, um das Netz zu testen und dadurch implizit dem Netz bekannt sind. Bei ganz neuen Bildern war das Netz ziemlich ungenau, den richtigen Lenkwinkel vorherzusagen. Es gab Teilweise Abweichungen von 50%. Man kann also sagen, dass die Accuracy nicht immer ein aussagekräftiger Wert ist. Der Grund warum dieses Netz bei diesem Versuch so schlecht war, ist wahrscheinlich darauf zurück zu führen, dass der Trainingssatz so schlecht war. Der Lenkung wurde mit einer Hand bedient, während die andere Hand das Blatt Papier vor der Kamera bewegt hat. So kann es öfters vorgekommen sein, dass für die selbe x Position des schwarzen Punktes zwei verschiedene Lenkwinkel vorliegen. Damit weiß das Netz nicht mehr, welcher jetzt der richtige ist und kann somit die Bilder nur auswendig lernen. Um das Problem zu lösen, kam die Idee von rein Computer generierten Bildern. Die Werte Lenkwerte sind so exakt auf die X Position des schwarzen Punktes im Bild abgeleitet. Diese generierten Bilder werden von einem selbstgeschriebenen Python Skript generiert, welches immer ein zufälliges Bild liefert. Zufällig ist dabei die x Position des Bildes, welches umgerechnet auch gleichzeitig der Lenkwinkel ist, und die y Position, da das Papier nicht immer exakt in die Mitte gehalten wird. Zudem ist auch der Radius des Punktes zufällig gewählt, da diese quasi die z Position angibt und diese ebenfalls bei einem Papier vor der Kamera variieren kann. Da es noch verschiedene Lichtverhältnisse gibt, ist die Farbtemperatur aus einem bestimmten Intervall von 5300 bis 9900 Kelvin zufällig gewählt und die Helligkeit ist ebenfalls zufällig gewählt.

```
1  def generateRandomImage():
2      radius = random.randrange(10,25,1)
3      angle = random.randrange(1000,2000,1)
4      x = ((angle / 1000)-1)*640
5      y = random.randrange(120,360,1)
6      temperature = random.randrange(5300,9900,100)
7      brightness_factor = random.randrange(40,100)/100
8
9      img = Image.new('RGB', (640, 480), color = 'white')
10
11     d = ImageDraw.Draw(img)
12
13     d.ellipse(xy=box(x,y,radius), fill=(0,0,0), outline=(0,0,0))
14     #image brightness enhancer
15     enhancer = ImageEnhance.Brightness(img)
16     img = enhancer.enhance(brightness_factor)
17
18     img = addTemperatureToImg(img, temperature)
19     return (img, angle)
```

Code 4.3. Generierung eines Bildes für den neuen Trainingssatz

Um solch ein Bild zu generieren, wurde die PIL Library verwendet. In Zeile 13 wird der schwarze Punkt erzeugt. Da diese Methode zwei Eckpunkte haben

möchte, in der die Ellipse (Kreis) liegt, gibt es eine Hilfsmethode die aus den Koordinaten des Mittelpunktes und dem Radius die Koordinaten liefert, die diese Methode benötigt.

```
1 def box(x,y,radius):
2     ux = (x-radius)
3     uy = (y-radius)
4     lx = (x+radius)
5     ly = (y+radius)
6     return (ux,uy,lx,ly)
```

Code 4.4. Umrechnung der Koordinaten für den schwarzen Punkt

Mit Hilfe dieses Skriptes wurde das Keras Modell komplett neu antrainiert und es wurden dabei nur die generierten Bilder benutzt, also keine echten, die die Kamera aufgenommen hat. Insgesamt wurde das Modell dann mit 6000 solcher Bilder trainiert.

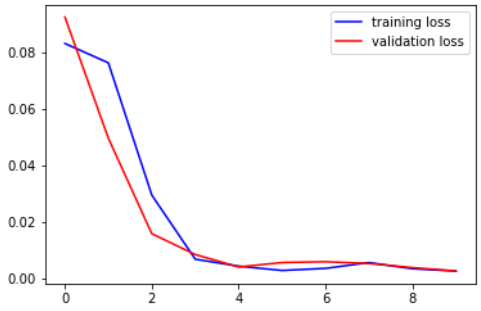

Fig. 4.5. Fehlerkurve des Trainings mit generierten Bildern

Das Modell wurde in 10 Epochen trainiert, wobei nach 4 Epochen sich der Fehler nicht mehr wirklich verkleinert hat. Dieses Modell hatte dann eine Accuracy von 96,79%. Beachtet man jedoch dass es sich um ein Regressionsproblem handelt und kleine Abweichungen vom tatsächlichen Wert keinen merklichen Unterschied macht, ist diese Genauigkeit schon sehr gut. Was allerdings viel interessanter ist, dass dieses Modell im Vergleich zum vorherigen in der Lage ist, einen ziemlichen genauen Lenkwinkel vorherzusagen, zu Bilder, die von der Kamera aufgenommen wurde mit dem Blatt Papier. Dieses Modell hat das Papier noch nie gesehen und dennoch hat es den schwarzen Punkt erkannt.
Was dieser Black Dot Versuch schlussendlich gezeigt hat, war erstens, dass das NVIDIA Modell tatsächlich wie gewünscht funktioniert und zweitens, welch eine Verbesserung Computer generierte Bilder machen, was die Präzision angeht. Denn hat man das Modell die Bilder in Echtzeit auswerten lassen, folgten die Reifen des Modellautos den schwarzen Punkt sehr präzise.

4.3 Optimierung des Modells

Was zu diesem Zeitpunkt noch kritisch am trainierten Modell ist, ist die Inferenz Zeit. Also die Zeit, die das Modell braucht um aus einem Bild den Lenkwert vorherzusagen. Das reine Keras Modell hatte auf dem Raspberry Pi eine Inferenzzeit von etwa 40ms. Bedenkt man die Zeit, die pro Main Schleifen Durchlauf zur Verfügung stehen (ca. 33 ms, siehe Kapitel 3), sind 40ms viel zu lang. Ein erster Optimierungsschritt war es das trainierte Keras Modell in ein Tensorflow Lite Modell umzuwandeln. Dies geschieht relativ einfach mit Hilfe des Tensorflow Lite Converters.

```
1  loaded_model = tf.keras.models.load_model('black_dot_generated.h5')
2  converter = tf.lite.TFLiteConverter.from_keras_model(loaded_model)
3  converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
4
5  tflite_model = converter.convert()
6
7  file = open('black_dot_generated.tflite', 'wb')
8  file.write( tflite_model )
```

Code 4.5. Umwandlung eines Keras Modells in ein Tensorflow Lite Modell

Lässt man die Lane Navigator Klasse (vgl. Kapitel 3.5) mit dem Tensorflow Lite Modell arbeiten, so verkürzte sich die Inferenzzeit auf etwa 30ms im Schnitt. Das ist bereits eine gute Verbesserung, allerdings noch nicht ausreichend. Daher wurde an den Raspberry Pi der Google Coral Stick angeschlossen, welcher es ermöglicht bestimmte Teile der Inferenz, die die Google Edge TPU schneller kann, auszulagern. Um jedoch das Modell auszulagern, muss es durch einen speziellen Compiler durchlaufen. Voraussetzung ist, dass das Modell quantisiert ist auf 8 Bit [2]. Standardmässig arbeitet Keras mit einem float32, so auch das bisherige Modell. Dieses muss aber auf 8 Bit quantisiert werden. Dafür gibt es zwei Möglichkeiten. Die eine ist es, das Modell von Grund auf mit 8 Bit zu trainieren. Die andere ist es, dass Modell bei der Umwandlung von Keras zu Tensorflow Lite zu quantisieren. In diesem Fall wird die zweite Methode verwendet, da das Modell sowieso konvertiert werden muss. So wird aus Code 4.5 folgender Code.

```
1  data_set = tf.data.Dataset.from_tensor_slices((data_valid)).batch(1)
2  def representative_data_gen():
3    for image in data_set.take(100):
4      yield [image]
5
6  loaded_model = tf.keras.models.load_model('black_dot_generated.h5')
7  converter = tf.lite.TFLiteConverter.from_keras_model(loaded_model)
8  converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
9  converter.representative_dataset = representative_data_gen
10
11 tflite_model = converter.convert()
12
```

```
13 file = open('black_dot_generated.tflite', 'wb')
14 file.write( tflite_model )
```

Code 4.6. Umwandlung eines Keras Modells in ein Tensorflow Lite Modell mit 8 Bit Quantisierung

Die 8 Bit Quantisierung findet automatisch statt, sobald man dem Converter einen Datensatz von Eingangsbildern mitgibt (siehe Zeile 9). Dieser Datensatz wird in Zeile 1-4 erzeugt. Es werden dabei lediglich 100 Bilder aus den Testdaten, die auch beim Training verwendet werden, mitgegeben. Diese Testdaten sind durch die Variable data_valid in Zeile 1 dargestellt. Hat man dann das Tensorflow Lite Modell, kann man dieses dem Edge TPU Compiler übergeben, welcher ebenfalls ein Tensorflow Lite Modell erzeugt, allerdings so, dass ein angeschlossener Coral Stick gewisse Aufgaben übernimmt. In dem hier verwendeten Modell können laut Compiler 9 Operationen auf dem Coral Stick ausgeführt werden und 2 werden auf der CPU des Raspberry ausgeführt. Lässt man dieses Modell nun die Inferenz für die LaneNavigator Klasse durchführen, so ergibt sich eine Inferenzzeit von 10ms. Dies ist eine perfekte Zeit für dieses Echtzeitsystem, da es so problemlos mit den gewünschten 30Hz laufen kann und noch genügend Puffer hat, sollte eine Operation mal länger brauchen.

4.4 autonome Spurhaltung

Um nun das Fahrzeug autonom fahren lassen zu können, wurde ein Rundkurs aus Straßenkreide auf eine Straße gemalt. Das Modellauto wurde dann über diesen Rundkurs ferngesteuert und mit 30Hz wurden die Trainingsdaten von der Kamera gesammelt. Es wurden pro Trainingssatz zwei Runden am Stück gefahren. Es existieren 2 Traingssätze im Uhrzeigersinn (nur Rechtskurven) und 2 Trainingssätze gegen den Uhrzeigersinn (nur Linkskurven). Insgesamt gab es 6052 Bilder. Abbildung 4.6 zeigt diesen Rundkurs, der fürs Training verwendet wurde.

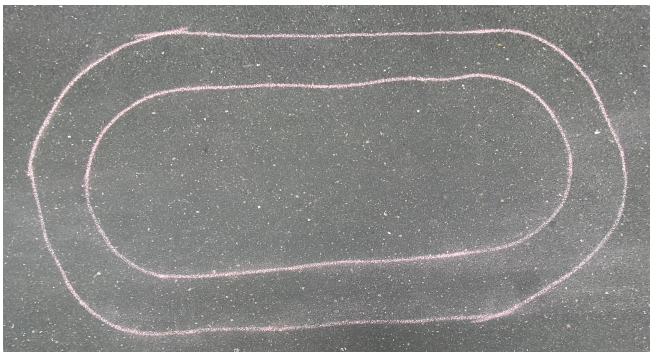

Fig. 4.6. Luftaufnahme des Rundkurses aus Straßenkreide

Das Modell für die Spurhaltung wurde genauso trainiert und Optimiert, wie im vorherigen Kapitel, für den Black Dot Versuch, erklärt. Nur eben mit gesammelten Daten anstatt mit generierten. In diesem Fall sind die gesammelten Daten allerdings ausreichend, da das Fahrzeug eben durch den Kurs gesteuert wurde und so die Lenkwinkel passen müssen. Dieses Modell wurde dann in 8 Epochen trainiert.

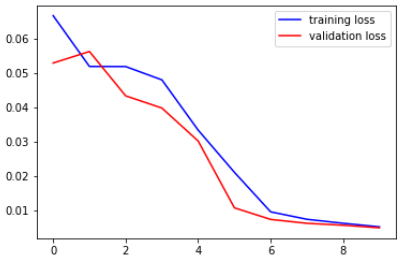

Fig. 4.7. Fehlerkurve des Trainings für die Spurhaltung

Nach diesen 8 Epochen hatte das Modell eine Accuracy von 89%. Nachdem das Modell dann in ein für den Coral Stick optimiertes Modell konvertiert wurde, wurde die LaneNavigator Klasse (vgl. Kapitel 3.5) mit diesem Modell ausgestattet. Das Fahrzeug war dann tatsächlich in der Lage komplett selbstständig über diesen Rundkurs zu fahren. Manchmal fuhr das Auto aus dem Kurs heraus, das lag aber meist an anderen Lichtverhältnissen, bei dem die Linien nicht gut zu erkennen waren auf der Kamera. Ebenfalls war das Fahrzeug in der Lage über Kurse zu fahren, mit denen es nicht trainiert hat. Mit Hilfe einer Python Library [3] konnten die Bilder analysiert werden und sich Heatmaps der Featuremaps anzeigen lassen.

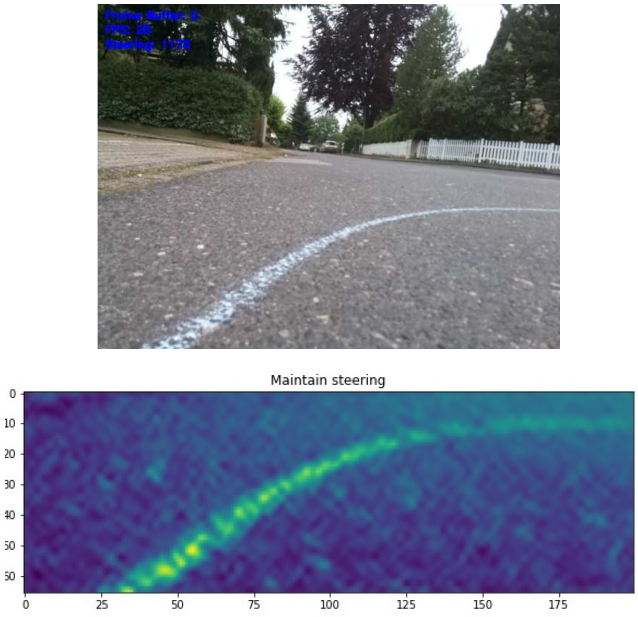

Fig. 4.8. Heatmap eines Bildes aus einer Rechtskurve

Schaut man sich Abbildung 4.8 genau an, sieht man in der Heatmap, wie die Fahrbahnmarkierung grün/gelblich ist. Das bedeutet, dass das Faltungsnetz diese Pixel als aussagekräftig für den Output findet. Man kann also darauf ableiten, dass das Faltungsnetz die Markierung richtig erkennt.
Bei einem weiteren Test wurden dann gestrichelte Markierungen verwendet.

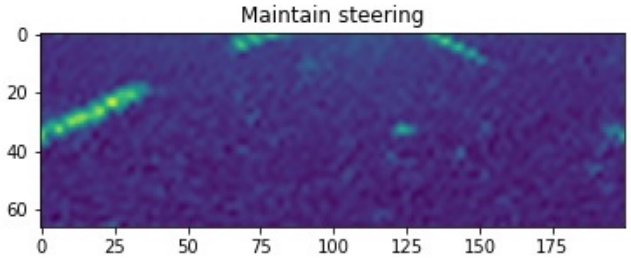

Fig. 4.9. Heatmap eines Bildes von einer gestrichelten Markierung

Der Output bei diesem Bild war 1466, sprich geradeaus, was man auch erwartet hat, obwohl die Markierung gestrichelt war. In dem Traingssatz kam nie ein Bild vor mit gestrichelter Markierung vor.
Die Stärke des NVIDIA Autos war es, dass es in der Lage war z.B. parkende Autos am Straßenrand als Hindernis zu erkennen und diese quasi als Fahrbahnmarkierung zu nutzen. Da keine parkenden Autos simuliert werden konnten, wurde stattdessen ein Kantstein verwendet.

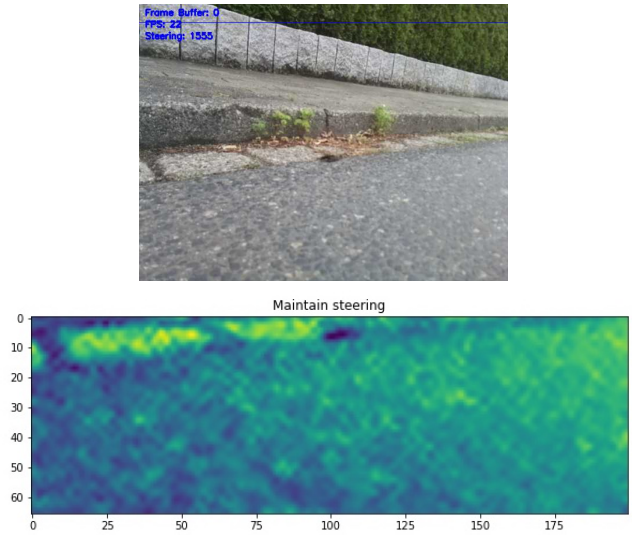

Fig. 4.10. Heatmap eines Bildes von einem Kantstein

Man sieht bereits, dass der rechte Teil des Bildes wohl irgendwas zu sagen hat, obwohl sich dort nichts befindet. Dennoch wurden die Steine erkannt. Zu dem Zeitpunkt des Bildes, war der Output noch geradeaus zu fahren, allerdings

sobald sich das Modellauto den Steinen näherte, lenkte er nach rechts ein und nutze den Kantstein quasi als Fahrbanmarkierung. Der Versuch zeigte ebenfalls, dass wenn man das Auto schräg auf eine gerade Markierung setze, dass er so einlenkt, dass er wieder parallel zur Linie steht. Durch weitere Versuche wurde dies bestätigt.

5 Fazit

Insgesamt lässt sich dieses Projekt als Erfolg einstufen. Es ist gelungen ein Modellauto rein durch Maschinelle Lernverfahren über verschiedene Kurse fahren zu lassen. Selbst unbekannte Kurse waren kein Problem für das Modellauto. Wenn mal eine Markierung gestrichelt war, wurde das Fahrzeug trotzdem perfekt um die Kurve geleitet. Für das ganze hatten dann bereits 6000 Bilder ausgereicht, was nicht besonders viel ist. Würde man noch mehr Daten sammeln von verschiedensten Situation etc., könnte man das Auto durchaus noch verbessern, sodass auch bei anderen Lichtverhätnissen das Auto navigiert werden kann. Interessant wäre es ebenfalls noch gewesen ein Skript zu schreiben, dass Bilder von Fahrbahnen generieren kann, um die Lenkung noch präziser zu machen, wie im Black Dot Versuch gezeigt wurde. Würde man solche generierten Bilder mit echten Bildern verwenden um ein Modell zu trainieren, wäre es bestimmt durchaus genauer in Kurven. Ebenfalls waren alle Optimierungen die getätigt wurden, erfolgreich, da das Auto am Ende erfolgreich mit 30Hz laufen kann, obwohl doch kleinste Hardware verwendet wurde. Interessant wäre es noch, wenn man die Skalierung nochmal eine Stufe runter treibt, sodass man eventuell ein Modellauto im Maßstab 1:87 autonom, rein durch Maschinelle Lernverfahren, durch einen Kurs fahren lassen kann.

Referenzen

[1] croston (2019). Rpi.gpio 0.7.0. https://pypi.org/project/RPi.GPIO/. Letzter Zugriff am Jul 12, 2020.

[2] Google (2020). Tensorflow models on the edge tpu. https://coral.ai/docs/edgetpu/models-intro/#quantization. Letzter Zugriff am Jul 12, 2020.

[3] Kotikalapudi, R. and contributors (2017). keras-vis. https://github.com/raghakot/keras-vis. Letzter Zugriff am Jul 12, 2020.

[4] Liechti, C. (2017). pyserial api. https://pyserial.readthedocs.io/en/latest/pyserial_api.html. Letzter Zugriff am Jul 12, 2020.

[5] Mariusz Bojarsk, Philip Yeres, A. C. K. C. B. F. L. J. U. M. (2017). Explaining how a deep neural network trained with end-to-end learning steers a car. https://arxiv.org/pdf/1704.07911.pdf. Letzter Zugriff am Jun 15, 2020.

[6] Raspberry_Pi_Ltd (2019). Raspberry pi 4 model b. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0_preliminary.pdf. Letzter Zugriff am Jul 12, 2020.

[7] Sawicz, D. (2002). Hobby servo fundamentals. https://www.princeton.edu/~mae412/TEXT/NTRAK2002/292-302.pdf. Letzter Zugriff am Jun 29, 2020.