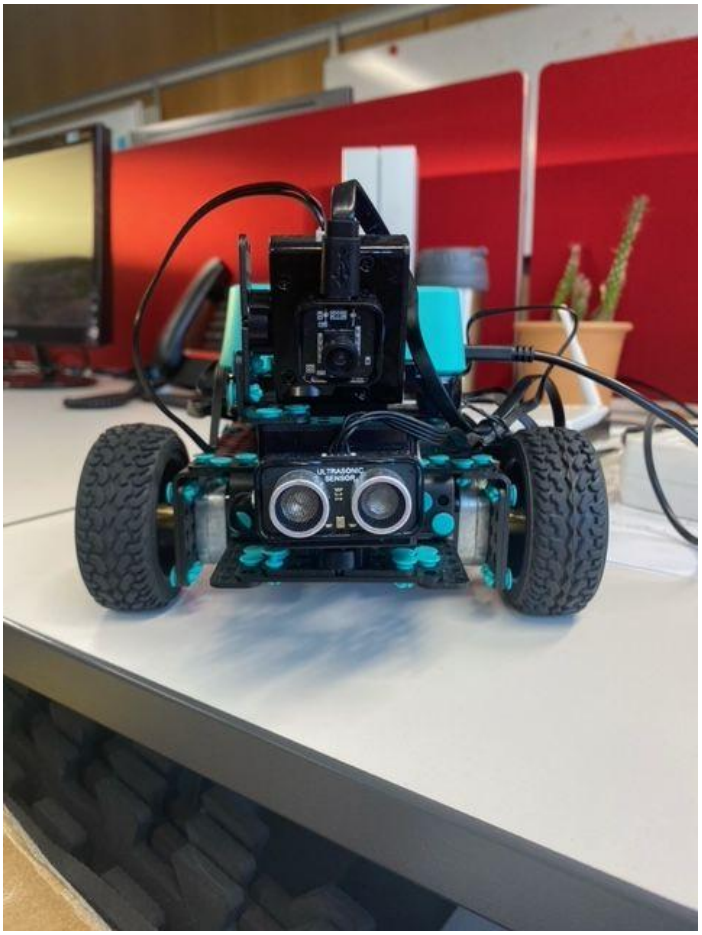
Wall-E: Implementierung eines autonomen Fahrzeugs

Ein Projekt von

Sarah Mauff, Mtrklnr. 40387
(WS 2022 / Studiengang Medieninformatik)

unter Betreuung von

Johannes Theodoridis und **Marcel Heisler**.



Abgegeben am 28.02.2023

Inhaltsverzeichnis

Vorstellung des Projektes

Im Rahmen eines Projektes des IAAI (Institute of Applied Artificial Intelligence) der Hochschule der Medien wurde ein autonomes Fahrzeug entwickelt, welches eigenständig lenkt und auf ausgewählte Verkehrsschilder entsprechend reagiert.
Das Github Repository ist hier verlinkt. Die nötigen Schritte zur Übernahme des Projektes sind in der README zu finden.

Zielsetzung

Das MVP (Minimum Viable Product) für Wall-E wurde folgendermaßen definiert: Wall-E fährt in Duckietown (Schaumstoffstraße) und reagiert auf die jeweiligen Verkehrszeichen. Beispielsweise verringert er in einer 30er-Zone seine Geschwindigkeit.

Dazu wird ein neuronales Netz implementiert, das die Verkehrszeichen erkennt. Der Output dieses Netzes wird dann an Wall-Es Motorsteuerung weitergegeben, welche entsprechend reagiert (z.B. Verringerung der Geschwindigkeit).

Insgesamt werden Stoppschilder, 30er-Zonen Schilder und Ende der 30er-Zonen Schilder erkannt.

Motivation

Ausgangspunkt dieses Projektes war das von einer Studierenden ein paar Jahre zuvor entwickelte Projekt mit dem Namen “Duckietown”. Darin wurde ebenfalls ein autonomes Fahrzeug entwickelt, welches auf seine Umwelt reagiert.

Da sich die Übernahme dieses Projektes als schwieriger darstellte als ursprünglich gedacht, wurde beschlossen das Projekt zu modernisieren und stattdessen die pi-top Umgebung zu verwenden. Die Schilder und Straße stammen noch vom “Duckietown”-Projekt.

Aufbau Wall-E und Entwicklungsumgebung

Wall-E

Der Aufbau des Fahrzeugs wurde zunächst auf den Use-Case von Wall-E angepasst.
Dazu wurde die Kamera an der Vorderseite des Roboters montiert, ebenso wie der Ultraschallsensor, um die Möglichkeit offenzulassen, ihn auch in den Code einzubinden. Ein späterer Einbau hätte eventuell Probleme mit der Schildererkennung gegeben, da der Kamerawinkel nicht mehr gepasst hätte.
Insgesamt besitzt Wall-E vorne zwei Motoren und hinten eine Kugel, welche “mitgezogen” wird. Die beiden Motoren sind jeweils an den Ports M1 (rechts) und

M3 (links) eingesteckt (siehe Abbildungen 1 und 2) und so mit dem Raspberry Pi verbunden.

Abbildung 1: Port M0

Abbildung 2: Port M3

Die Kamera ist mit dem Pi über einen USB-Port verbunden.

Umgebung

Um die pi-top Entwicklungsumgebung nutzen zu können, muss zunächst eine SD Karte geflasht werden. Mithilfe dieser Anleitung von pi-top gelingt dies problemlos.
Anschließend wird nach diesem Beispiel die Internetverbindung des pi-tops konfiguriert. Nach erfolgreicher Durchführung hat man dann die Möglichkeit, entweder Tastatur und Monitor an den pi-top anzuschließen, direkt über die Browseroberfläche zu entwickeln oder eine VNC (Virtual Network Computing) Verbindung zwischen dem Remotecomputer und dem pi-top aufzubauen. In

letzterem Fall wird die Desktop Ansicht des pi-tops auf den Remotecomputer, sowie der Input der Tastatur und Maus des Remotecomputers auf den pi-top übertragen.
Bei der Entwicklung von Wall-E wurde die VNC Verbindung aufgesetzt. Dazu folgt man den Instruktionen unter diesem Link und wählt im nächsten Schritt ‘Another computer’ aus.
Die default Benutzerdaten des pi-tops sind:
Benutzername: pi; Passwort: pi-top.

Damit steht die Verbindung zum pi-top und die Entwicklung kann beginnen. Beispielsweise kann man dann Visual Studio Code verwenden (verlinkt in der Taskleiste der Benutzeroberfläche des pi-top), um als Funktionstest ein Beispielskript aus den pi-top SDK Docs auszuführen. Nach Klick auf den grünen Pfeil rechts oben (Run) reagiert Wall-E dann entsprechend des verwendeten Skripts, wenn alles korrekt aufgesetzt wurde.

Zuletzt benötigt man noch eine Teststrecke - in diesem Fall die Schaumstoffstraße des Projektes “Duckietown”. Es wurden hellgelbe, matte Klebestreifen auf die Strecke geklebt, um die Fahrbahn zu markieren. Genaueres zur Farbwahl wird im Abschnitt Lane Detection erläutert.

Konzeptionierung

Nun wird die Zusammenarbeit der verschiedenen Skripte erläutert (vgl. Abbildung 3).

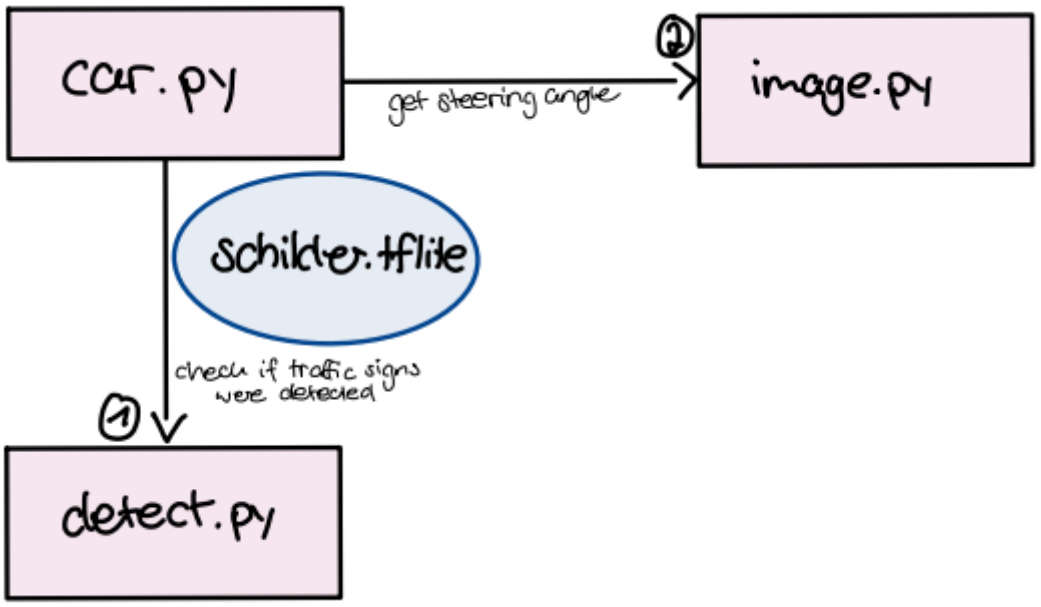


Abbildung 3: Reihenfolge der Skripte

Das Hauptskript ist *car.py*. Dort wird innerhalb einer Loop in Zeile 91 das Skript *detect.py* aufgerufen, welches nach Mitgabe des TensorFlow Modells als Parameter ein Objekt zurückliefert, das alle erkannten Schilder in dem jeweiligen Frame enthält (ein leeres Objekt, falls keine Schilder erkannt wurden). Die Funktion *handleDetection()* in *car.py* setzt dann je nach erkanntem Schild die notwendigen Motorparameter.
Im Anschluss daran wird der Frame dann dem Skript *image.py* weitergereicht, welches nach sukzessivem Ausführen der Funktionen den gewünschten Lenkwinkel zurückgibt.
Zuletzt wird der Lenkwinkel dann in der Methode *turn()* (in *car.py*) weiterverarbeitet. Dort wird dann gecheckt, ob der Winkel um mehr als 15 Grad von der Geraden (90 Grad) abweicht.
Erst falls dies zutrifft, wird dann entsprechend dem Grad des Winkels gelenkt.

Lenken bedeutet in diesem Kontext allerdings, dass die Geschwindigkeit des äußeren Reifens erhöht wird - bei einer Rechtskurve (Winkel kleiner als 90 Grad) wird also der linke Reifen schneller bewegt als der rechte, was dazu führt, dass sich Wall-E nach rechts dreht. Genau so dreht sich der rechte Reifen schneller als der linke, wenn Wall-E eine Linkskurve (größer als 90 Grad) fahren muss.

Sobald die *turn()* Methode fertig bearbeitet wurde, wird dann ein Timer gesetzt. Mit diesem Timer wird jeweils am Anfang des Skriptes *car.py* überprüft, ob genügend Zeit vergangen ist, um einen neuen Frame verarbeiten zu können. Näheres dazu wird im Unterpunkt Fahrlogik beschrieben.

Implementierung

Lane Detection

Bei der Implementierung des Lane Following wurde dieses Tutorial zum Vorbild genommen und einige Anpassungen gemacht.
Die jeweiligen Output Bilder sind in dem jeweiligen Kapitel eingefügt.

Filtern des Farbraums

Zunächst wird der Input Frame nach dem Farbraum gefiltert, in welchem sich die gelben Fahrbahnmarkierungen befinden. Verwendet wird dafür zunächst die Funktion *inRange()* von OpenCV, welche die obere und untere Grenze des gewünschten Farbraums als Parameter im rgb-Format entgegennimmt. Im Anschluss wurde die Funktion *bitwise_and()* von OpenCV verwendet, um den validen Farbraum auf das Bild zu mappen.

Durch das Filtern der Farbe sorgt man dafür, dass irrelevante Teile des Bildes, welche nicht zum Lane Following beitragen, ignoriert werden. Die Bestimmung des Farbraumes gestaltete sich etwas kompliziert, da dieser je nach Lichteinfall

(also Position der Straße im Raum) anders ausfiel. Sobald allerdings stabile Lichtverhältnisse vorherrschen, funktioniert auch die Filterfunktion verlässlich. Die verwendeten Werte in diesem Beispiel sind als unterer Richtwert (20, 50, 100) und als unterer Richtwert (30, 255, 255). Alternative Farbräume wurden im Code belassen, da diese je nach Position der Straße im Raum relevant werden.

Canny Edge Detection

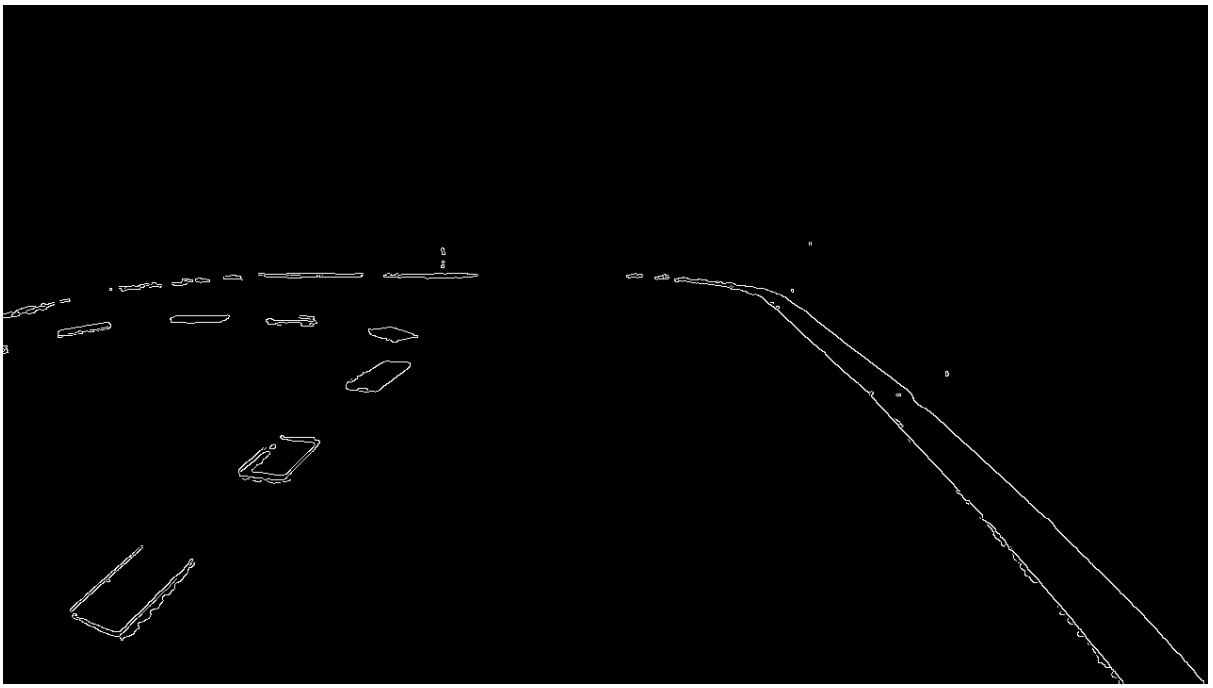


Abbildung 4: Output nach Canny Edge Detection

Im Anschluss an das Herausfiltern irrelevanter Farbinhalte wird der Frame auf Kanten untersucht. Dazu wird die Canny Edge Detection von OpenCV verwendet (vgl. Abbildung 4). Diese akzeptiert als Input Parameter zu dem Bild noch jeweils einen oberen und unteren Threshold (vgl. Abbildung 5), welche beide im Bereich von 0 bis 255 liegen. Zwar werden auch größere Werte als 255 akzeptiert, aber die Ergebnisse sind ähnlich wie bei einem Overflow Error unvorhersehbar.

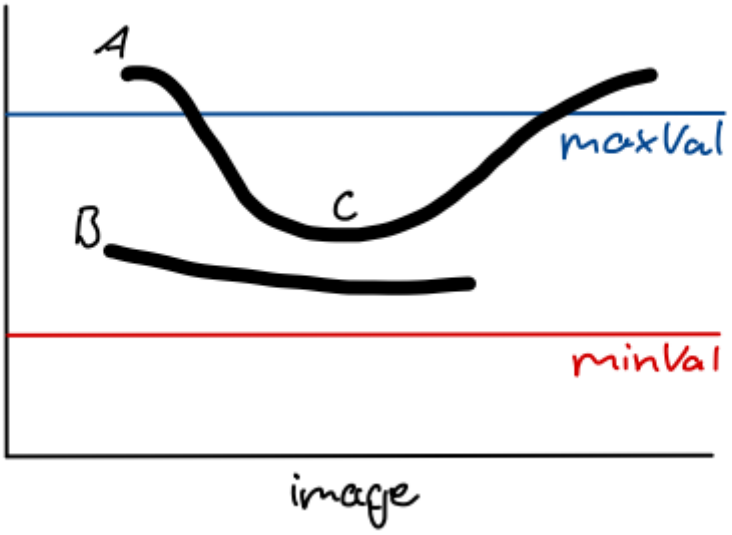


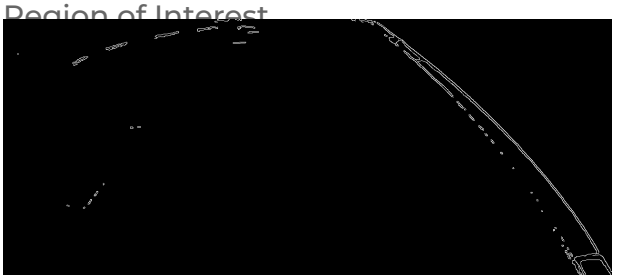
Abbildung 5: Darstellung Threshold Kantenerkennung

Es gilt, dass alle Pixel, welche über dem oberen Threshold liegen, mit Sicherheit Kanten darstellen (hier A). Alle Pixel unter dem unteren Threshold werden dabei nicht als Kanten klassifiziert. Bei allen Pixeln dazwischen kommt es darauf an, ob eine Verbindung zu einer sicheren Kante vorliegt (hier C als Teil einer Kante).

Diese Thresholds müssen optimiert werden, damit spätere Berechnungen nicht durch eine zu feine oder zu grobe Kantenerkennung sabotiert werden. Eine Herangehensweise an die Optimierung der Canny Edge Detection gibt es nicht, da diese je nach Problemstellung anders implementiert werden muss. Allerdings steht in Dokumentationen von OpenCV, dass eine Relation der Thresholds von 2:1 beziehungsweise 3:1 in der Regel gute Ergebnisse liefert. Im Sinne der peniblen Erkennung von Kanten trifft dies auch zu. Jedoch musste die Canny Edge Detection im Fall von Wall-E “schlechter” funktionieren, das bedeutet, dass eine feine Erkennung der Kanten eher unvorteilhaft war.

Da die Oberfläche der Straße, auf welcher Wall-E fährt, aus einer verhältnismäßig starken Struktur besteht, werden bei bestimmten Thresholdwerten sowohl die Teile der Straße, die unmittelbar vor Wall-E liegen, als auch diejenigen, welche seitlich von Licht angestrahlt werden, als Kanten erkannt und verschlechtern so den Erkennungsalgorithmus der Fahrbahn.

Region of Interest



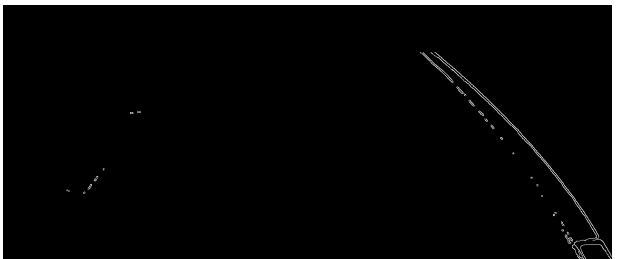


Abbildung 6: vor Canny Edge Detection *Abbildung 7: nach Canny Edge Detection*

Als nächsten Schritt wird dann eine definierte Region of Interest angewendet. Dabei handelt es sich um eine Vorverarbeitung des Frames für die Eckenerkennung. So kann sichergestellt werden, dass Teile des Frames, welche mit Sicherheit nicht zum relevanten Teil der Fahrbahn vor Wall-E gehören (Hintergrund, Wand, Schilder), nicht in die Eckenerkennung mit einfließen.
Im referenzierten Tutorial wird an dieser Stelle ein Dreieck definiert. Im Beispiel von Wall-E allerdings erwies sich ein spitz zulaufendes Viereck hingegen als geeigneter. Die definierten Koordinaten lauten: [(0, height), (width, height), (width, 100), (0, 100)].

Zusammenfassend wird also eine definierte Form aus dem verarbeiteten Frame ausgeschnitten, wie in den Abbildungen 6 und 7 dargestellt.

Bestimmung der Fahrbahnlinien und des Lenkwinkels

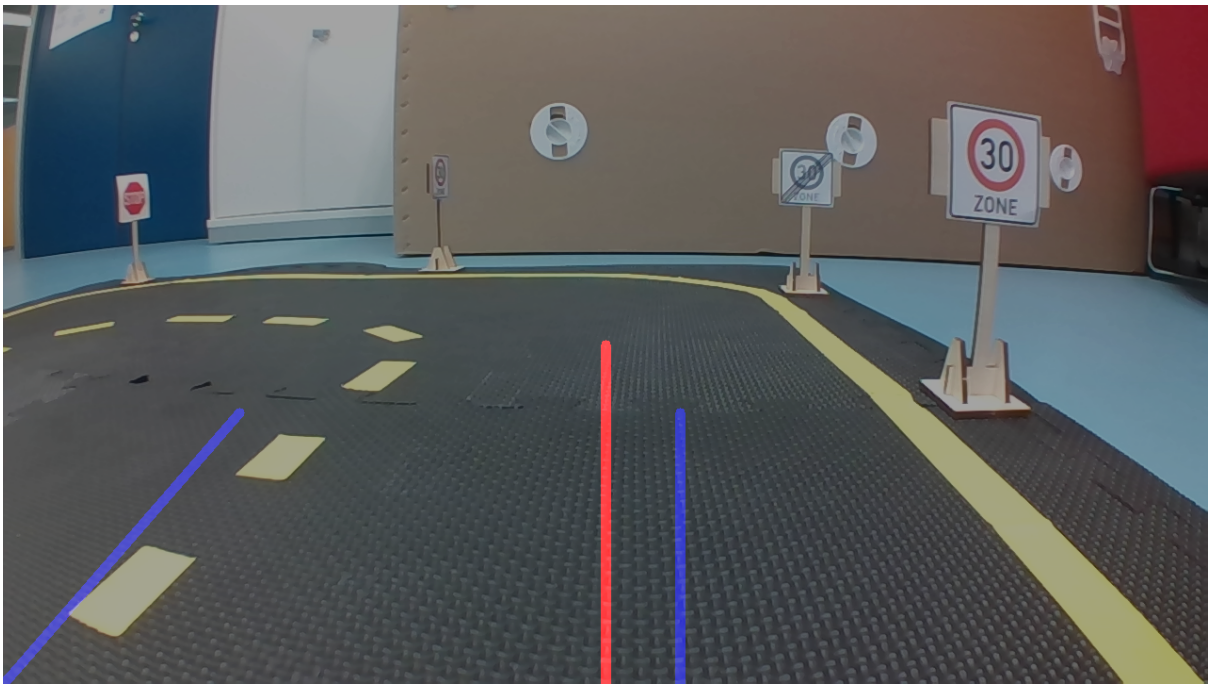


Abbildung 8: endgültiger Output

Im Anschluss daran wird die Funktion *average_slope_intercept()* aufgerufen, um gemäß dem Tutorial die Linien zu bestimmen.
Die Methoden *calculate_offset()* und *calculate_steering_angle()* dienen dazu, den Lenkwinkel (steering angle) zu bestimmen.

In der Abbildung 8 ist der berechnete Lenkwinkel in rot markiert.

Physische Anpassungen

Zusätzlich zur programmatischen Bildverarbeitung wurden auch physische Anpassungen vorgenommen. Zum einen wurde eine Bande gebaut, die gleich zwei Funktionen hat. Sie sorgt dafür, dass eventuelle Gelbtöne im Hintergrund nicht berücksichtigt werden und hilft dabei, die Lichtverhältnisse stabil bzw. stabiler zu halten. Jedoch funktioniert die Lane Detection von Wall-E soweit verlässlich, dass die Bande nur manchmal in der Dämmerung benötigt wird (wenn weniger natürliches Licht vorhanden ist). Zum anderen wurde auch die Position der Kamera optimiert. Dabei ist darauf zu achten, dass Wall-E weit genug 'sehen' kann und dennoch den unmittelbar vor ihm liegenden Teil genau erkennen kann.
An dieser Stelle musste die Implementierung der Region of Interest eventuell überarbeitet werden, da diese direkt mit dem Kamerawinkel zusammenhängt.

Fahrlogik

Grundsätzlich bieten sich verschiedene Arten an, die Wall-E-Motoren anzusprechen.
Zum einen über das Modul "DriveController" und zum anderen direkt über "EncoderMotor".
Der große Vorteil der zweiten Importart liegt darin, dass die Lenkung durch die direkte Ansteuerung des jeweiligen Motors intuitiver war. Die Steuerung nach dem Import über den "DriveController" führte in der Implementierung immer wieder zu Problemen.

Als zweites Problem erwiesen sich die gesammelten Frames. Da die gekoppelten Algorithmen aller Skripte eine Weile brauchen, um vollständig durchzulaufen, muss diese Zeit als Bedingung für das Bearbeiten eines neuen Frames gesetzt werden. In *car.py* wird dies durch die Zeile 82 erfüllt.
Wenn diese Zeile nicht implementiert ist, neigt Wall-E dazu, in Kurven "durchzudrehen". Das bedeutet, dass er am Ende einer Kurve weiter gelenkt hat, obwohl er sich nicht mehr in einer Kurve befand, da er noch alte Frames verarbeitet hat.
Durch das Setzen der 0.7 Sekunden bis zum Laden eines neuen Frames wird also sichergestellt, dass erst dann neue Frames verarbeitet werden, wenn die alten vollständig abgearbeitet sind. Leider führt dies aber auch dazu, dass Wall-E zum Stocken neigt. Der Fehler konnte Stand Wintersemester 2023 noch nicht behoben werden.

Sammeln der Daten

Sobald ein stabiles Lichtverhältnis vorlag und das Lane Following verlässlich funktionierte, konnte man sich dem Sammeln der Trainingsdaten widmen. Es

wurden alle abgefragten Frames in einem Ordner gespeichert. Dabei wurde auf jegliche Variationen geachtet.
Sowohl die Tageszeit, die Reihenfolge und Häufigkeit der Schilder als auch der Hintergrund (mit und ohne Bande) wurden abgewechselt. Dies ergab einen robusten Datensatz (1179 Trainingsbilder und 583 Testbilder), mit welchem man das Neuronale Netz trainieren konnte.
Doch bevor das geschah, mussten noch Bounding Boxen gezogen und gelabelt werden.

Dafür wurde das Tool “labelImg” verwendet.
Sehr empfehlenswert ist an dieser Stelle die Verwendung der verfügbaren Shortcuts: ‘w’ um die Bounding Box zu ziehen bzw. zu triggern, ‘d’ um zum nächsten Bild zu wechseln (bzw. ‘a’ um zu vorherigen Bild zurückzukehren) und nach dem Spezifizieren des Speicherordners über den entsprechenden Button auf der linken Leiste ‘strg + s’ um das gelabelte Bild zu speichern.
Bilder, auf denen kein Schild zu finden war, wurden ignoriert und später über ein Skript entfernt. Da zu jedem Frame mit Bounding Boxen eine .xml Datei erstellt wird, mussten lediglich alle Bilder entfernt werden, welche keine dazugehörige .xml Datei besaßen (Speichername des Frames und der dazugehörigen .xml Datei sind nach Erstellung gleich).

Training

Im nächsten Schritt ging es an die Auswahl eines Neuronalen Netzes. Die Entscheidung fiel auf ein vortrainierstes Modell von TensorFlow Lite, da dieses sowohl weniger Speicherplatz benötigte und so einfach auf den Raspberry Pi übertragen werden konnte, als auch das Training einfacher gestaltete, da die Optimierung der Architektur auf Object Detection nicht selbst übernommen werden musste.
Für das Training wurde dieses Tutorial von TensorFlow befolgt und auch das “efficientdet_lite0” Modell verwendet.
Das Training selbst lief, wie auch im Tutorial gezeigt, über “Google Colab”, nachdem allein die Installation der notwendigen Packages auf einem MacBook Pro von 2020 zeitlich unbegrenzt anzudauern schien. Die Implementierung auf Google Colab verlief per se sehr intuitiv, da eine hohe Ähnlichkeit zu Jupyter Notebooks besteht.
Lediglich das File Management war anfangs etwas kompliziert. Nachdem allerdings ein Google Drive über die Schaltfläche auf der linken Seite verknüpft wurde, verlief auch das harmonisch.
Letztendlich ergab das Training vor dem Export des Modells eine Average Precision von circa 0.7 und nach dem Export von 0.69. Aus verschiedenen Gründen wurde die Precision durch den Export minimal verringert, was aber normal ist.

Ergebnis

Das MVP von Wall-E wurde erfüllt.
Lediglich neigt Wall-E dazu zu stocken, was, wie bereits beschrieben, dem Framehandling geschuldet ist.

Lessons learned und Fazit

Das Projekt während des Praxissemesters durchzuführen hatte definitiv seine Nachteile. Vor allem wenn es im Betrieb stressiger wird, ist es schwer, beides unter einen Hut zu bekommen.
Nichtsdestotrotz bleibt als größter Tipp festzuhalten, dass man einfach dranbleiben muss. Am Ende wird es mit Sicherheit funktionieren.
Auch ist es ratsam, frühzeitig mit dem Sammeln der Bilder zu beginnen.

Alles in allem ist es ein Projekt, welches sehr zu empfehlen ist.