WRO Saison 2024- Future Engineers

Team: Lego Sapiens

Heisenberg-Gymnasium Bruchsal

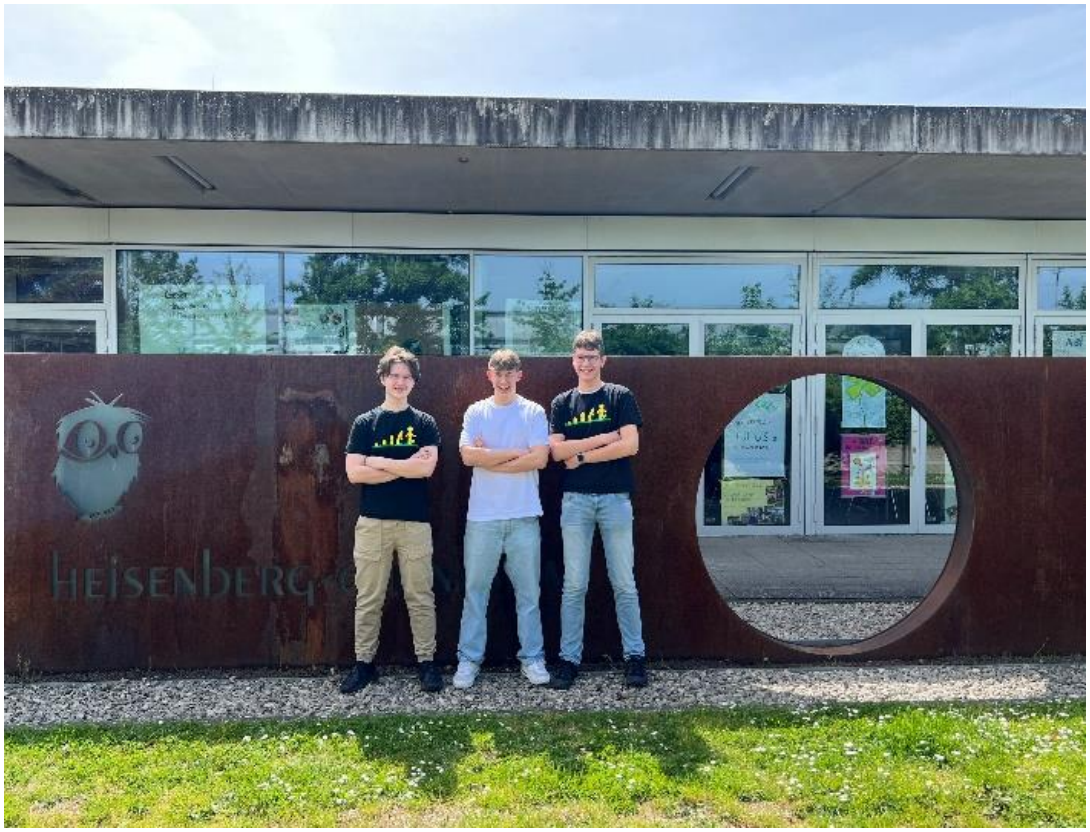
Team-Mitglieder:

Richard Schütz

Malte Eikmeier

Emil Petini

Coach: Jenny Schütz



Inhalt

Einleitung und Motorisierung

Überblick

Das Team Lego Sapiens nimmt zum ersten Mal am Wettbewerb WRO Future Engineers teil. Zielsetzung ist, sich in diesem Projekt auch mit den praktischen Anwendungsmöglichkeiten künstlicher Intelligenz zu befassen.

Es sollte keiner der angebotenen Fertigbausätze zur Teilnahme an der WRO verwendet werden. Da das Fahrzeug von Grund auf neu konzipiert wurde, waren mehrere Fahrzeuggenerationen zur Entwicklung notwendig.

Mit dem vorgestellten Fahrzeug wird der Nachweis erbracht, dass die Aufgabenstellung von WRO Future Engineering 20204, also des autonomen Fahrens im Hindernisparcours, mit einem für Edge Computing geeigneten, einfachen neuronalen Netzwerk (1D CNN) lösbar ist. Uns ist klar, dass die Aufgabenstellung grundsätzlich auch mit weniger Aufwand lösbar ist. Der Vorteil der KI liegt in den sehr ausgeglichenen Fahrkurven und des quasi weitsichtigen Fahrverhaltens wie bei einem menschlichen Fahrer.

Fahrzeugkonzept

Bei der Erfassung der Umgebung setzen wir im Wesentlichen auf einen LIDAR-Scanner. Der große Vorteil eines LIDAR-Scanners liegt in der schnellen Erfassung der Konturen der kompletten Umgebung in Form von Abstandsdaten bis zu etwaigen Wänden oder Hindernissen. Ein weiterer Vorteil des LIDAR-Scanners ist die völlige Unabhängigkeit von der Beleuchtungssituation des Spielfelds.

Der Nachteil des LIDAR Scanners ist die fehlende Farbinformation: Im Hindernisrennen müssen rote und grüne Hindernisse voneinander unterschieden werden. Mittels zweier Open MV H7+ - Kameras kann diese Information bereitgestellt werden:

Die Kombination der LIDAR-Scanner- und der Farbinformation in einem Datensatz erlaubt dem verwendeten KI-Modell, zu jedem Messzeitpunkt einen geeigneten Lenkbefehl für das Fahrzeug zu errechnen. Dazu reicht die verfügbare Zeit (0.1 s) zwischen zwei LIDAR-Scans aus.

Die Geschwindigkeit des Fahrzeugs regeln wir mit einer Schlitzscheibe und einer Lichtschranke am rechten Hinterrad: Wir messen die Umdrehungsgeschwindigkeit. Die Regulierung der Motorleistung erfolgt mit dem hierfür einschlägigen PID-Algorithmus. Die Geschwindigkeit wird im Eröffnungsrennen automatisch höher eingestellt als beim Hindernisrennen, weil weniger enge Kurven zu fahren sind, in denen mehr Sensorinformationen benötigt werden.

Erwähnenswert ist auch die verwendete, hochflexible Softwarearchitektur: Das Steuerprogramm ist auf mehrere asynchron und parallel laufende Prozesse verteilt, die über Nachrichten kommunizieren. Als Basis dafür wird die Open Source Software ROS2 (Robotic Operating System) eingesetzt.

Die Entwicklung und Tests erstreckten sich über einen Zeitraum von über einem Jahr. Zur Vorbereitung auf den Wettbewerb wurden umfangreiche Probefahrten zum Training des neuronalen Netzes durchgeführt. Während der Probefahrten wurden auch die bereitzustellenden Demovideos gedreht.

Bedienungsanleitung

Die Bedienung erfolgt am Wettkampftag über drei Elemente:

1. In einem der **USB-Ports** des Raspberry PI Bordrechners befindet sich ein **Dongle**. Ist dieser eingesetzt, befindet sich der Roboter im Betriebsmodus Eröffnungsrennen, ist dieser

entfernt, im Betriebsmodus Hindernisrennen. In der Bauphase zwischen Eröffnungs- und Hindernisrennen muss der Dongle entfernt werden.

2. Das **Einschalten** des Roboters erfolgt nach Einsetzen des Akkus über einen **Kippschalter** (in Fahrtrichtung vorne rechts). Der Bordrechner bootet nach dem Einschalten selbsttätig und startet ROS2 sowie das Steuerprogramm. Dieser Vorgang dauert ca. eine Minute. Im OLED-Display auf der Fahrzeug-Oberseite wird nach Herstellung der Einsatzbereitschaft „Ready." ausgegeben. Neben dem OLED-Display befindet sich eine kleine Digitalanzeige zur Kontrolle der Akkuspannung. Bei vollgeladenem Akku wird eine Spannung von 12.4 V angezeigt, ab einer Spannung von weniger als 10.5 Volt muss der Akku ausgewechselt werden.
3. Nach Herstellung der Einsatzbereitschaft und Positionierung des Roboters auf dem Spielfeld erfolgt der **Start** des Renn-Modus mit dem **Touch-Button** neben dem OLED-Display. Der Roboter verharrt noch einige Sekunden, bis er losfährt und beschleunigt (wegen des langsamen Hochfahrens des PID-Controllers). Durch nochmaliges Betätigen des Touch-Buttons kann das Rennen vorzeitig abgebrochen werden. Am Ende des Rennens löst das Steuerprogramm den Shutdown des Bordrechners aus. Dann kann die Abschaltung mittels Kippschalter erfolgen. (Die Betätigung des Kippschalters im Betrieb kann ggf. zu Inkonsistenzen im Dateisystem führen!)

In der Wettkampfumgebung ist der Roboter per Funk nicht erreichbar. In der Test- und Trainingsumgebung zuhause steht für die Steuerung des Roboters bei den Trainingsfahrten ein Xbox-Controller zur Verfügung. Dieser verbindet sich via Bluetooth mit dem Bordrechner des Roboters. Zusätzlich kann zu Wartungszwecken ein Login via SSH in den Bordrechner im verschlüsselten WLAN der Test- und Trainingsumgebung erfolgen. Zur Datensicherung, zum Update des Steuerprogramms und zum Einspielen der KI-Modelle / Steuerdaten wird ein NFS-Mount verwendet, der ebenso nur im Test- und Trainings-WLAN zur Verfügung steht.

Motorisierung

Zum Antrieb wird ein bürstenloser Motor vom Typ „Castle 10th Scale SCT 1410 Sensored Motor - 3800KV" eingesetzt (Abbildung 1: Fahrmotor und Lenkservo). Grund der Auswahl des Motors sind das hohe Drehmoment bei niedriger Geschwindigkeit und die Vermeidbarkeit des „Cogging" (Ruckeln) durch die Nutzung der Hall-Sensorschnittstelle des ESC. Dies erlaubt sanftes Anfahren und extreme Langsamfahrt beim Parkmanöver. Zusätzlich wurde die Getriebeübersetzung des Kits durch Wahl eines alternativen Ritzels auf 3.5:1 erhöht.

Der Antrieb erfolgt über vier Räder und einen Riemenantrieb. An der Hinterachse steht ein Differential zur Verfügung. Damit sind gute Traktion und Manövrierbarkeit gegeben.

Die Lenkung erfolgt an der Vorderachse mittels eines Lenkservos. Zum Einsatz kommt ein „Miuzei 20KG Servo Motor High Torque Digital Servo Metal Waterproof 1/10 Scale Servo for R/C Car Robot, DS3218 Control Angle 270°". Dieser Servo ist ausreichend schnell und hat eine hohe Stellkraft (20kg). Wichtig ist, dass der Lenkeinschlag groß genug ist, um enge Kurven fahren zu können.

Abbildung 1: Fahrmotor und Lenkservo

Energie und Sensoren

Die Energieversorgung des Fahrzeugs erfolgt mit einem 3S-LiPo-Akku mit einer Nominalspannung von 11.1 V sowie einer Kapazität von 450 mAh. Die Kapazität des vollgeladenen Akkus reicht aus, um damit mindestens ein Rennen zu absolvieren. Ausschlaggebend für die Wahl des Akku-Typs war neben der erforderlichen Kapazität ein möglichst niedriges Gewicht (45g), um das Maximalgewicht von 1500 g für den Roboter einhalten zu können. Zur Sicherheit bringen wir zum Rennen zwei geladene Reserve-Akkus mit, welche in der Bauphase gewechselt werden können. Die Spannung des Akkus lässt sich mit einem kleinen Digitalmultimeter an der Oberseite des Roboters kontrollieren. Sie sollte 10.5 V nicht unterschreiten.

Um mit dem Akku können sowohl der Fahrmotor (12V), der Steuerservo (3-5V) sowie die Steuerungselektronik (5V) betreiben zu können, wird als Spannungswandler ein „LM2596 DC-DC Abwärtswandler“ mit LED-Anzeige genutzt

Als ESC zur Motorsteuerung kommt ein „RC Auto HobbyWing QuicRun 10BL120 120A“ zum Einsatz. Der Lüfter wurde zur Gewichtsersparnis entfernt. Die kurze Wirkbetrieb beim Rennen kann nicht zu einer Überhitzung führen. Mit dem zusätzlichen Hallsensor-Anschluss kann Ruckeln (Cogging) bei Langsamfahrt vermieden werden

Die Anbindung von Steuerservo und ESC erfolgt mit dem „SUNFOUNDER PCA9685 16 Channel 12 Bit PWM Servo Driver“. Dieser wird an die I2C-Busschnittstelle des Raspberry Pi angeschlossen.

Zur Erleichterung des Anschlusses der Sensoren und Aktoren an den Raspberry Pi wird das „FREENOVE Breakout Board for Raspberry Pi, Terminal Block HAT mit GPIO Status LED“ genutzt (Abbildung 2).

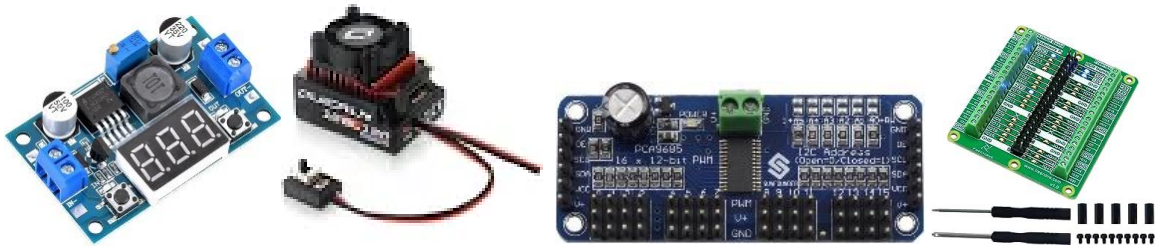


Abbildung 2: Spannungswandler, ESC, I2C-Schnittstelle, Sense Hat Breakout Board

Verwendete Sensoren

Zur Erfassung der Umgebungsparameter werden diverse Sensoren eingesetzt (Abbildung 3: LIDAR-Scanner, Open MV H7+ Kamera, Sense Hat, Geschwindigkeitsmesser, Touch Button):

- 1D Lidar Scanner 10 Hz mit 30 Meter Reichweite und 3240 Datenpunkten im Vollkreis mit Ethernet-Anschluss: „SLAMTEC RPLIDAR S2E 360° LASER SCANNER (30 M)“ zur Erkennung der Fahrbahnbegrenzungen und Hindernisse. Der LIDAR-Sensor ist über Kopf verbaut, um die Erfassungshöhe um ca. 2 cm zu erniedrigen. Zusätzlich ist die Hinterachse des Fahrzeugs um ca. 3mm erhöht: So zielen die Laserstrahlen leicht abwärts und treffen immer die 10 cm hohen Begrenzungswände des Spielfelds sowie die Hindernisse. (Hinweis: Zunächst wurde mit einem einfacheren Lidar vom Typ Lidar RPLIDAR A1 ROS S1 mit seriellem Anschluss experimentiert. Leider war dieser nicht genau genug.)

- Zwei Open MV H7+ Kamerasysteme mit Fisheye-Objektiv: Diese Kamerasysteme verfügen über einen integrierten Chip, der die Ausführung von MicroPython-Programmen erlaubt. Damit kann die Bildverarbeitung direkt auf den Kameramodulen erfolgen. Um Hindernisse möglichst früh für eine vorausschauende Steuerungsentscheidung sowohl bei Fahrten im Uhrzeigersinn als auch im Gegenuhrzeigersinn erkennen zu können, sind die Kameras jeweils um 45 Grad zur Fahrtrichtung zur Seite geschwenkt. Die Bilder werden schon im Kameramodul nach aufrechtstehenden Rechtecken in roter oder grüner Farbe durchsucht. Die Kameras schicken nur die X-Koordinaten der gesehenen Hindernisse an den zentralen Bordrechner. So lässt sich eine Frame-Rate von bis zu 25 fps erreichen. Da der LIDAR-Scanner nur mit 10 Hz arbeitet, ist dies mehr als ausreichend. Die Erkennung von Farben ist generell nur bei ausreichenden Lichtverhältnissen möglich. Deshalb brachten wir in der Schlussphase der Konstruktion zusätzliche Frontscheinwerfer (ultrahelle weiße LEDs aus einer Taschenlampe) an, um uns unabhängig von der Ausleuchtungssituation des Spielfelds zu machen. Die Lampe wird nur bei Bedarf vor Start des Rennens eingeschaltet. Dieser Schalter muss zu Beginn des Rennens nicht bedient werden und ist daher regelgerecht. Der Scheinwerfer wird über einen separaten Spannungswandler mit einer Betriebsspannung von 4V versorgt und zieht 1 A Strom. (Dies stellt eine große Belastung für den klein dimensionierten Fahrakku dar.)
- Gyroskop (Komponente des Raspberry PI Sense Hat) zur Erkennung der Ausrichtung des Fahrzeugs und Ermittlung des Endes nach drei Runden: Bei jeder Runde dreht sich das Fahrzeug einmal komplett (360 Grad) um die Vertikalachse/Z-Achse. Leider haben sich die Messwerte des Gyroskops in der Praxis als sehr ungenau erwiesen, insbesondere bei schnellen Richtungswechseln. Dadurch kommt es vor, dass beim Ende des Rennens das Fahrzeug nicht korrekt im Start-/Zielabschnitt anhält. In der nächsten Fahrzeugversion muss ein genaueres Gyroskop verbaut werden
- Geschwindigkeits- und Streckenmessung mit Lichtschranke / Schlitzscheibe am rechten Hinterrad zur PID-basierten Geschwindigkeitssteuerung auf Basis von Geschwindigkeitsmesssensor „LM393 Geschwindigkeitsmessmodul Tacho Sensor Slot Typ IR Optokoppler mit Encoder“.
- Touch-Button für Start und Stop des Rennens: Kapazitiver Touch Sensor Taster TTP223b. (Abb. 9).

Abbildung 3: LIDAR-Scanner, Open MV H7+ Kamera, Sense Hat, Geschwindigkeitsmesser, Touch Button

Einzelheiten zur Verarbeitung der LIDAR-Daten

In ROS2 ist ein Hardware-Treiber für den verwendeten LIDAR-Scanner verfügbar. Dieser fungiert als Publisher und versendet die mit 10 Hz gescannten 360 Grad-Umgebungsdaten als Liste von jeweils 3240 Entfernungsdaten (Vollkreis). Genutzt werden im Rennmodus die vorderen 180 Grad des Sichtfelds. Der zentrale Knoten zur Datenerfassung (s2e_lidar_reader_node) bzw. Renndurchführung (full_drive_node) empfängt die Nachrichten mit den Entfernungsmessungen und verarbeitet diese weiter:

Quellcode-Ausschnitt Subscription:

```
self.subscription_lidar = self.create_subscription(
  LaserScan,
  '/scan',
  self.lidar_callback,
  qos_profile
)
```

Die vom LIDAR-Scanner empfangenen Rohdaten werden vorverarbeitet:

1. Fehlende Messwerte und Ausreißer (Entfernungswerte > 2.8 m) werden ersetzt und interpoliert.
2. Die gemessenen Entfernungsdaten werden invertiert, um naheliegenden Objekten höheres Gewicht zu geben.
3. Zur Verbesserung der Robustheit im Hinblick auf Messfehler und Abweichungen des Spielfelds wird von den Rohdaten eine Kopie erzeugt, verrauscht und wieder hinzugefügt. Dadurch steht zudem die doppelte Datenmenge fürs Training der KI zur Verfügung.

Quellcode-Ausschnitt Weiterverarbeitung Scan-Daten:

```
scan = np.array(msg.ranges[self.num_scan+self.num_scan2:]+msg.ranges[:self.num_scan2])
scan[scan == np.inf] = np.nan
scan[scan > self.scan_max_dist] = np.nan
x = np.arange(len(scan))
finite_vals = np.isfinite(scan)
scan_interpolated = np.interp(x, x[finite_vals], scan[finite_vals])
scan_interpolated = [1/value if value != 0 else 0 for value in scan_interpolated]
scan_interpolated = list(scan_interpolated)
```

Einzelheiten zur Bildverarbeitung

Auf den Kameras wird lokal Python Code ausgeführt. Dieser erkennt Hindernisse auf der Fahrbahn in Form von Blobs (farbige Blöcke) und überträgt deren Koordinaten mittels serieller Kommunikation via USB an die zentrale Steuerung. Dort werden die Daten von einem separaten ROS2-Knoten asynchron empfangen und weiterverarbeitet. Der Quellcode zur Bildbearbeitung auf den Open MV H7+-Kameras und den zugehörigen ROS2-Knoten auf dem Raspberry PI findet sich im Anhang.

Trainingsdaten und Training der KI

Bei Trainingsfahrten werden Daten zur Fahrbahn und potenzieller Platzierung von Hindernissen gesammelt: Zehnmal pro Sekunde wird ein Datensatz der Steuerungs- und Umgebungsparameter erzeugt (ergibt sich aus LIDAR-Scan-Frequenz). Der Datensatz enthält folgende Werte:

Feld #	**Feldinhalt**
0	Lenkeinschlag -1 … 1 – 0 ist Neutralstellung (geradeaus)
1	Gas -1 …1 (rückwärts … vorwärts)
2..1621	LIDAR-Daten – Hindernisentfernung von 0..2.8m/Inf (unendlich)
1622..1941	Kamera 1 (links) GRÜN – Pixelwert 1 an allen Positionen, an denen Hindernis gesehen wird
1942..2261	Kamera 1 (links) ROT – dito
2262..2581	Kamera 2 (rechts) GRÜN – dito
2582..2901	Kamera 2 (rechts) ROT – dito

Mit den vorverarbeiteten Daten wird das neuronale Netzwerk trainiert. Dazu wird das Programmpaket Tensorflow verwendet. Zur Beschleunigung der Berechnungen wird die CUDA-Bibliothek in Verbindung mit einer Nvidia RTX 4070 TI-Grafikkarte auf einem PC eingesetzt.

Hindernisse

Als Besonderheit erfolgt die Steuerung des Fahrzeugs mit künstlicher Intelligenz. Das komplette Programmpaket ist in Github abgelegt unter dem Account **rrrschuetz** und dem Paketnamen **s2e_lidar_reader**. Damit sind verteilte Entwicklung und Konfigurationsmanagement möglich. Auch die Entwicklungshistorie lässt sich lückenlos nachvollziehen.

Programmierung

Die komplette Software wurde in Python erstellt und besteht aus den folgenden Elementen:

Aufgabe	Dateinamen	Erläuterung
Sammeln von Trainingsdaten	s2e_lidar_reader_node.py	Datensammlung Rennen
Training der KI	calc_model_new.py	Training KI Rennen
Steuerung Rennen	full_drive_node.py	Zentraler ROS2-Knoten (Launch)
ROS2-Knoten zur Steuerung von Sensoren und Aktoren	display_node.py, openmv_h7_node1.py, openmv_h7_node2.py, speed_control_node.py, xbox_reader_node.py, touch_button_node.py	Statusanzeige Kamera links Kamera rechts Motorsteuerung Fernsteuerung für Datensammlung Startknopf für das Rennen
Kamera-Steuerung	main.py h7_cam_exec.py	Start Lokale Verarbeitung der Hinderniserkennung auf Kamerachip
Dateninspektion	cam2_viewer.py, data_viewer.py	Visualisierung der Dateien mit Trainingsdaten

Rennen mit KI-Steuerung

Beim Rennen werden dieselben Daten erfasst wie beim Training. In Echtzeit werden mittels Interferencing mit dem vortrainierten neuronalen Netzwerk Steuerungsdaten (Lenkung) auf dem Raspberry PI 4B-Bordrechner ermittelt. Hierdurch entstehen Fahrkurven um die Hindernisse, wie sie ein menschlicher, vorausschauender Fahrer wählen würde.

Beim Start des Rennens muss ermittelt werden, ob das Fahrzeug für ein Rennen im Uhrzeigersinn oder im Gegenuhrzeigersinn aufgestellt ist. Dies entscheiden wir durch Messungen mit dem LIDAR-Sensor: Wenn in der Startaufstellung bei Sicht nach vorn rechts mehr freie Fläche gemessen wird, handelt es sich um ein Rennen im Uhrzeigersinn, bei mehr freier Sicht nach links ist es ein Rennen im Gegenuhrzeigersinn.

Befinden sich im Start- und Zielbereich Hindernisse im Sichtfeld vor dem Roboter, fahren wir bis an die rückwärtige Spielfeldwand zurück, um von dort aus die Messung zur Bestimmung der Umlaufrichtung vorzunehmen. Hierdurch können Fehlentscheidungen vermieden werden.

Datenverarbeitung / Modellarchitektur des neuronalen Netzes

Die Architektur ist speziell darauf ausgelegt, die Daten von LIDAR- und Farbsensoren zu verarbeiten, um Entscheidungen zur Fahrzeugnavigation im Hindernisparcours in Echtzeit zu treffen: Während die LIDAR-Daten präzise Konturen der Fahrbahnbegrenzung und der Hindernisse liefern, wird die Information der Kameras benötigt, um den Hindernissen die korrekte Farbe (rot, grün, magenta) zuzuordnen. Nur so kann die KI entscheiden, ob ein Hindernis links oder rechts passiert werden muss oder ob es sich um ein Parkplatzhindernis handelt.

Eingabeschichten

LIDAR Input: Nimmt die LIDAR-Daten auf, welche die Distanzen zu nahegelegenen Objekten und Hindernissen messen. Diese Daten werden durch mehrere Conv1D Schichten verarbeitet. Diese Schichten helfen, die räumlichen Merkmale aus den LIDAR-Daten zu extrahieren, die für die Erkennung und Bewertung von Hindernissen wesentlich sind.

Color Input: Verarbeitet die Kameradaten. Diese Daten durchlaufen ebenfalls Conv1D Schichten. Die Verarbeitung der Farbdaten hilft dem Modell, visuelle Merkmale wie die Farbe und Textur der Hindernisse zu erkennen

Merkmalsextraktion und Datenfusion

Nach den initialen Conv1D und MaxPooling1D Schritten werden die Features beider Sensortypen normalisiert durch Batch Normalization, um das Lernen zu beschleunigen und zu stabilisieren.

Die Merkmale aus beiden Eingabetypen werden mittels einer Concatenate Schicht zusammengeführt. Diese Fusion erlaubt dem Modell, eine umfassendere Sicht der unmittelbaren Umgebung des Fahrzeugs zu entwickeln, indem es sowohl räumliche als auch visuelle Informationen kombiniert.

Dichte Schichten und Entscheidungsfindung

Mehrere Dense-Schichten folgen der Fusion, um die kombinierten Daten weiter zu verarbeiten. Diese Schichten dienen dazu, komplexe Muster und Beziehungen zwischen den extrahierten Features zu erkennen und zu lernen, die für die Steuerungsentscheidungen des Fahrzeugs kritisch sind.

Die Ausgabeschicht des Modells besteht aus Dense Neuronen, die direkt die Steuerungsbefehle für das Fahrzeug (wie Lenkwinkel und Geschwindigkeit) generieren, basierend auf der aktuellen Wahrnehmung der Umgebung.

Training, Optimierung, Konversion

Das Modell wird mit einem Adam-Optimierer trainiert. Der Verlust wird als Mean Squared Error berechnet. Damit wird die Genauigkeit der Steuerungsbefehle bei der Trajektorienberechnung maximiert. EarlyStopping wird verwendet, um Overfitting zu verhindern und das Training zu überwachen.

Das Modell wird als TensorFlow Lite Modell gespeichert, um die ressourcenschonende Verwendung auf dem Raspberry Pi 4B zu gestatten.

Quellcode zur Definition des neuronalen Netzes

```
def create_cnn_model(lidar_input_shape, color_input_shape):
    # LIDAR data path
    lidar_input = Input(shape=lidar_input_shape)
    lidar_path = Conv1D(64, kernel_size=5, activation='relu')(lidar_input)
    lidar_path = MaxPooling1D(pool_size=2)(lidar_path)
    lidar_path = Conv1D(128, kernel_size=5, activation='relu', kernel_regularizer=l2(0.01))(lidar_path)
    lidar_path = MaxPooling1D(pool_size=2)(lidar_path)
    lidar_path = Flatten()(lidar_path)

    color_input = Input(shape=color_input_shape)
    color_path = Dense(64, activation='relu')(color_input)
    color_path = Dropout(0.3)(color_path)  # Use dropout
```

```
color_path = Dense(128, activation='relu', kernel_regularizer=l2(0.01))(color_path)  # Regularization
color_path = Flatten()(color_path)

# Concatenation
#concatenated = Concatenate()([lidar_path, color_path])
concatenated = WeightedConcatenate(weight_lidar=0.2, weight_color=0.8)([lidar_path, color_path])

# Further processing
combined = Dense(64, activation='relu')(concatenated)
combined = Dense(64, activation='relu')(combined)
combined = Dense(64, activation='relu')(combined)
combined = Dense(64, activation='relu')(combined)
combined = Dense(32, activation='relu')(combined)
output = Dense(2)(combined)

model = Model(inputs=[lidar_input, color_input], outputs=output)
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['accuracy'])
return model
```

Besonderheiten Eröffnungsrennen

Beim Eröffnungsrennen kann eine höhere Fahrgeschwindigkeit gewählt werden, weil keine engen Kurven um die roten und grünen Hindernisse gefahren werden müssen und die Informationen der Kameras bei der Ermittlung der Steuerungsimpulse keine Rolle spielen. Bei der Wahl der Geschwindigkeitsvorgabe muss beachtet werden, dass die Geschwindigkeit nur am rechten Hinterrad gemessen wird. Daher ist der Sollwert für Fahrten im Uhrzeigersinn etwas niedriger eingestellt.

Besonderheiten Parkmanöver

Nach Beenden der drei Umläufe des Hindernisrennens sucht der Roboter mit der auf die Außenseite des Spielfelds ausgerichteten Kamera nach dem magentafarbenen Parkhindernis und hält in der Mitte des jeweiligen Abschnitts an.

Während des Rennens zählen wir die Sichtung magentafarbener Hindernisse, um zwischen einer Fahrt mit oder ohne Parkmanöver entscheiden zu können: Am Ende des Hindernisrennens mit Parkmanöver steht dieser Zähler je nach Lichtverhältnissen ungefähr auf dem Wert 200, ansonsten auf < 10. Ab 50 gehen wir von einem Hindernisrennen aus. Damit senken wir das Risiko von Fehlentscheidungen bei Sichtung magentafarbener Objekte am Spielfeldrand.

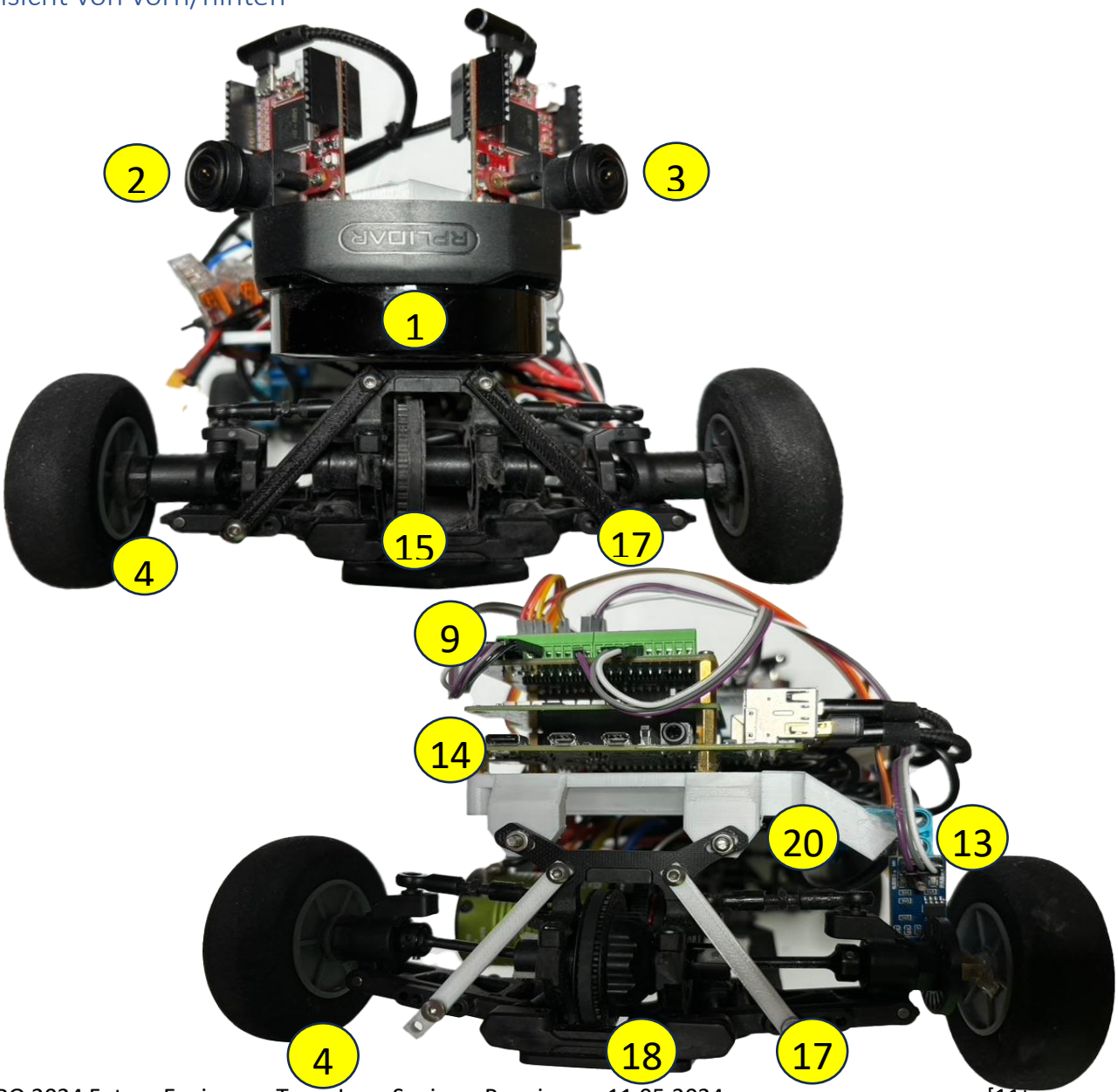
Aufgrund des Fehlens rückwärtiger Sensoren und der niedrigen Präzision unseres Antriebs bei sehr kurzen Fahrstrecken parken wir nur vorwärts und teilweise ein (Regel 7.13). Durch kontinuierliche Abstandsmessungen vermeiden wir eine Berührung des Parkhindernisses.

Fotos

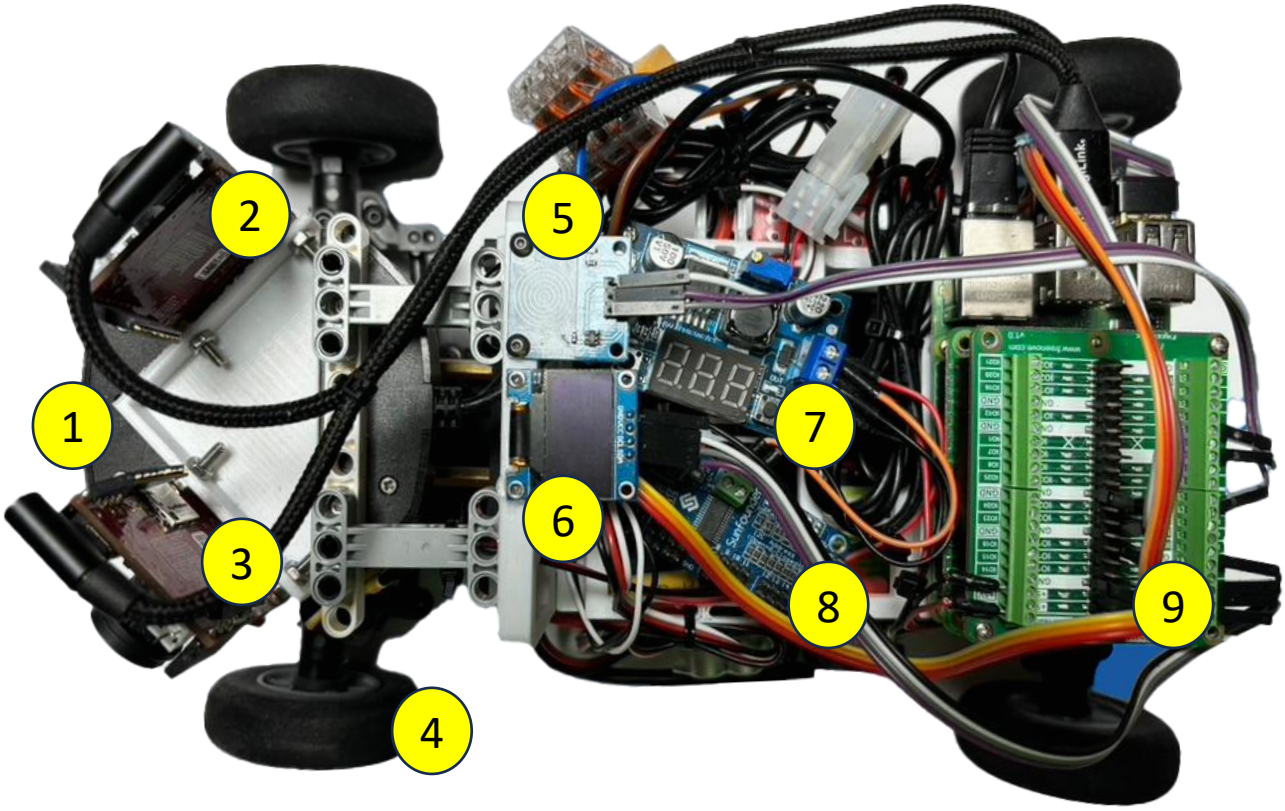
Auf den nachfolgenden Abbildungen sind die wesentlichen Komponenten des Fahrzeugs ersichtlich. Hier die Beschreibungen, die sich anhand der Nummern in den gelben Markierungen zuordnen lassen.

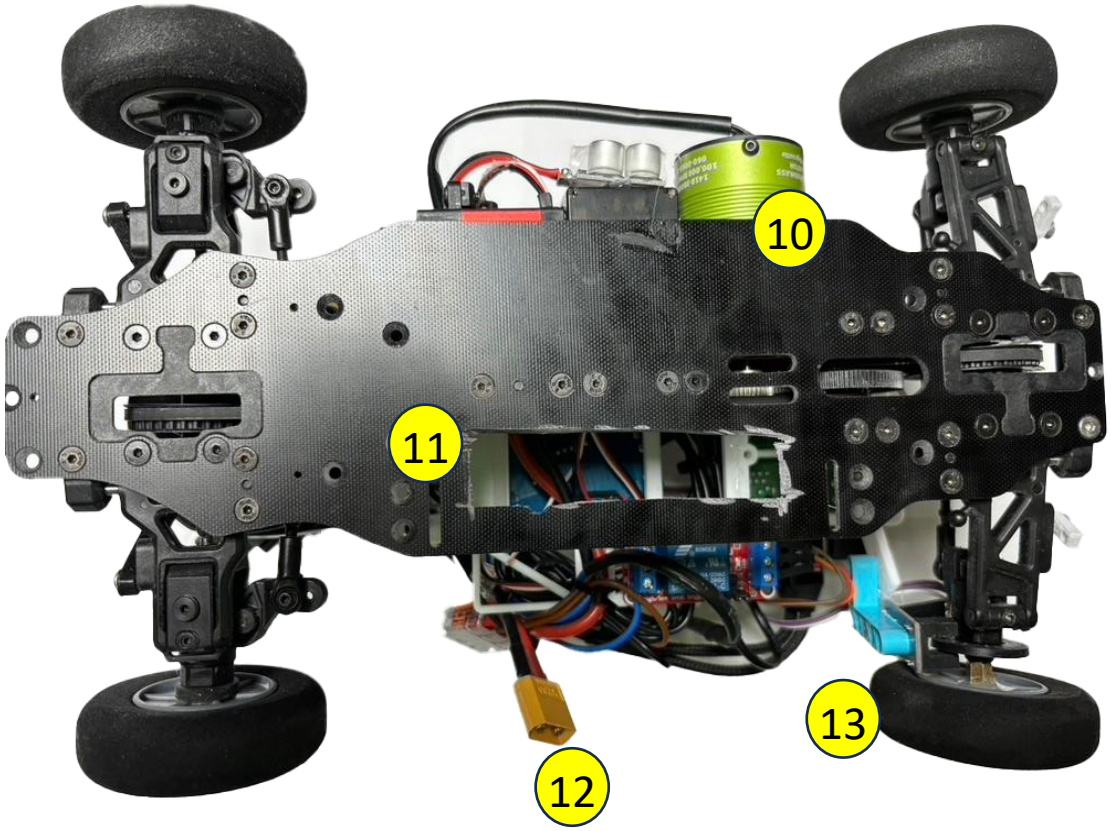
#	Bezeichnung	#	Bezeichnung
1	LIDAR-Scanner	11	Aussparung zur Gewichtsminderung
2	Kamera rechts	12	Akku-Anschluß
3	Kamera links	13	Geschwindigkeitsmesser (Lichtschranke mit Schlitzscheibe)
4	Rad mit Moosgummi-Bereifung	14	Bordrechner Raspberry PI 4B mit Sense Hat
5	Touch Sensor (Startknopf)	15	Vorderradantrieb (Riemen)
6	OLED-Display zur Zustandsanzeige	16	Lenkservo
7	Spannungswandler	17	Leichtgewicht-Ersatz für Dämpferbein
8	I2C-Bus-Interface zur Ansteuerung von ESC, Lenkservo	18	Hinterachsdifferential und -antrieb
9	Raspberry Pi Breakout Board	19	Antriebsritzel
10	Fahrmotor mit Hallsensor-Anschluß	20	Tragkonstruktion für Elektronik

Ansicht von vorn/hinten

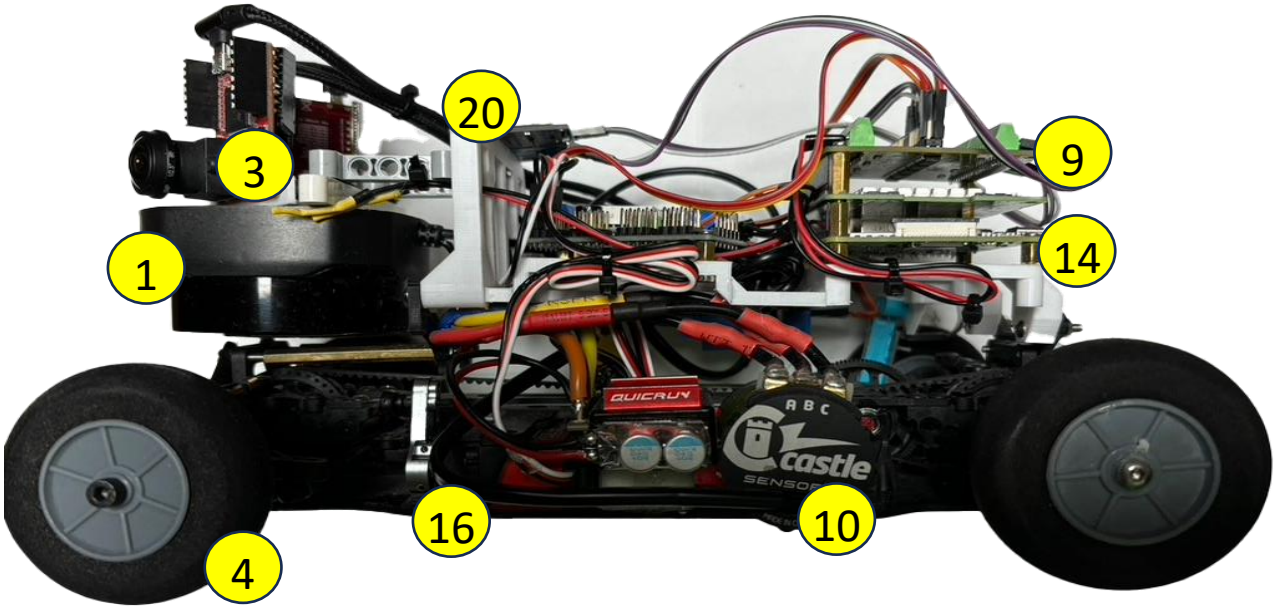


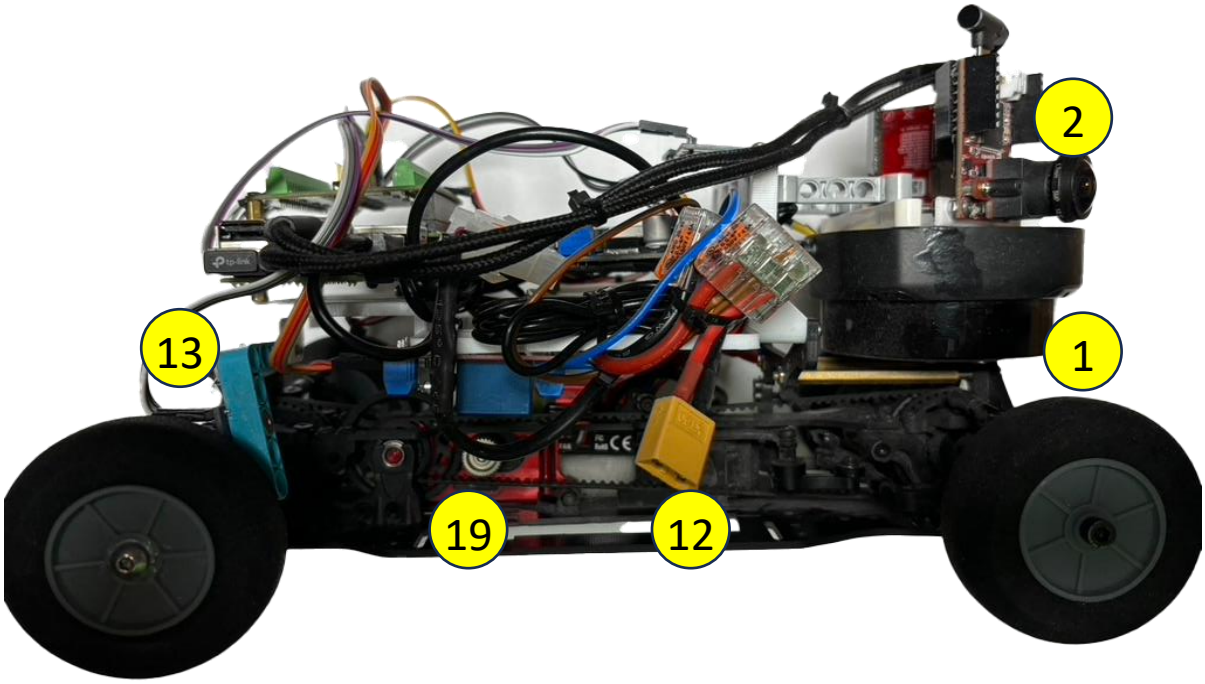
Ansicht von oben/unten





Ansicht von links/rechts





Engineering / Design

Entwicklungshistorie

Es wurde keines der empfohlenen Standard-Chassis eingesetzt, um das Konzept der Steuerung mit LIDAR und KI umsetzen zu können. Das Fahrzeug ist eine Eigenentwicklung unter Verwendung von Standardbauteilen. Insgesamt wurden drei Evolutionsstufen des Fahrzeugs mit unterschiedlichen Chassis-Komponenten durchlaufen:

1. **Erster Prototyp** zum Machbarkeitsnachweis der KI-Steuerung auf Basis: Freenove 4WD Smart Car Kit for Raspberry Pi 4 B. Getestet wurde ausschließlich die Erfassung der Umgebung mit dem LIDAR-Sensor und das Training einer ersten Fassung der KI. Die Kameras und die Verarbeitung der für das Hindernisrennen wichtigen Farbinformationen fehlten noch.
2. **Zweiter Prototyp** mit regelkonformer Fahrzeugsteuerung aber Übermaßen und Übergewicht auf Basis Reely TC-04 Onroad-Chassis 1:10 RC (4WD) ARR (überwiegend aus Metallteilen). Diese Fahrzeugvariante diente zum Testen sämtlicher Varianten von Sensoren. Neben den Kameras wurden auch IR- und Ultraschallsensoren zur rückwärtigen Entfernungsmessung getestet. Auch die Erkennung der farbigen Linien auf dem Spielfeld wurde getestet.
3. **Rennfahrzeug in Leichtbauweise** mit Einhaltung aller Regeln auf Basis eines abgespeckten Bausatzes XPRESS 90005 - EXECUTE XM1S - 1:10 4WD M-Chassis (überwiegend aus Kunststoffteilen). Neben den gewichtsreduzierenden Massnahme wurde eine besonders niedrige Anordnung des LIDAR-Sensors verwirklicht, so dass die Fahrbahnbegrenzungen und Hindernisse auf einer Höhe von deutlich unter 10 cm erfasst werden.

Abbildung 4: Fahrzeuggenerationen: Freenove 4WD, Reely TC-04 und EXECUTE XM1S sowie Elektronik-Träger (3D-Druck, Eigenentwicklung)

Maßnahmen zur Gewichtsreduktion

- Weglassen nicht unbedingt notwendiger Teile: Karosserieträger, Stoßdämpfer, sämtliche nicht für den Antrieb erforderlichen Teile.
- Einfügen von Aussparungen im Chassis (mit Dremel): Chassis ist im Rennbetrieb aufgrund des geringen Tempos und der ebenen Fahrbahn keinen starken Kräften ausgesetzt.
- Um Elektronik und Sensoren auf dem Chassis montieren zu können, wurde eine Tragkonstruktion mittels 3d-Druck mit minimalem Infill erstellt. Das Design ist aus Abbildung 4 ersichtlich.
- Es wurden schmale Räder aus dem Modellflugzeugbau in Leichtbauweise mit Moosgummi-Bereifung ausgewählt. Um den notwendigen Grip und Lenkpräzision zu erreichen, wird ein Chassis mit Allrad-Steuerung und Riemenantrieb verwendet. Nur an der Hinterachse ist ein Differential verbaut.
- Verkürzung der schweren Kupferkabel zwischen Akku, ESC und Motor.
- Kürzestmögliche Verbindungskabel zwischen den Komponenten (z.B. USB-Kabel für die Kameras)
- Leichter LiPo-Akku mit gerade so viel Kapazität wie für die Stromversorgung während der Rennen erforderlich.

Bordrechner

Die Steuerung erfolgt via Raspberry PI 4B unter Ubuntu Linux 22.03 und ROS2 (Humble). Bei ROS2 handelt es sich um eine modulare Open Source Robotik-Middleware zur asynchronen Integration von Aktoren und Sensoren und dynamischer Steuerung. Die einzelnen Komponenten werden in Knoten abgebildet, die mittels Nachrichtenversand in Echtzeit miteinander kommunizieren.

Zur Echtzeitfähigkeit der Ermittlung der Steuerungsinformationen wurden folgende Maßnahmen ergriffen:

- Die Bildauswertung erfolgt zur Senkung des Rechenaufwands auf dem Raspberry Pi direkt auf den Open MV H7+-Kameras, s.u.
- Während des Rennens werden nur die Daten der kurveninneren Kamera ausgewertet. Es wird eine horizontale Bildauflösung der Kameras von nur 320 Pixeln verwendet.
- Das KI-Interferencing erfolgt mit Tensorflow Lite. Gegenüber Tensorflow kann der Rechenaufwand des Interferencing deutlich gesenkt werden.
- Es wird nur der vorwärtsgerichtete Erfassungsbereich des Scanners ausgewertet. Dadurch kann die LIDAR-Datenmenge halbiert werden (1620 statt 3240 Datenpunkte).

Die von der KI errechnete Fahrmotor-Einstellung wird bei den Rennfahrten nicht genutzt: Die hohe Leistung des Motors führte sonst zu einer zu starken Beschleunigung und Kontrollverlust. Stattdessen wird die Geschwindigkeit mit einem PID-Controller im ROS2-Motorsteuerungsknoten konstant gehalten. Die Verwendung des PID-Controllers führt bei seiner Einregulierung beim Start zu einer Anfahrverzögerung von einigen Sekunden. Dies wird in Kauf genommen: Ein ruckartiger Start führte zu Oszillationen der Geschwindigkeit mit Auswirkungen auf die Orientierungsfähigkeit.

Die Maximalgeschwindigkeit des Fahrzeugs ist im Wesentlichen durch die Abtastfrequenz des LIDAR-Scanners (10 Hz) bestimmt. Damit kann das Hindernisrennen nahezu fehlerfrei in ca. 90-100 Sekunden gefahren werden. Der Ortsabstand zwischen zwei Umgebungsscans liegt somit bei weniger als 5 cm. Dies ermöglicht eine sehr präzise Steuerung: Wird die Fahrgeschwindigkeit deutlich erhöht, steigt das Risiko für Kollisionen.

Vor dem Start des Rennens wird der vollgeladene Akku angeschlossen, das Fahrzeug bleibt aber stromlos. Die eigentliche Aktivierung erfolgt gem. Regelwerk mit einem einfachen Kippschalter. Nach dem Booten des Bordrechners wird per Autostart das ROS2-Netz mit den Steuerungsprogrammen gestartet und verharrt im Bereitschaftsmodus. Dies wird auf dem kleinen OLED-Display auf der Oberseite des Fahrzeugs angezeigt. Durch Berühren des daneben liegenden Touch Sensors wird der Rennmodus gestartet. Erst ab diesem Zeitpunkt werden Daten erfasst, Berechnungen gemacht und Steuerungskommandos ausgeführt.

Während der Test- und Trainingsfahrten erfolgt ein umfangreiches Logging, welches mit einem SSH-Login via WLAN an der Konsole zur Verlaufskontrolle und zum Debugging eingesehen werden kann.

Einschränkungen bzw. Verbesserungsmöglichkeiten

Separates Training je eines KI-Modells für Fahrt im Uhrzeiger- und Gegenuhrzeigersinn erforderlich: Es wird jeweils ein KI-Modell für die Fahrt im Uhrzeiger- und im Gegenuhrzeigersinn trainiert. Theoretisch wäre dies nicht notwendig, wenn man bei der Steuerung die Symmetrieeigenschaften der Rennstrecke ausnutzte. Dies haben wir getestet: Allerdings stellte sich heraus, dass aufgrund der leicht asymmetrischen Bauweise des Fahrzeugs - insbesondere wegen des ausgestellten rechten Hinterrads zur Unterbringung des Geschwindigkeitsmessers - die Fahreigenschaften bei Links- und Rechtskurven zu unterschiedlich sind.

Begrenzung der Fahrgeschwindigkeit durch die Abtastfrequenz des LIDAR-Sensors: Die Kosten schnellerer LIDAR-Sensoren mit einer Abtastfrequenz von deutlich mehr als 10 Hz sind unverhältnismäßig höher. Außerdem sind diese Sensoren zu schwer, um die Gewichtshöchstgrenze einhalten zu können.

Minimaler Kurvenradius: Aufgrund der Konstruktion des Chassis und dem Maximalausschlag des Lenkservos ergibt sich ein minimaler Kurvenradius, der die Bewältigung einer Hinderniskonstellation ohne Rangiermanöver unmöglich macht. Das Rangieren mit abwechselnden Vor- und Rückwärtsfahrabschnitten ist konzeptionell nicht vorgesehen. Auch der Einbau einer zusätzlichen Hinterachslenkung ist aus Gewichtsgründen nicht möglich. Treten Hindernis-Konstellation im Rennen auf, die solch enge Radien erforden, führt dies zu einem entsprechenden Fehler bzw. Punktabzug. Ein Beispiel ist in Abbildung 6 zu sehen.

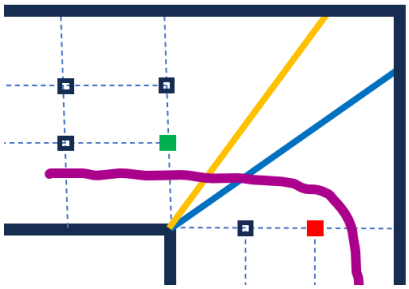


Abbildung 5: Nicht fahrbare Hinderniskonstellation

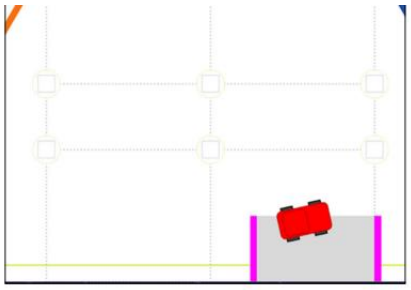


Abbildung 6: Fahrzeug teilweise eingeparkt gem. Regel 7.13

Langsamfahrten: Es wäre vorteilhaft, wenn die Getriebeübersetzung weiter erhöht werden könnte. Damit könnte das Fahrzeug langsamer fahren und präziser sehr kurze Fahrstrecken zurücklegen, wie dies beim Parken erforderlich ist. Leider war für das verwendete Chassis keine Zahnradkombination zu finden, die dies erlaubt hätte. Auf einen Austausch des kompletten Antriebsstrangs zu diesem Zweck wurde aus Platz- und Gewichtsgründen verzichtet.

Einschränkung beim Parkmanöver durch Gewichtslimit (1500g): Wir mussten zur Einhaltung des Maximalgewichts auf den Einbau weiterer Sensoren und Features zur Sicht nach hinten verzichten. Damit lässt sich beim Parken nicht die notwendige Präzision erreichen, wie sie die geringen Platzverhältnisse in der Parklücke erforderten. Verschärft wird dieses Problem durch das ruckartige Anfahrverhalten. Damit ist das Risiko einer Berührung der magentafarbenen Parkplatzhindernisse unattraktiv hoch. Wir beschränkten uns folglich auf eine Teilrealisierung gem. Regel 7.13 (Abbildung 7).

Keine Erkennung der Farben der Hindernisse bei ungünstigen Lichtverhältnissen: Bei schlechter Ausleuchtung des Spielfelds erkennen die Kameras die Farben der Hindernisse nicht. Die KI wählt dann den kürzesten Weg und fährt u.U. auf der falschen Seite eines Hindernisses vorbei. Durch einen Frontscheinwerfer werden deshalb definierte Lichtverhältnisse hergestellt werden (Nachteile: Hitze, Energieverbrauch).

Kein Fahrtrichtungswechsel mit KI: Aktuell kann mit der KI nur vorwärtsgefahren werden: Kreuzten sich beim Training Trajektorien, auf denen sowohl vorwärts als auch rückwärts gefahren wird, kann das Netzwerk keine eindeutige Steuerungsantwort auf die eingegebenen Umgebungsparameter ermitteln. Das Ergebnis wären zufällig aufeinanderfolgende Kommandos für Vor- und Rückwärtsfahrt. Dies hätte ein vollautomatisches Parkmanöver mit KI erschwert. Um der KI die Unterscheidung der Fahrtrichtung zu erlauben, hätte ein aufwändigeres LSTM-Modell verwendet werden müssen, mit dem auch Zeitreihen trainiert werden können. Dies hätte den Trainingsaufwand wesentlich erhöht.

Anhang: Quellcode zur Bildverarbeitung

Quellcode auf Open MV H7+:

```
import sensor, image, time, math, pyb, os
import machine

# Get the unique machine ID
unique_id = machine.unique_id()
unique_id_hex = ''.join(['{:02x}'.format(byte) for byte in
unique_id])

usb = pyb.USB_VCP()
red_led = pyb.LED(1)
green_led = pyb.LED(2)
blue_led = pyb.LED(3)

red_led.on()
green_led.off()
blue_led.off()

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.set_auto_gain(True)
sensor.set_auto_whitebal(True)
sensor.skip_frames(time = 2000)

green =  (30, 100, -64, -8, -32, 32)  # generic green
red =  (30, 100, 15, 127, 15, 127)   # generic red
magenta = (0, 100, 32, 127, 127, -94)
thresholds=[green, red, magenta]
roi = [0,0,320,140]

while True:
  while usb.any():
    data = usb.recv(4096)  # Receive 64 bytes at a time

  try:
    img = sensor.snapshot()
    img.lens_corr(strength=2.6, zoom=1.0)
    img.gamma_corr(gamma=1.0,contrast = 1.0,brightnes =0.2)

    blob_entries = []
    blobs = img.find_blobs(thresholds,0,roi,pixels_threshold=160,
merge=False)
    for blob in blobs:
      (b_x,b_y,b_w,b_h) = blob.rect()
      b_c = blob.code()
      if (b_c == 4 and b_h/b_w < 1) or (b_c in [1,2] and b_h/b_w >
1):
        blob_entries.append("{},{},{}".format(b_c, b_x, b_x+b_w))

    bloblist = ','.join(blob_entries)
    if bloblist:
      usb.write("{},".format(unique_id_hex))
      usb.write(bloblist)
      usb.write("\n")
      usb.flush()

  except Exception as e:
    pass
```

Quellcode des ROS2-Knotens zur Weiterverarbeitung der Kamerainformationen:

```
def openmv_h7_callback(self, msg):
    try:
      data = msg.data.split(',')

      if data[0] == '240024001951333039373338': cam =
      elif data[0] == '2d0024001951333039373338': cam =
      else: return

      if cam == 1:
        self._color1_g = np.zeros(self.HPIX, dtype=int)
        self._color1_r = np.zeros(self.HPIX, dtype=int)
        self._color1_m = np.zeros(self.HPIX, dtype=int)
      elif cam == 2:
        self._color2_g = np.zeros(self.HPIX, dtype=int)
        self._color2_r = np.zeros(self.HPIX, dtype=int)
        self._color2_m = np.zeros(self.HPIX, dtype=int)

      blobs = ((data[i],data[i+1],data[i+2]) for i in range (1,len(data),3))
      for blob in blobs:
        color, x1, x2 = blob
        color = int(color)
        x1 = int(x1)
        x2 = int(x2)
        if color == 1:
            if cam == 1 and not self._clockwise:
                self._color1_g[x1:x2] = self.WEIGHT
            if cam == 2 and self._clockwise:
                self._color2_g[x1:x2] = self.WEIGHT
        if color == 2:
            if cam == 1 and not self._clockwise:
                self._color1_r[x1:x2] = self.WEIGHT
            if cam == 2 and self._clockwise:
                self._color2_r[x1:x2] = self.WEIGHT
        if color == 4:
            if cam == 1: self._color1_m[x1:x2] = self.WEIGHT
            if cam == 2: self._color2_m[x1:x2] = self.WEIGHT
            self._parking_lot += 1

    except:
      self.get_logger().error('Faulty cam msg received: "%s"' % msg)
```

Hinweis: Zur korrekten Zuordnung der Kameras zur jeweiligen Blickrichtung nach links und rechts werden die Hardware-Adressen abgeglichen.

Anhang: Informationen zu den KI-Trainingsläufen und Tests

Bei den Trainingsläufen ist es wichtig, alle relevanten Konstellationen von Hindernissen zu trainieren. Zur Reduktion der Anzahl der Szenarien haben wir von den Symmetrieeigenschaften des Spielfelds Gebrauch gemacht.

Das Training für das Eröffnungs- und Hindernisrennen führten wir mit getrennten Datensätzen durch. Dies führte in unseren Tests zu wesentlich präziseren Ergebnissen.

Für die Planung der Trainingsläufe haben wir alle denkbaren Hinderniskonstellationen aufgezeichnet und im Training dann sowohl im Uhrzeigersinn als auch im Gegenuhrzeigersinn abgefahren.

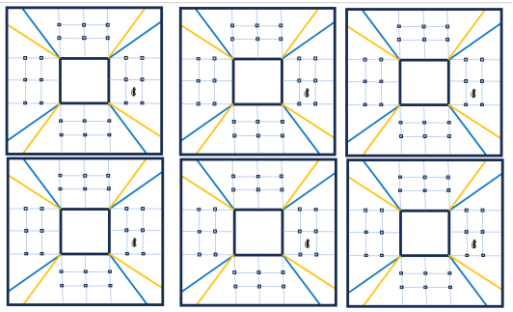


Abbildung 7: Leere Seite aus dem Dokument zur Trainingsplanung

Für jede Umlaufrichtung waren ca. eine Stunde Fahrten zur Aufnahme der Trainingsdaten erforderlich. Aufgrund der Aufzeichnungsfrequenz von 10 Hz sammelten wir so insgesamt etwa 100.000 Datensätze in lesbaren .txt-Dateien. Diese ließen sich sehr flexibel kopieren, zerschneiden und kombinieren: Die Datensammlung haben wir in vielen einzelnen Trainingsläufen vorgenommen und die Daten erst für das KI-Training in einer Datei zusammengefasst.

Das Trainieren der KI-Modelle benötigt aufgrund der eingesetzten Hardware (High End-Grafikkarte mit CUDA-Bibliothek) nur wenige Minuten. Ohne die Grafikkarte hätte die Berechnung über eine Woche benötigt. Dies hätte den Test- und Entwicklungsprozess unmöglich gemacht.

Die Vorhersagegenauigkeit unserer KI-Modelle für das jeweils nächste Fahrmanöver liegt bei mehr als 80%. Dies ist mehr als ausreichend: Haupteinfluss auf die Präzision der Steuerung hat ja die Fahrgeschwindigkeit bei konstanter LIDAR-Scanfrequenz (10Hz). Eine durch die KI verursachte Ungenauigkeit kann somit bereits nach 0.1 s beim nächsten Steuerkommando kompensiert werden.

Die Daten der fertig trainierten vier KI-Modelle (je eines für das Eröffnungs- und Hindernisrennen, jeweils im Uhrzeigersinn und im Gegenuhrzeigersinn) befinden sich auf der MicroSD-Karte des Bordrechners. Sie werden beim Start des Steuerprogramms in den Speicher geladen. Je nach Fahrsituation wird zwischen ihnen umgeschaltet.

Die für die Planung der Trainingsläufe verwendete Unterlage haben wir auch zur Testfalldokumentation verwendet: Wir haben systematisch alle Konstellationen von Hindernissen ausprobiert. Bei den Testfahrten identifizierte Fehler wie z.B. die Berührung von Hindernissen konnten wir durch Nachtrainieren mit zusätzlichen Daten für die spezielle Fehlersituation ausmerzen. Um dabei nicht die Übersicht zu verlieren, haben wir zusätzlich die Fehler mittels Videoaufnahmen mit dem Handy dokumentiert.

Glossar und Literaturverzeichnis

1D CNN (eindimensionales Convolutional Neural Network, im Programm auch **Conv1D** genannt): Neuronales Netzwerk zur Verarbeitung sequentieller, eindimensionaler Daten

Adam-Optimierer: Algorithmus zur Optimierung von neuronalen Netzwerken, der auf stochastischem Gradientenabstieg basiert und adaptive Schätzungen von niedrigeren Ordnungen verwendet.

Antriebsritzel: Kleines Zahnrad auf der Antriebsachse des Elektromotors. Mit der Größe des Ritzels lässt sich das Übersetzungsverhältnis anpassen.

Dense Layer: Ein vollständig verbundener Layer in einem neuronalen Netzwerk, in dem alle Eingaben mit allen Ausgaben verbunden sind. Die Anzahl der Dense Layer entscheidet über die Gedächtnisleistung des neuronalen Netzwerks.

CUDA: Eine von Nvidia entwickelte Programmierschnittstelle, die es ermöglicht, allgemeine Berechnungen auf Grafikprozessoren (GPUs) durchzuführen. Durch CUDA und eine geeignete Grafikkarte (hier Nvidia RTX 4070 TI) lassen sich die Berechnungen mit Tensorflow drastisch beschleunigen.

ESC (Electronic Speed Controller): Steuermodul für Geschwindigkeit und Richtung von Elektromotoren, häufig in Drohnen und RC-Fahrzeugen verwendet.

GPIO (General Purpose Input/Output): Pins an Mikrocontrollern und Computern, die zur Interaktion mit anderen elektronischen Komponenten verwendet werden. Über diese Schnittstelle werden die Sensoren und die Steuerkomponenten an den Raspberry PI angebunden.

Gyroskop: Sensor auf dem Raspberry Pi Sense Hat zur Messung der Orientierung und Winkelgeschwindigkeit

Hall-Sensor: Umdrehungszähler am Elektromotor basierend auf kleinen Magneten, deren elektrische Felder beim Rotieren elektrische Signale erzeugen. Der Hall-Sensor wird benötigt, um beim Langsamfahren das Cogging (Stottern) des Motors mit dem ESC zu unterdrücken.

I2C-Bus: Kommunikationsbus des Raspberry Pi zur Anbindung von externen Sensoren oder Aktoren

Interferencing: Prozess des Anwendens eines trainierten KI-Modells auf neue Daten zur Vorhersage oder Analyse, hier zur Ermittliung der Lenkbefehle für den Steuerservo.

KI-Modell (Künstliche Intelligenz-Modell): Ein computergestütztes Modell, das darauf trainiert ist, menschenähnliche Aufgaben durchzuführen oder Entscheidungen zu treffen.

LIDAR (Light Detection and Ranging): Eine Technik zur Entfernungsmessung, die Licht in Form eines gepulsten Lasers verwendet. Beim verwendete 1D-LIDAR rotiert der Laser mit einer Frequenz von 10 Hz und einem Puls von 32 kHz. Damit werden für einen Vollkreis (360 Grad) 3240 Entfernungsdatenpunkte ermittelt und via Ethernet übertragen.

LSTM (Long Short-Term Memory): Typ von rekurrenten neuronalen Netzwerken, die speziell dafür entwickelt wurden, Langzeitabhängigkeiten zu lernen

OPEN MV H7: Kleines, leistungsstarkes Computer-Vision-Modul, das für maschinelles Sehen und ML-Anwendungen optimiert ist. Die Kamera (mit Fisheye-Objektiv) ist direkt auf der Platine mit dem Prozessor-Chip aufgebracht, dadurch sehr kompakte, leichte Bauweise. USB-Schnittstelle verfügbar.

PID-Algorithmus – Regelkreis zur dynamischen Steuerung eines Motors. Schwankungen werden automatisch ausgeglichen, so dass eine gleichmässige Leistungsentfaltung erfolgt.

Raspberry PI 4B: Kleiner, preiswerter Computer, der häufig in Bildung, Hobbyelektronik und DIY-Projekten verwendet wird.

ROS2 (Robot Operating System 2): Open Source Set von Softwarebibliotheken und Tools, das hilft, Robotikanwendungen zu bauen. Unterstützt werden Programm-Module in den Programmiersprachen Python und C/C++.

Tensorflow: Open-Source-Softwarebibliothek für maschinelles Lernen, entwickelt von Google, um neuronale Netzwerke zu trainieren und einzusetzen.

(1) Mastering ROS for Robotics Programming - Third Edition: Best practices and troubleshooting solutions when working with ROS Taschenbuch – 15. Oktober 2021, Englische Ausgabe, Lentin Joseph (Autor), Jonathan Cacace (Autor)

(2) Learn Robotics Programming - Second Edition: Build and control AI-enabled autonomous robots using the Raspberry Pi and Python Taschenbuch – 12. Februar 2021, Englische Ausgabe Danny Staple (Autor)

(3) Algorithmen in Python: Das Buch zum Programmieren trainieren. 32 Klassiker der Informatik, von Damenproblem bis Neuronale Netze, broschiert – 28. Juni 2020 von David Kopec (Autor)

(4) Think Python: Systematisch programmieren lernen mit Python (Animals) Taschenbuch – 1. Juli 2021 von Allen B. Downey (Autor), Peter Klicman (Übersetzer)

(5) Let's code Python: Programmieren lernen mit Python ohne Vorkenntnisse. Ideal für Kinder und Jugendliche Taschenbuch – 28. Dezember 2018 von Hauke Fehr (Autor)

(6) Parallel and High Performance Programming with Python: Unlock parallel and concurrent programming in Python using multithreading, CUDA, Pytorch and Dask. (English Edition) Taschenbuch – 14. April 2023, Englische Ausgabe, Fabio Nelli (Autor)

(7) Hands–On Machine Learning with Scikit–Learn and TensorFlow Taschenbuch – 24. März 2017, Englische Ausgabe, Aurelien Geron (Autor)

(8) Hands-on Convolutional Neural Networks with Tensorflow Taschenbuch – 29. August 2018, Englische Ausgabe, Iffat Zafar (Autor), Giounona Tzanidou (Autor)

(9) Neural Networks and Deep Learning: A Textbook Gebundene Ausgabe – 13. September 2018, Englische Ausgabe, Charu C. Aggarwal (Autor)

(10) Statistik: Der Weg zur Datenanalyse (Springer-Lehrbuch) Taschenbuch – 15. September 2016, von Ludwig Fahrmeir (Autor), Christian Heumann Rita Künstler

(11) CUDA by Example: An Introduction to General-Purpose GPU Programming Taschenbuch – Illustriert, 19. Juli 2010, Englische Ausgabe, Jason Sanders / Kandrot (Autor)

(12) Raspberry Pi: Kompendium: Linux, Programmierung und Projekte Taschenbuch – 29. Juni 2020 von Sebastian Pohl (Autor)

(13) Das offizielle Raspberry Pi Handbuch: Von den Machern von MagPi, dem offiziellen Raspberry Pi Magazin Taschenbuch – 12. Februar 2024 von Elektor International Media (Herausgeber)

(14) Mastering Ubuntu Server - Fourth Edition: Explore the versatile, powerful Linux Server distribution Ubuntu 22.04 with this comprehensive guide Taschenbuch – 22. September 2022 Englische Ausgabe, Jay LaCroix (Autor)

(15) Linux Pocket Guide Taschenbuch – 23. Juni 2016, Englische Ausgabe, Daniel J. Barrett (Autor)

(16) OpenAI Chat GPT 4.0 (diverse Recherchen)