

Duale Hochschule Baden-Württemberg Mannheim

**Seminararbeit**

**Intelligente Spur- und Objekterkennung als Teil des  
autonomen Fahrens**

**Studiengang Wirtschaftsinformatik**

**Studienrichtung Data Science**

Verfasser und Matrikelnummer: Andreas Bernrieder - 7876007

Thorsten Hilbradt - 5034067

Jan Niklas Brebeck - 8016697

Simon Scapan - 6699329

Kurs: WWI18DSB

Studiengangsleiter: Prof. Dr. Bernhard Drabant

Wissenschaftlicher Betreuer: Prof. Dr. Bernhard Drabant

Bearbeitungszeitraum: 01.12.2020 – 25.01.2021

# Ehrenwörtliche Erklärung

Wir versichern hiermit, dass wir die vorliegende Arbeit mit dem Thema: *Intelligente Spur- und Objekterkennung als Teil des autonomen Fahrens* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Mannheim, den 25.01.2021

Andreas Bernrieder  
Andreas Bernrieder

Thorsten Hilbradt  
Thorsten Hilbradt

Jan Niklas Brebeck  
Jan Niklas Brebeck

Simon Scapan  
Simon Scapan

# **Abstract**

In dieser wissenschaftlichen Arbeit wird sich mit dem Projekt „Intelligente Spur- und Objekterkennung als Teil des autonomen Fahrens“ beschäftigt. Dieses Projekt ist Teil des Integrationsseminar an der Duale Hochschule Baden-Württemberg (DHBW) im Wintersemester 2020/21.

Im Verlauf der Arbeit wird dargestellt wie Object und Lane detection eingesetzt werden, um autonomes Fahren zu ermöglichen. Die Arbeit wurde hauptsächlich in drei Aufgabenbereiche aufgeteilt. In der Lane Detection wird ein Ansatz anhand maschinellen Lernens erläutert. Eine Ausführliche Implementierung wird dann mithilfe der weitreichenden Bibliothek OpenCV, welche in einem eigenen Kapitel betrachtet wird, erläutert. Die Object Detection betrachtet drei verschiedene Modelle, ein Region-based Convolutional Network (R-CNN), den Single Shot MultiBox Detector (SSD) und letztendlich You Only Look Once (YOLO). Mit letzterem wird das vorliegende Projekt auch durchgeführt. Im dritten Aufgabenbereich wird erläutert, wie die Modelle auf einem Raspberry Pi zum Steuern eines Modellautos verwendet werden können. Dazu wurde ein Modellauto mit einem Raspberry Pi ausgestattet, um dies mit den erstellten Methoden der Spur- und Objekterkennung zu steuern. Dieser Abschnitt unterteilt sich zum einen in Soft- als auch Hardwarebetrachtung.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>v</b>
<b>Quelltextverzeichnis</b>	<b>vi</b>
<b>Abkürzungsverzeichnis</b>	<b>vii</b>
<b>1 Einleitung</b>	<b>1</b>
<b>2 Computer Vision</b>	<b>2</b>
2.1 Einleitung . . . . .	2
2.2 Grayscaling . . . . .	2
2.3 Gaußsche Unschärfe . . . . .	4
2.4 Canny Transformation . . . . .	5
2.5 Hough Transformation . . . . .	7
<b>3 Spurerkennung</b>	<b>10</b>
3.1 Einführung . . . . .	10
3.2 State of the Art der Spurerkennung . . . . .	10
3.2.1 Umsetzung anhand maschinellen Lernens . . . . .	11
3.2.2 Bildmanipulation anhand OpenCV . . . . .	13
3.3 Implementierung mit OpenCV . . . . .	14
3.3.1 Eingangsdaten Vorverarbeitung . . . . .	15
3.3.2 Spurerkennung . . . . .	16
3.3.3 Ausgabedatenaufbereitung . . . . .	17
3.4 Steuerung . . . . .	18
<b>4 Objekterkennung</b>	<b>20</b>
4.1 Einführung . . . . .	20
4.2 Algorithmenvergleich . . . . .	20
4.2.1 Region-based Convolutional Networks . . . . .	21
4.2.2 Single Shot MultiBox Detector . . . . .	22
4.2.3 You Only Look Once . . . . .	23
4.3 YOLO Implementierung . . . . .	24
4.3.1 Vortrainierte Modelle . . . . .	24
4.3.2 Individuell trainiertes Modell . . . . .	26
4.3.3 Performance . . . . .	28
4.4 Distanzberechnung . . . . .	30

<b>5 Praktische Umsetzung</b>	<b>32</b>
5.1 Hardware . . . . .	32
5.1.1 Raspberry Pi . . . . .	32
5.1.2 Modellauto . . . . .	32
5.1.3 Verkabelung . . . . .	33
5.2 Steuerung des Modellautos . . . . .	34
5.2.1 Ansteuerung der GPIO-Pins mit Python . . . . .	34
5.2.2 Modellautosteuerung mittels Object- und Lanedetection . . . . .	35
<b>6 Fazit</b>	<b>37</b>
6.1 Spurerkennung . . . . .	37
6.2 Objekterkennung . . . . .	37
<b>Literaturverzeichnis</b>	<b>39</b>

# Abbildungsverzeichnis

Abbildung 2.1 Darstellung des Grayscaleing [29, S.2] . . . . .	3
Abbildung 2.2 Visualisierung des Gaussian Blur [19] . . . . .	4
Abbildung 2.3 Visualisierung der Canny Edge Detection . . . . .	5
Abbildung 2.4 Hough Transformation . . . . .	8
Abbildung 3.1 Spurerkennung in unterschiedlichen Situationen. [23, S.1] . . . . .	11
Abbildung 3.2 Architektur des beschriebenen hybriden Deep Neural Network [23, S.1] . . . . .	12
Abbildung 3.3 Ein- und Ausgabe der Datenstrecke von Behera [4] . . . . .	14
Abbildung 3.4 Datenstrecke Projekt SELMA . . . . .	15
Abbildung 4.1 R-CNN Aufbau, [12, S.2] . . . . .	21
Abbildung 4.2 SSD framework, [15, S.3] . . . . .	22
Abbildung 4.3 YOLO Detection System, [26, S.1] . . . . .	23
Abbildung 4.4 Model, [26, S.2f.] . . . . .	24
Abbildung 4.5 PlaymoDB Bild, Playmobil aus Google Bildersuche . . . . .	26
Abbildung 4.6 LabelImg Anwendung (links PlaymoDB, rechts aus Google Bildersuche) . . . . .	27
Abbildung 4.7 oben: Original, mitte: tinyYOLO, unten: YOLO . . . . .	29
Abbildung 4.8 Test Youtube Video . . . . .	30
Abbildung 4.9 Marker (Original links, Kontur rechts) . . . . .	31
Abbildung 5.1 PWM-Signal . . . . .	33
Abbildung 5.2 Graphische Darstellung der Verkabelung . . . . .	34

# Quelltextverzeichnis

2.1 Grayscale . . . . .	3
2.2 Gaussian Blur . . . . .	5
2.3 Canny Transform . . . . .	6
2.4 Hough Transformation . . . . .	8
3.1 Threshold Berechnung für Canny Edge Detection . . . . .	16
3.2 Steering Funktion aus der Pipeline . . . . .	18
4.1 Yolo Modelle . . . . .	25
4.2 Yolo Erkennung für jeweils ein Bild . . . . .	25
5.1 Initialisierung der General Purpose Input/Output (GPIO) . . . . .	35
5.2 Anpassung des Pulsweltenmodulation (PWM)-Signal . . . . .	35

# Abkürzungsverzeichnis

<b>DHBW</b>	Duale Hochschule Baden-Württemberg
<b>OD</b>	Object Detection
<b>OpenCV</b>	Open Source Computer Vision Library
<b>LiDAR</b>	Light Detection And Ranging
<b>YOLO</b>	You Only Look Once
<b>OR</b>	Object Recognition
<b>OC</b>	Object Classification
<b>CNN</b>	Convolutional Neural Network
<b>R-CNN</b>	Region-based Convolutional Network
<b>SVM</b>	Support Vector Machine
<b>FPS</b>	Frames per Second
<b>SSD</b>	Single Shot MultiBox Detector
<b>Pi</b>	Raspberry Pi
<b>GPIO</b>	General Purpose Input/Output
<b>SELMA</b>	<b>Selbstfahrendes Modellauto</b>
<b>PWM</b>	Pulsweitenmodulation
<b>LD</b>	Lane Detection
<b>GPS</b>	Global Positioning System
<b>RGB</b>	Grundfarben: Rot, Grün und Blau
<b>DCNN</b>	Deep Convolutional Neural Network
<b>DRNN</b>	Deep Recurrent Neural Network
<b>LSTM</b>	Long Short Term Memory

# 1 Einleitung

In dieser wissenschaftlichen Arbeit wird sich mit dem Projekt „Intelligente Spur- und Objekterkennung als Teil des autonomen Fahrens“ beschäftigt. Dieses Projekt ist Teil des Integrationsseminar an der DHBW Mannheim im Wintersemester 2020/21.

Das Ziel des Projektes ist es ein System zu entwickeln welches es ermöglicht Spuren und Objekten auf Bildern zu erkennen. In der Planungsphase wurde sich dafür entschieden eine reale Umsetzung des Projektes mithilfe eines ferngesteuerten Modellautos durchzuführen.

Die verwendeten Algorithmen und Methoden belaufen sich auf YOLO für die Objekterkennung und unterschiedliche Methoden aus Open Source Computer Vision Library (OpenCV) zur Spurerkennung.

YOLO ist ein Algorithmus, welcher eine abgewandelte Version eines Convolutional Neural Network (CNN) verwendet, um in einer kürzeren Zeit Objekte zu erkennen. Die genaue Implementierung und Verwendung des YOLO Algorithmus wird im Laufe der Arbeit detailliert beschrieben. Zusätzlich zur Objekterkennung wird hier auch die Entfernung zu den erkannten Objekten berechnet.

Die Spurerkennung basiert auf der OpenCV Bibliothek. OpenCV stellt viele verschiedene Algorithmen und Methoden zur Verfügung um Computer Vision umzusetzen. Die verwendeten Methoden und Algorithmen werden im entsprechenden Kapitel „Spurerkennung“ näher beschrieben.

Abschließend wird der Umbau des Modellautos erläutert. Dabei wird auf den Umbau der Elektronik, insbesondere dem integrieren eines Raspberry Pi (Pi) in die Steuerelektronik, eingegangen. Außerdem wird das Vorgehen beschrieben, wie der Pi das Modellauto mithilfe der Funktionen aus der Objekt- und Spurerkennung steuert.

# 2 Computer Vision

## 2.1 Einleitung

Die Computer Vision ist ein zentrales Element in diesem Projekt. Denn diese ermöglicht die von Kameras aufgenommenen Bilder mit der Hilfe von unterschiedlichen Algorithmen und mathematischen Methoden zu verarbeiten und zu analysieren. Dabei helfen sie deren Inhalt zu verstehen oder geometrische Informationen aus den Bildern zu extrahieren. [22]

Im Verlauf des Projekts wird OpenCV für die Objekterkennung sowie die Spurerkennung verwendet. OpenCV besteht aus mehr als 2500 optimierten Algorithmen zur Computer Vision. Diese Algorithmen können verwendet werden, um Gesichter, oder bestimmte Objekte zu erkennen und sich bewegenden Objekten in einem Videostream zu folgen. [18]

Die verwendeten OpenCV Methoden welche in diesem Projekt verwendet werden belaufen sich auf:

1. Grayscaleing (Kapitel 2.2)
2. Gauße Unschärfe (Kapitel 2.3)
3. Canny Transformation (Kapitel 2.4)
4. Hough Transformation (Kapitel 2.5)

Diese Methoden werden im Folgenden jeweils detailliert beschrieben.

## 2.2 Grayscaleing

Die Anwendung des Grayscaleing bedeutet, dass ein Bild, welches aus den Grundfarben: Rot, Grün und Blau (RGB) besteht in ein Schwarz-Weiß Bild mit verschiedenen Graustufen umgewandelt wird. Dadurch sind die Farben der Bilder nicht mehr in den drei verschiedenen Farben des RGB Farbraumes mit jeweils 255 Ausprägungen vorhanden, sondern lediglich in schwarz mit insgesamt 255 Ausprägungen.

Um diese Transformation durchzuführen, werden die einzeln Werte der drei Grundfarben verwendet, um zu bestimmen, was die gleichwertige Graustufe dieser Farbe ist.

Die Formel 2.1 [29, S.3] beschreibt, mit welchen Faktoren jedes Pixel aus dem übergebenen Bild in Graustufen umgerechnet wird.

$$\text{Graustufen} = 0.33 \times (\text{blauwert}) + 0.56 \times (\text{grünwert}) + 0.11 \times (\text{rotwert}) \quad (2.1)$$

Ein Beispiel dieser Anwendung wird in der Abbildung 2.1 dargestellt.

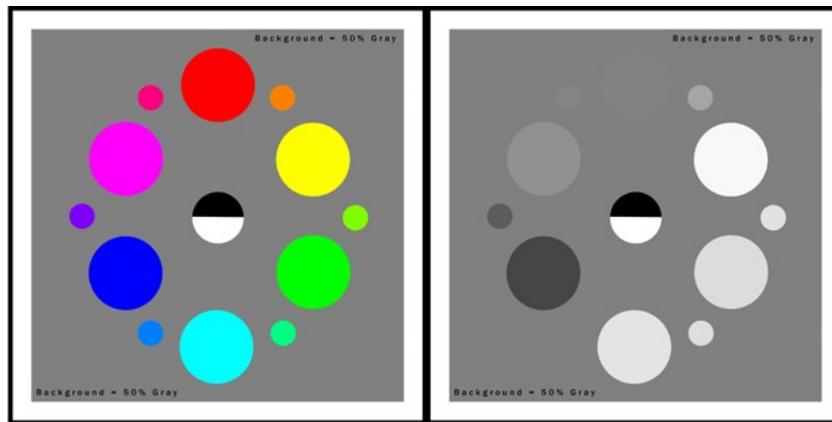


Abbildung 2.1: Darstellung des Grayscaleing [29, S.2]

Durch die Anwendung des Grayscaleing wird die Komplexität des Bildes, aufgrund der fehlenden Farbtiefe, stark reduziert. Außerdem führt das Grayscaleing dazu, dass verschiedene Elemente des Bildes nur noch als Kontrast zu erkennen sind.

Im Quelltext 2.1 wird dargestellt wie das Grayscaleing in den Quelltext für das Projekt eingearbeitet wird.

```
1 def grayscale(img):
2     return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Quelltext 2.1: Grayscaleing

Hier wird der Funktion `grayscale()` das Bild, welches transformiert werden soll, übergeben. Mithilfe der `cv2.cvtColor` Funktion wird das Bild in das `cv2.COLOR_BGR2GRAY` Format transformiert. Dadurch wird das Bild in Graustufen umgewandelt und anschließend zurückgegeben.

## 2.3 Gaußsche Unschärfe

Die Gaußsche Unschärfe oder auch bekannte als das weichzeichnen von Bildern, ist das verschwommen machen von Bildern mithilfe einer Gaußfunktion. Dies hilft dabei, ungewolltes Rauschen, in den Bildern zu verringern und so den Kontrast der Hauptelemente des Bildes zu erhöhen. [19, S.1]

In der folgenden Abbildung 2.2 ist dies dargestellt:



Abbildung 2.2: Visualisierung des Gaussian Blur [19]

Auf der linken Seite der Abbildung 2.2 ist das Original dargestellt. Hier existiert um die einzelnen Elemente viel ungewolltes rauschen. Um es nun einem System zu erleichtern das Bild zu verarbeiten, wird das Bild mit der Hilfe eines Weichzeichners bearbeitet. So werden die Elemente auf der rechten Seite der Abbildung 2.2 klarer erkennbar, was unter anderem darauf zurückzuführen ist, dass die Linien der einzelnen Elemente nicht mehr so viele Unreinheiten haben. [19]

Dies ist eine große Hilfe in der Spurerkennung, da je stärker das Weichzeichnen ist, desto geringer wird die Anzahl der Linien die erkannt werden jedoch nicht die gesuchte Spur darstellen. Dadurch werden die Linien besser erkannt welche einen höheren Kontrast zu der Umgebung haben. Dies wird im Abschnitt 2.4 detailliert beschrieben.

Die Methode die dafür im System verwendet wird ist im Quelltextbeispiel 2.2 dargestellt.

```
1 def gaussian_blur(img, kernel_size):  
2     return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)
```

Quelltext 2.2: Gaussian Blur

Hierbei wird die Methode *cv2.GaussianBlur* verwendet. Es wird das Bild übergeben, welches vorher in seine Graustufen herunter gebrochen wurde. Zusätzlich wird die Kernelgröße übergeben. Diese muss immer positiv und eine ungerade Zahl sein. In diesem Fall sind X und Y gleich groß.

## 2.4 Canny Transformation

Die Canny Transformation oder auch bekannt als Canny Edge Detection ist ein Ansammlung an Algorithmen, welche es ermöglicht, Kanten in einem Bild zu erkennen.

In der Abbildung 2.3 ist sichtbar, wie diese Transformation aussieht.

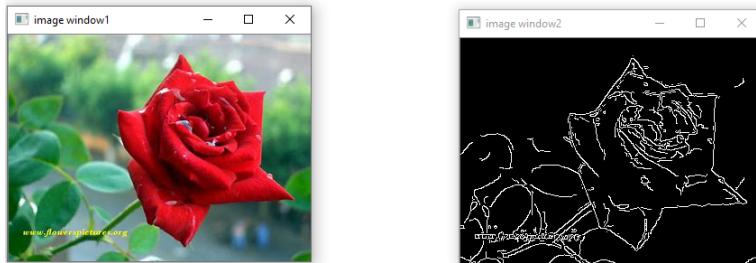


Abbildung 2.3: Visualisierung der Canny Edge Detection  
Quelle: <https://java2blog.com/cv2-canny-python/>

Die Canny Tranformation kann in den Folgenden 5 Schritten beschrieben werden.

1. Grayscale / Gaußsche Unschärfe
2. Bildung eines Intensitätsgradienten
3. Nicht-Maximum Unterdrückung
4. Doppelter Schwellwert

## 5. Hysterese Schwellwert

Die Canny Transformation beruht darauf, dass im ersten Schritt das in Sektion 2.2 beschriebene Grayscale und die in Sektion 2.3 beschriebene Gaußsche Unschärfe bereits auf das zu bearbeitenden Bild angewendet wurden.

Im nächsten Schritt werden die Intensitätsgradienten für das Bild bestimmt. Das bedeutet, dass die Richtungen, in welche die Kanten eines Bildes zeigen, mithilfe eines Filters bestimmt werden. Dieser Filter gibt den Gradienten  $G_x$  in der horizontale und  $G_y$  in der vertikalen Richtung für jeden Pixel an. Aus den Werten lässt sich die Richtung  $\theta$  und der Wert des Pixel  $G$  bestimmen.[17, S.4]

Daraufhin wird eine nicht-maximale Unterdrückung durchgeführt. Das bedeutet, dass alle Werte, welche nicht die Maximalwerte der Intensitätsgradienten sind, als Null gezählt und daher unterdrückt werden.[5, S.5]

Da jedoch weiterhin einzelne Randpixel verbleiben, welche durch Rauschen oder Farbabweichungen entstanden sind, müssen diese im nächsten Schritt entfernt werden. Dazu wird ein oberer und unterer Schwellenwert festgelegt. Wenn also ein Pixel einen Wert hat, welcher größer ist als der obere Schwellenwert, wird dieser als starker Randpixel markiert. Wenn der Wert des Pixel zwischen den oberen und unteren Schwellenwert fällt, wird dieser als schwaches Randpixel markiert und wenn der Wert des Pixel unterhalb des unteren Schwellenwertes fällt, wird dieser unterdrückt.[17, S.5]

Im letzten Schritt wird untersucht, ob die als schwach markierten Randpixel mit den als stark markierten Randpixeln verbunden sind. Wenn das der Fall ist wird die schwache Kante behalten. Wenn nicht, wird die schwache Kante unterdrückt. Dies ist auch bekannt als ein Hysterese-Schwellenwert, welcher hier verwendet wird.[17, S.5]

Wie die Implementierung in diesem Projekt aussieht ist im Quelltextbeispiel 2.3 zu erkennen.

```
1 def canny(img, low_threshold, high_threshold):
2     return cv2.Canny(img, low_threshold, high_threshold)
```

Quelltext 2.3: Canny Transform

Hier wird die `cv2.Canny` Funktion der OpenCV Bibliothek aufgerufen. Dieser wird das zuvor bereits verarbeitete Bild übergeben und die Werte für den oberen und unteren Schwellwert.

## 2.5 Hough Transformation

Die Hough Transformation wird verwendet, um Linien in einem Bild zu erkennen. Das bedeutet, dass bei der Hough Transformation, die Linien erkannt werden, welche mit Hilfe der in Abschnitt 2.4 beschriebenen Canny Transformation in das Bild gezeichnet wurden. Ohne diese Vorverarbeitung wäre eine Anwendung der Hough Transformation nicht möglich. [20]

Im Prozess der Hough Transformation wird ein Dual-Raum Aufgebaut. Dieser besteht aus dem vorverarbeiteten Kantenbild und einem Parameterraum.

Nun wird für jedes Pixel, das auf einer Gerade liegt, ein Punkt in den Parameterraum projiziert. Dafür wird für jeden Punkt der Y-Achsenabschnitt und der Gradient berechnet. Anschließend wird dies in ein alternatives Koordinatensystem übertragen, welches aus dem Y-Achsenabschnitt und dem Gradient besteht auch bekannt als das Polarkoordinatensystem.[10]

Da diese Operation nur den Wert für einen Punkt ausgibt und eine Linie benötigt, wird diese Operation für alle möglichen Winkel durchgeführt. Dies ist in Abbildung 2.4 gut zu erkennen. Dort werden für alle Punkte auf dem Eingabebild die entsprechende Linie auf dem mittleren Koordinatensystem gezeichnet.[10]

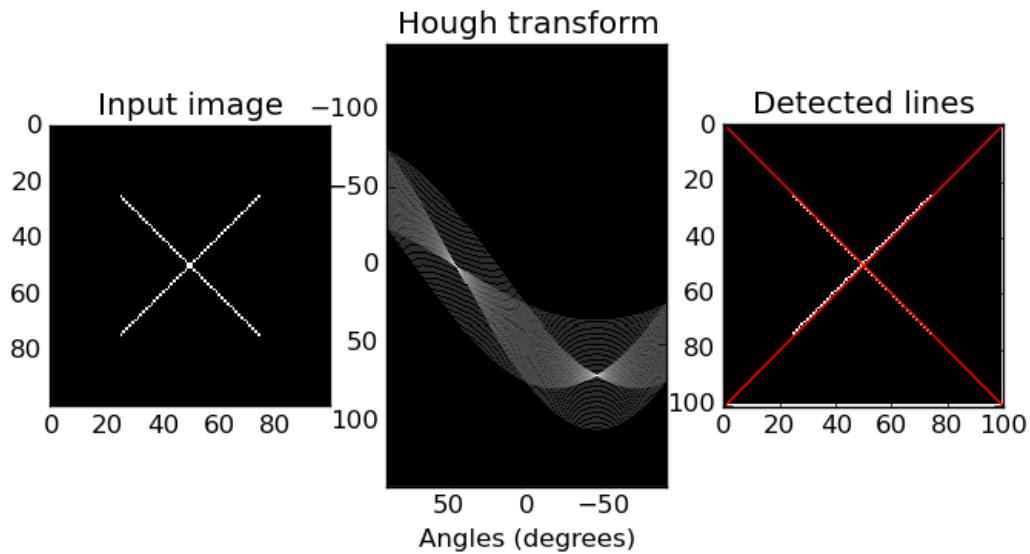


Abbildung 2.4: Hough Transformation

Quelle:

[https://scikit-image.org/docs/0.11.x/auto\\_examples/plot\\_line\\_hough\\_transform.html](https://scikit-image.org/docs/0.11.x/auto_examples/plot_line_hough_transform.html)

Die Kanten werden nun in dem Winkel gezeichnet, wo die Linien sich im Parameterraum am häufigsten überschneiden. Das bedeutet sie werden dort gezeichnet wo der Y-Achsenabschnitt und der Gradient der Kanten aus dem Orginalbild sich am häufigsten überschneiden bzw. gleich sind.[10]

Die Implementierung, wie sie hier verwendet wird, ist im Quelltextbeispiel 2.4 zu erkennen.

```

1 def hough_lines(img, rho, theta, threshold, min_line_len,
2                 max_line_gap):
3     lines = cv2.HoughLinesP(img, rho, theta, threshold, np.
4                             array([]), minLineLength=min_line_len, maxLineGap=
4                             max_line_gap)
5     line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=
5                         np.uint8)
6     draw_lines(line_img, lines)
7     line_img = slope_lines(line_img, lines)
8     return line_img

```

Quelltext 2.4: Hough Transformation

Hier wird die Funktion `cv2.HoughLinesP` aus der OpenCV Programmierbibliothek verwendet. Dazu werden die benötigten Parameter übergeben, welche unter anderem die Geometrie der Berechnung beeinflussen, aber auch eine maximale und minimale Breite der Linien, sodass eine korrekte Erkennung garantiert werden kann. Ausgegeben wird das Bild mit den eingezeichneten Linien.

# 3 Spurerkennung

## 3.1 Einführung

Eine der wichtigsten Funktionalitäten beim autonomen Fahren ist die Spurerkennung und die damit einhergehende Steuerung des Fahrzeugs im Bereich der Fahrbahnlinien. Das Forschungsgebiet wird unter dem englischen Begriff Lane Detection (LD) zusammengefasst.

Es gibt unterschiedliche Ansätze und Modelle, wie eine Fahrbahnlinienerkennung implementiert werden kann. Zum einen durch den Einsatz von künstlicher Intelligenz, welche anhand gelernter Datensätze neue Spuren erkennen und zum Anderen gibt es den Ansatz Fahrspuren allein mithilfe Bildbearbeitender Werkzeuge zu erkennen. Da schon tief reichendes Wissen im Bereich der Bildbearbeitung im Team vorhanden ist, wird dieses Prinzip weiter verfolgt.

In den folgenden Unterkapiteln wird auf die verschiedenen Möglichkeiten der Spurerkennung eingegangen, dabei hilft insbesondere die Python Bibliothek OpenCV, welche im vorherigen Kapitel 2 ausführlich beschrieben wurde und hierfür als Grundlage dient. Aufbauend darauf wird eine Datenstrecke implementiert und erläutert, um am Ende diesen Kapitels den Übergabewert für die Steuerung zu berechnen.

## 3.2 State of the Art der Spurerkennung

Für die Spurerkennung werden, wie bei der Objekterkennung im Regelfall mehrere Informationsquellen verarbeitet. Das sind vor allem Kamera- und Sensordaten aber auch Informationen mithilfe des Global Positioning System (GPS) darüber, wo sich das Fahrzeug zur Zeit befindet. Für dieses Projekt werden ausschließlich Kameradaten verwendet, was sich auch bei der Auswahl der Modelle widerspiegelt.

Wie in der Einführung in dieses Kapitel bereits angemerkt, kann für die Spurerkennung ein rein bildbearbeitender Ansatz oder maschinelles Lernen genutzt werden. Beide Bereiche werden im Folgenden betrachtet.

### 3.2.1 Umsetzung anhand maschinellen Lernens

In diesem Unterkapitel wird die Lane Detection mit einem Ansatz des maschinellen Lernens betrachtet. Im Detail wird auf einen Ansatz im Bereich des Supervised Learning gesetzt.

Der Hintergrund ist, dass eine Szene nicht einzeln betrachtet werden kann. Wenn nur eine einzelne Momentaufnahme herangezogen wird, um eine Entscheidung zu treffen, wird diese schlechter ausfallen, als wenn zusätzlich in die Vergangenheit geschaut wird. [23] Das heißt im Detail, dass zum Beispiel das aktuelle Bild analysiert wird und dazu die letzten drei, jedoch mit einer geringeren Gewichtung. Damit kann das Ziel erreicht werden, Fehler in der Erkennung zu vermeiden, da aus den vorherigen Eingangsdaten der Verlauf der Straße hervorgeht und der aktuelle Input dann die jüngste Veränderung aufweist. Die Notwendigkeit kann multipel begründet werden. Zum einen kann die Fahrbahnmarkierung verschmutzt oder durch Fahrzeuge und Gegenstände unterbrochen sein. In den meisten Fällen sind es jedoch Schatten, schlechte Lichtverhältnisse, Tunnel aber vor allem auch Dreck auf der Straße. Die Performance bei der Nutzung auch vorangegangener Bilder, kann der folgenden Abbildung 3.1 entnommen werden.



Abbildung 3.1: Spurerkennung in unterschiedlichen Situationen. [23, S.1]

Erste Reihe: Drei verschiedene Fahrszenen. Zweite Reihe: Erkennung nur anhand des aktuellen Bildes. Dritte Reihe: Spurerkennung anhand des aktuellen und der letzten vier Bilder.

Es gibt hauptsächlich zwei Arten von Deep Neural Networks. Zum einen Deep Convolutional Neural Network (DCNN), welches Eingabedaten häufig mit mehreren Stufen der

Convolution verarbeitet und gut geeignet für die Abstraktion von Merkmalen für Bilder und Videos ist. Zum anderen das Deep Recurrent Neural Network (DRNN), das Eingangssignale rekursiv verarbeitet, indem es diese in aufeinander folgende Blöcke aufteilt und vollständige Verbindungsschichten für die Statusausbreitung zwischen ihnen aufbaut. Die Vorteile zeigen sich vor allem bei der Vorhersage von Informationen für Zeitreihendaten.[23]

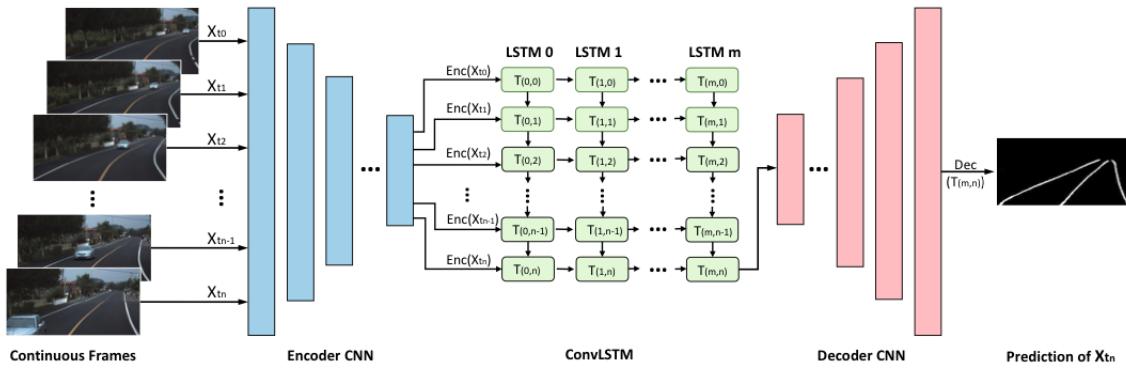


Abbildung 3.2: Architektur des beschriebenen hybriden Deep Neural Network [23, S.1]

Das Team um Qin Zou [23] erhofft sich aus der Kombination eben dieser beiden Netzwerke performante Ergebnisse. Die Architektur des Netzwerkes ist der Abbildung 3.2 zu entnehmen. So wird ein hybrides Deep Neural Network zur Spurerkennung implementiert. Dabei werden mehrerer kontinuierliche Fahrszenen als Bilder verarbeitet. Das hybride Netzwerk verbindet also die Ansätze von DCNN und DRNN. Global betrachtet ist das vorgeschlagene Netzwerk ein DCNN, welches mehrere Rahmen als Eingabe verwendet und die Spur des aktuellen Rahmens auf Segmentierungsweise semantisch vorhersagt. Eine vollständig gefaltete (fully convolution) DCNN-Architektur wird benutzt, um das Segmentierungsziel zu erreichen. Es enthält ein Encoder-Netzwerk und ein Decoder-Netzwerk, diese garantieren, dass die endgültige Ausgabe dieselbe Größe, analog zum Eingabebild hat. Aus der lokalen Perspektive werden vom Encoder-Netzwerk die von DCNN abstrahierten Merkmale von einem DRNN weiterverarbeitet. Ein Long Short Term Memory (LSTM) Netzwerk wird verwendet, um die Zeitreihen codierter Merkmale zu verarbeiten. Die Ausgabe des DRNN führt die Informationen der kontinuierlichen Eingangsrahmen zusammen, welche dann in das Decoder-Netz des DCNN eingespeist werden, um dann letztendlich die Spuren vorherzusagen beziehungsweise die Erkennung zu unterstützen. [23]

### 3.2.2 Bildmanipulation anhand OpenCV

Ein weiterer Ansatz ist die Nutzung bildmanipulatorischer Funktionen. Da, wie eingangs beschrieben, im Team viel Wissen in diesem Bereich vorherrscht, sind Grundlagen vorhanden. Für den speziellen Anwendungsfall wurde sich der Datenstrecke von Soumya Ranjan Behera [4] als grober Anhaltspunkt bedient. Nachfolgend wird die Datenstrecke erläutert und auf die wichtigsten Punkte eingegangen.

Am Anfang steht der Inputstream der zu verarbeitenden Datei. Dieser wird in ein Bild aus Graustufen umgewandelt. Zum einen bringt es den Vorteil, dass eine Unterscheidung verschiedener Farben der Fahrbahnmarkierung, wie sie aus zum Beispiel Baustellenbereichen bekannt ist, entfällt. Des Weiteren kann dadurch die zu verarbeitende Dateigröße um zwei drittel verringert werden. Dieser Effekt spiegelt sich hauptsächlich in der Verarbeitungszeit der späteren Prozesse wieder. Der Detaillierte Prozess ist im Kapitel Grayscale näher erläutert.

Das in Graustufen transformierte Bild wird mithilfe einer Gausschen Funktion weich gezeichnet. Die Funktionalität des Gausschen Weichzeichnens ist unter Gaußsche Unschärfe nachzulesen. Die Weichzeichnung ist notwendig, da so Details reduziert und somit die Erkennung der Spur verbessert wird. Algorithmen zur Erkennung von Kanten reagieren meist empfindlich auf Rauschen. Damit wird also eine weitere potenzielle Fehlerquelle bereinigt.

Die Erkennung der Kanten folgt mithilfe der aus OpenCV stammenden Funktion Canny Edge Transformation, welche unter dem Kapitel Canny Transformation grundlegend erklärt wird. Anhand dieser werden dann Kanten, also Kontrastsprünge, erkannt und als Linien ausgegeben. Die Ausgabe ist wie folgt vorzustellen: die Kanten sind weiß auf schwarzem Hintergrund zu sehen. Mit dem menschlichen Auge kann hier schon klar die Spur erkannt werden. Jedoch muss in vielen Fahrszenarien der zu betrachtende Bereich eingegrenzt werden, da sonst auch Wolken, Häuser oder andere Gegenstände mit ihren Kanten eingezeichnet sind.

Die Eingrenzung des Bereiches wird mit Masken implementiert. Hierfür wird anhand prozentualer Angaben von Höhe und Breite des Bildes eine polygonale Maske auf das Bild gelegt. Die Maske ist analog der Farbe des Hintergrunds, was zum Ergebnis führt, dass nur noch die Fahrbahnbegrenzungen als Kanten im Bild zu sehen sind.

Diese Kanten werden anhand der Hough Lines Transformation zu Linien der Form  $y = m * x + b$  umgeformt. Das Verfahren ist unter dem Kapitel Hough Transformation näher

beleuchtet. Anhand der errechneten Linien kann dann die Fahrbahn, also der Bereich zwischen linker und rechter Fahrbahnmarkierung angezeigt werden. In diesem Bereich darf sich das Fahrzeug bewegen. Sind die beiden Linien und der Bereich dazwischen markiert, wird zum Schluss für eine bessere Ausgabe der Ergebnisse das ursprüngliche Bild des Streams als Hintergrund eingefügt.

Ein mögliches Ergebnis, im Input - Output Vergleich, kann der folgenden Abbildung 3.3 entnommen werden.



Abbildung 3.3: Ein- und Ausgabe der Datenstrecke von Behera [4]

### 3.3 Implementierung mit OpenCV

Im Team wurde die Entscheidung getroffen, die Spurerkennung anhand der bildbearbeitungs Methodik zu implementieren. Dazu wird sich der im vorherigen Kapitel erläuterten Datenstrecke von Behera [4] bedient. Genutzt wird dabei die Python Bibliothek OpenCV. Die wichtigsten benutzten Funktionalitäten des Pakets sind unter dem Kapitel Computer Vision tiefgreifend erläutert.

Die nachfolgenden Kapitel beinhalten die einzelnen Schritte der Datenstrecke, welche Produktiv für das Projekt **Selbstfahrendes Modellauto (SELMA)** genutzt wird. Für eine bessere Verständlichkeit wird die Thematik in die drei Bereiche

- Eingangsdaten Vorverarbeitung
- Spurerkennung
- Ausgabedaten Aufbereitung

unterteilt. Um die einzelnen Schritte visuell nachvollziehen zu können, werden die Inhalte anhand folgender Abbildung 3.4 Schrittweise durchgearbeitet.

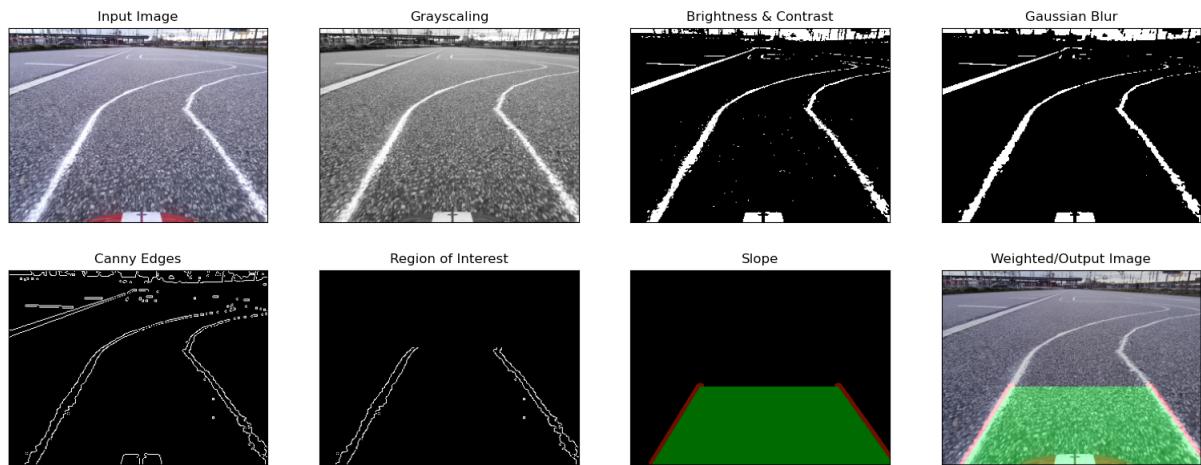


Abbildung 3.4: Datenstrecke Projekt SELMA

### 3.3.1 Eingangsdaten Vorverarbeitung

Die von der Kamera gelieferten Daten durchlaufen in der Vorverarbeitung drei Schritte:

1. Grayscale
2. Helligkeit und Kontrast
3. Gaußsche Unschärfe

Die Schritte Grayscale und Gaußsche Unschärfe sind in den Kapiteln 2.2 und 2.3 bereits tiefreichend erläutert und werden deswegen nicht näher betrachtet.

Bei Außentests ist ein vorher nicht bedachtes Problem aufgetreten. Mit dem Auge wirkt ein asphaltierter Parkplatz schwarz, was die Grundlage der Überlegung war. Betrachtet man den Parkplatz jedoch aus Sicht eines Modellautos, also um einiges näher am Boden, ist festzustellen, dass Asphalt viele helle Steine hat. Diese wurden zu einem großen Problem, denn der Unterschied zwischen der weißen Linie aus Kreide und den Punkten auf dem Asphalt war nicht groß genug. Sogar für das menschliche Auge nur schwer zu unterscheiden. Das Problem kann gelöst werden, indem der Kontrast sehr hoch und die Helligkeit sehr niedrig gesetzt wird. In der Grafik Datenstrecke Projekt SELMA ist dieser Effekt deutlich zu erkennen. Im Abschnitt Grayscale ist zu sehen, dass die Fahrbahn keine einheitliche Farbe aufweist. Im nächsten Schritt Brightness Contrast hebt sich die Fahrbahnmarkierung dann deutlich von der Straße ab. Wie aus der Abbildung Datenstrecke Projekt SELMA zu

sehen, ist die erste Zeile der Eingangsdatenvorverarbeitung zugewiesen. Das zeigt auf die Wichtigkeit dieser Prozedur.

### 3.3.2 Spurerkennung

Nach der Vorverarbeitung liegt ein Bild vor, welches weiße Linien auf schwarzem Hintergrund enthält. (siehe Gaussian Blur in Datenstrecke Projekt SELMA) Der nächste Schritt ist es, Kanten zu erkennen. Dies ist Notwendig, um später daraus Linien zu berechnen. Dafür wird die Funktion Canny Edges aus der Bibliothek OpenCV genutzt. Details zur Funktionalität sind unter sec:CannyTransformation beschrieben. Die Besonderheit bei der Nutzung im Projekt SELMA ist, dass die beiden Threshold Parameter für jedes Bild während der Laufzeit berechnet und nicht statisch festgelegt werden. Die Berechnung im Quelltext nach Alkesaiberi [2] ist folgende:

```

1 # smoothed_img ist das Resultat der Gaussian Blur Funktion
2 med_val      = np.median(smoothed_img)
3 lower        = int(max(0 ,0.7*med_val))
4 upper        = int(min(255,1.3*med_val))
5
6 canny_img   = canny(img = smoothed_img ,
7                      low_threshold = lower,
8                      high_threshold = upper)

```

Quelltext 3.1: Threshold Berechnung für Canny Edge Detection

Der Threshold ist der Bereich in dem eine Kante als solche vom Algorithmus erkannt wird. Der Bereich ist zwischen 0 und 255, da ein Pixel nicht mehr Ausprägungen haben kann. Die Berechnung wird im Folgenden anhand des unteren Threshold erläutert. Für die Berechnung wird im Vorhinein der Mittelwert der Farbtiefe aller Pixel berechnet und der Variable *med\_val* zugewiesen. Der untere Schwellenwert ist demzufolge der höchste Wert zwischen 0 und 0.7 mal dem Mittelwert der Farbtiefe. Der höchste Wert, da ein Schwellen-Wertebereich zwischen 0 und 255 alle Kanten als solche erkennen, und somit die Notwendigkeit eines Threshold überflüssig machen würde. Für den oberen Threshold wird analog der kleinste Wert aus der Berechnung benutzt. Das Resultat der Canny Edge Detection ist der Abbildung Datenstrecke Projekt SELMA unter Canny Edges zu entnehmen.

Bei der Betrachtung diesen Resultates fällt auf, dass auch die Umgebung um die eigentliche Fahrbahn erkannt und deren Kanten ebenfalls umrisseen werden. Da dies bei der Spurerkennung zu Fehlern führen kann, wird der zu betrachtende Bereich eingegrenzt. Dies wird mithilfe der Funktion `cv2.fillPoly` aus OpenCV [9] implementiert. Hierfür wird ein Trapez in der Funktion `get_vertices` mit prozentualen Angaben anhand des zu berechnenden Bildes erstellt. Dieses wird dann an die Funktion `region_of_interest` übergeben, welche die Fläche des Trapezes mit Bildpunkten füllt. Der Bildbereich ist danach eingeschränkt, wie aus der Abbildung Datenstrecke Projekt SELMA unter „Region of Interest“ zu sehen ist.

Das Bild verfügt nun lediglich der vom Canny Edge Algorithmus erkannten Linien. Auf dieser Grundlage werden aus den Linien die Hough Lines berechnet. Dafür wird die Funktion `HoughLines` aus OpenCV benutzt, welche unter Kapitel Hough Transformation ausführlich erläutert wurde. Aus der Berechnung ergeben sich dann jeweils die erkannten Fahrbahn Linien auf der linken und rechten Seite.

Die Liniendaten werden Anhand der Funktion `slope_lines` von Behera [4] berechnet und liegen danach als Parameter  $m$  (Slope) und  $c$  (Intercept) in der folgenden Form vor:

$$y = m * x + b \quad (3.1)$$

### 3.3.3 Ausgabedatenaufbereitung

Nach Umformung der Linien in die allgemeine Form einer linearen Funktion in Spurerkennung können diese weiterverarbeitet werden. In der Funktion `slope` werden mithilfe der OpenCV Funktionen `line` und `fillPoly` zum einen die Linien als auch der Bereich dazwischen gekennzeichnet. Die Ausgabe dieser ist in der Abbildung Datenstrecke Projekt SELMA unter Slope zu sehen. Hierbei ist zu erkennen, dass die Linien bereits denen der vorherigen Ausgabe: „Region of interest“ gleichen. Um die gezeichneten Linien und die Spur auf dem Eingangsbild zu sehen, wird die OpenCV Funktion `addWeighted` benutzt. Dieser Funktion werden das Eingangsbild und das davor mit den Linien und der Spur gekennzeichnete Bild übergeben. Als Parameter wird  $\alpha = 0.8$  übergeben, was bedeutet, dass das Linienbild mit einer Transparenz von 80% auf das Eingangsbild gelegt wird. Die Ausgabe und damit auch das Resultat der Datenstrecke, ist der Abbildung Datenstrecke Projekt SELMA unter „Weighted/Output Image“ zu entnehmen.

## 3.4 Steuerung

Die Steuerung ist ein wichtiger Bestandteil des Projektes SELMA, denn nur aus der Ausgabe der eingezeichneten Spur kann das Fahrzeug nicht lenken. Es muss ein Parameter übergeben werden, welcher besagt, ob sich das Fahrzeug in der Mitte der Spur befindet oder nicht.

Als Eingangsparameter für die Berechnung werden das Eingabebild als auch die rechte und linke berechnete Linie herangezogen. Die Funktion für die Berechnung sieht wie folgt aus:

```

1 def steer(image, left_line, right_line):
2     img_y, img_x = image.shape[:2]
3
4     y = int(img_y*0.6) # height of slope
5
6     slope_l = (y - left_line[1]) / left_line[0]
7     slope_r = (y - right_line[1]) / right_line[0]
8     slope_with = (slope_r - slope_l)
9
10    x_slope = int( ( slope_with / 2) + slope_l )
11    x_img = int(img_x/2)
12
13    if slope_with <85:
14        return None
15    else:
16        steering = (x_img - x_slope) /100
17        return steering

```

Quelltext 3.2: Steering Funktion aus der Pipeline

Um die Steuerung auf den Bereich der eingezeichneten Linien zu begrenzen und damit die Weitsich zu verringern, wird der Wert in Zeile 4 um 40% verringert. Danach werden x-Werte für die linke und rechte Spur im höchsten Punkt, also  $y_1$  in Zeile 6 und 7 berechnet. Das folgt aus der Umstellung der allgemeinen linearen Gleichung nach  $x$ :

$$y = m * x + b \Rightarrow x = \frac{y - b}{m} \quad (3.2)$$

Die Breite der Spur wird berechnet und dient später als Indiz für fehlerhafte Erkennung der Spur. Wenn die Spurbreite zu gering ist, wird davon ausgegangen, dass ein Fehler vorliegt und es wird kein Ergebnis übergeben.

Um eine Aussage zu treffen, welcher Lenkeinschlag gewählt werden muss, wird in Zeile 10 der Mittelpunkt der beiden Linien und in Zeile 11 der Mittelpunkt des Bildes berechnet. Darauf folgend kann aus den beiden Mittelpunkten die Abweichung ermittelt werden. In Zeile 16 wird als Grundlage der Bildmittelpunkt herangezogen und davon der Mittelpunkt der Spur abgezogen. Wenn sich das Fahrzeug im linken Bereich der Spur befindet, also der `x_slope` Wert größer als der `x_img` Wert ist, wird eine positive Zahl übergeben, welche den prozentualen Lenkeinschlag nach rechts beinhaltet und vice versa.

# 4 Objekterkennung

## 4.1 Einführung

Im Rahmen des autonomen Fahrens spielt die Objekterkennung eine bedeutende Rolle. Eine möglichst fehlerfrei agierende Object Detection (OD) bildet die Basis für das sichere Fortbewegen eines autonomen Fahrzeugs. Hierbei kommen im industriellen Maßstab verschiedene Technologien zum Einsatz, vornehmlich die Erkennung anhand von Radar und / oder Light Detection And Ranging (LiDAR) Sensor Daten, sowie die Klassifizierung von Kamerabildern. Die LiDAR und die Radar Erkennung basiert hierbei auf demselben Prinzip, nämlich die Reflektion von Laserstrahle bzw. Radarwellen, wodurch sich ein dreidimensionales Abbild der Umgebung darstellen lässt [8, S.30f.]. Bis auf wenige Ausnahmen, wie Tesla, setzen alle Entwickler autonomer Fahrzeuge auf die LiDAR Technologie als Ergänzung der Kameragestützten Erkennung [3]. Die Befürworter dieser Technologie sehen sie vor allem als Ergänzung der Kamera und Radarerkennung, während die Kritiker auf die hohen Preise der LiDAR Sensoren verweisen, sowie auf annährend genau so hohe Erkennungsrauten durch reine Kamera OD [33, S.8]. Demzufolge können Kameras als Hauptwerkzeug für die OD benutzt werden, wobei sie bei schwierigen Sichtverhältnissen (Schnee, Regen, etc.) von Radarsensoren unterstützt werden. Im Rahmen dieses Projektes bzw. dieser Arbeit wird mittels der Kamera OD gearbeitet, wobei nur eine einzelne Kamera zum Einsatz kommt, was Auswirkungen auf die Distanz Berechnungen hat (siehe Kapitel 4.4). Für die OD verschiedener Objekte und Personen auf Bildern gibt es verschiedene Technologien. Einige davon werden in Kapitel 4.2 näher betrachtet, miteinander verglichen und bewertet, bevor sich Kapitel 4.3 mit der Implementierung des für dieses Projekt verwendeten Algorithmus, sowie Beobachtungen bei der Anwendung befasst. Abschließend wird in Kapitel 4.4 dargelegt, wie die Distanz zu erkannten Objekten berechnet wird.

## 4.2 Algorithmenvergleich

Die OD setzt sich aus zwei Teilbereichen zusammen, die jeweils auch eigenständig funktionieren können, der Object Recognition (OR) und daran anschließend die Object Classification (OC). Die OC ist dabei im Grundsatz eine Image Classification, also ein Supervised

Machine Learning Problem, bei dem Bilder verschiedener Objekte gelabelt und mithilfe eines CNN erlernt werden [16, S.615f.]. Die OR hingegen soll visuell verdeutlichen, wo sich erkannte Objekte im Bild befinden. Dies wird üblicherweise mithilfe farbcodierter Boxen, die um die einzelnen Objekte gelegt werden erreicht.

### 4.2.1 Region-based Convolutional Networks

Girshick et all stellten 2013 sogenannte R-CNN zum Zwecke der Object Detection vor [12] und beschreiben damit eine frühe Methode der effizienten OD, die durch weitere Paper 2015 [11] verbessert und 2016 [28] als Echtzeitalgorithmus angewendet wurden.

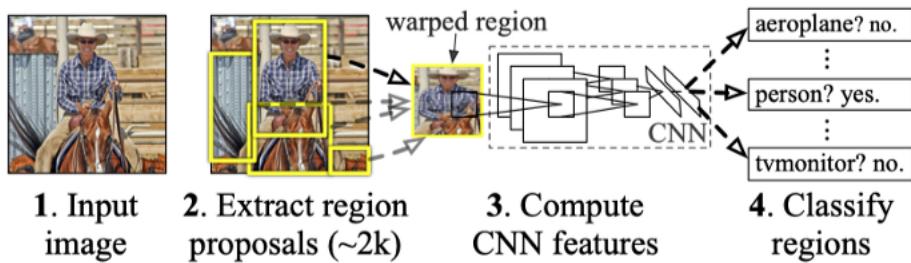


Abbildung 4.1: R-CNN Aufbau, [12, S.2]

Abbildung 4.1 zeigt den Aufbau des zuerst beschriebenen R-CNN. Hierbei greifen drei Module ineinander [12, S.3-6]. Die Aufgabe des ersten Moduls ist die Erzeugung von Region Proposals (dt. Region Vorschlag) für das Bild. Es sollen also die Regionen markiert bzw. ausgegeben werden in denen sich mögliche Objekte befinden. Hierfür wurde auf die Selective Search [31] zurückgegriffen. Die Autoren verweisen auf die Schwierigkeit mit Überlappungen, also dem sich nur teilweise in einer Region befindenden Objekt. Diese Probleme wurden durch das Einführen eines Tresholds von 0.3 [12, S.5] gelöst. Alle Regionen, die unter diese Grenze fallen, werden als negativ betrachtet und werden somit nicht weiter verarbeitet. Auf den gefundenen Regionen aufbauend arbeitet nun ein CNN, im speziellen eine Implementation nach Jia et all. desselben [13]. Die gewünschte Ausgabe dieses CNNs ist ein Feature Vektor, der an das letzte Modul, einer Anzahl Support Vector Machine (SVM), übergeben wird. Es handelt sich dabei um klassenspezifische SVM, die im späteren Verlauf eine weitere Rolle übernehmen. Bei Erkennen eines Objekts wird durch den Output der SVM eine klassenspezifische Bounding Box um das Objekt gelegt, um die genaue Lokalisierung zu verbessern [12, S.12f.]. Wie bereits beschrieben wurde der ursprüngliche Ansatz der R-CNN mehrfach verbessert, über das Fast R-CNN [11] hin zum

Faster R-CNN [28]. Diese zielen vornehmlich auf die schnellere Berechnung und Verarbeitung der Region Proposals, die den zeitaufwendigsten Schritt der OD darstellt [28, S.1]. Die Verarbeitungszeit wird bei Fast R-CNN reduziert, indem nicht mehr für jede Region Proposal durchgeführt werden muss, es werden Berechnungen geteilt. Das CNN nimmt als Input das Bild und alle Region Proposals und erzeugt darauf aufbauend eine Feature Map, die für die Erzeugung der Feature Vektoren benutzt wird. [28, S.2f.]. Faster R-CNN verbessert die Gesamlaufzeit weiter, indem die Berechnungszeit der Region Proposals verringert wird. Hierfür werden Regional Proposal Networks genutzt, die als Ergänzung der bisher benutzten CNN dienen können [28, S.3f.]. Durch dieses Modul kann auf dem verwendeten Setup der Autoren eine OD bei 5 Frames per Second (FPS) durchgeführt werden.

#### 4.2.2 Single Shot MultiBox Detector

Trotz der erreichten Verbesserungen der R-CNN erreichen diese dennoch keine zufriedenstellende Performance für Echtzeit OD. Dies gelingt erst mit den YOLOv1 Netzwerken (siehe Kapitel 4.2.3) und den SSD. Letztere liefern eine etwa 74 prozentige Accuracy bei 59 FPS [15, S.2] und übertreffen damit Faster R-CNN bei weitem, sowie YOLOv1. Diese Verbesserung wird erreicht, indem unter anderem auf die "bounding box proposals"[15, S.2] verzichtet wird. Umgesetzt wird dies durch den Einsatz kleiner convolutional filter die Objektkategorien, und die dazu passenden bounding box locations, predicten. Die Filter werden auf verschiedenen Feature Maps im ganzen Verlauf des Netwerkes eingesetzt, so dass eine multipel skalierte Prediction erreicht wird [15, S.2].

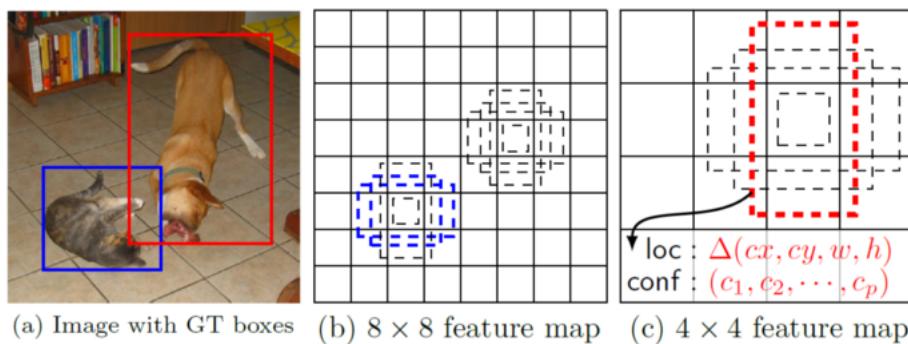


Abbildung 4.2: SSD framework, [15, S.3]

Abbildung 4.2 zeigt ein Beispiel für die Basis Funktionsweise eines SSD. Als Input wird ein Bild genommen auf dem Boxen für verschiedene Klassen (hier Hund und Katze) vermerkt

sind. In Feature Maps mit verschiedenen Skalen (hier 8x8 und 4x4) sollen die Boxen des Bildes predicted werden. Dafür wird eine kleine Anzahl (hier 4) vordefinierter bounding boxen über das Bild gelegt und davon ausgehend predicted welche Klasse sich in ihr befindet bzw. eine Anpassung der Dimensionen der Bounding Box vorgenommen [15, S.3f.].

### 4.2.3 You Only Look Once

Im Gegensatz zu herkömmlichen OD Systemen verzichtet YOLO auf langwierige Vorberechnungen, wie z.B. bei R-CNN die Region Proposals. Das zuvor angesprochene SSD arbeitet nach dem selben Prinzip, wurde jedoch erst nach der originalen YOLO Vorstellung beschrieben. Aus diesem Grund soll in diesem Kapitel zunächst auf die Basis Funktionsweise eines YOLO Netzwerks eingegangen werden, bevor sich mit dem aktuellen (und für dieses Seminar verwendete) YOLOv3 beschäftigt wird.

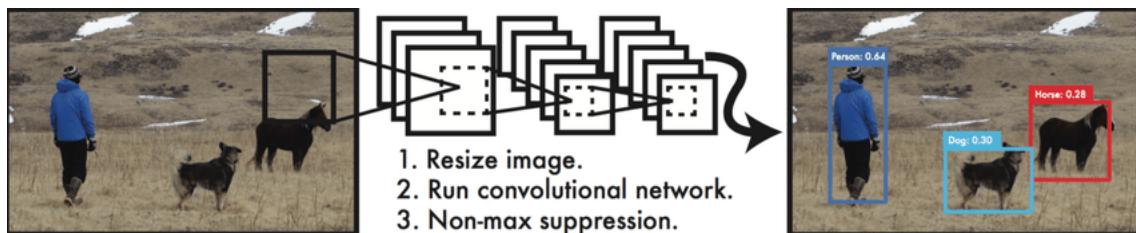
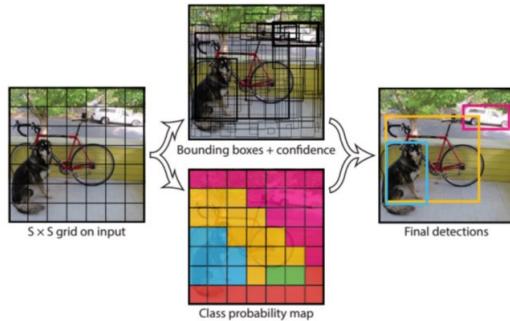


Abbildung 4.3: YOLO Detection System, [26, S.1]

Abbildung 4.3 zeigt dabei vereinfacht die Funktionsweise auf. Das Eingabebild wird auf eine vorgegebene Größe geändert, bevor in einem CNN simultan verschiedene Bounding boxes, sowie die zugehörigen Klassenwahrscheinlichkeiten erzeugt werden [26, S.1]. YOLO betrachtet dabei das Bild im Ganzen, womit auch kontextuelle Klasseninformationen in die OD miteinbezogen werden können [26, S.1f.]. Dies unterscheidet es vom R-CNN, das im Hintergrund mehr als doppelt so oft falsche Objekte erkennt als YOLO [26, S.2]. Für die Erkennung wird jedes Input Bild in ein  $S \times S$  Grid aufgeteilt (siehe Abbildung 4.4). Im nächsten Schritt wird für jede der  $S \times S$  Boxen eine Anzahl ( $B$ ) an möglichen Bounding Boxen predicted. Die Bounding Boxen enthalten dabei Werte über die x,y Position, die Weite und Höhe, sowie die Confidence mit der etwas in dieser Box liegt. Im selben Schritt werden für jede der Grids  $C$  viele mögliche Klassenwahrscheinlichkeiten berechnet. Abschließend werden die Wahrscheinlichkeiten mit der Confidence kombiniert, wodurch schließlich die OD durchgeführt wurde. Das Ergebnis sind  $n$  viele Bounding Boxen mit der jeweiligen Wahrscheinlichkeit des darin enthaltenen Objekts. Diese Grundfunktionsweise bleibt auch



**Figure 2: The Model.** Our system models detection as a regression problem. It divides the image into an  $S \times S$  grid and for each grid cell predicts  $B$  bounding boxes, confidence for those boxes, and  $C$  class probabilities. These predictions are encoded as an  $S \times S \times (B * 5 + C)$  tensor.

Abbildung 4.4: Model, [26, S.2f.]

in den nachfolgenden Versionen YOLOv2 und YOLOv3 erhalten, die Netzwerke werden jedoch besser und stärker. Vor allem YOLOv3 erzielt deutlich bessere Ergebnisse bei der Erkennung kleiner Objekte [14].

## 4.3 YOLO Implementierung

### 4.3.1 Vortrainierte Modelle

Für die Implementierung der YOLO OD wurde auf vortrainierte config und weight Dateien zurückgegriffen. Redmon et al. stellen diese auf ihrer Website [27] zur Verfügung. Für diese Arbeit wurde YOLOv3, sowie YOLOv3-tiny verwendet. In dem dieser Arbeit beigefügtem Git-Repository befindet sich die OD in der Datei /AutonomousCar/main/object-detection/object-detection.py. In der nun folgenden Erklärung werden kurze Code Abschnitte verwendet, für eine komplette Durchsicht des Codes sei auf die oben genannte Datei verwiesen. Die Implementierung basiert dabei auf Tutorials von Adakane [6, 7]. Die vortrainierten weights / cfgs basieren auf dem coco Datensatz (Z. 10). Wie bereits in Kapitel 4.2.3 dargelegt wird das Image auf eine bestimmte Größe, hier 416x416 skaliert (Z.3). Unser Code bietet die Möglichkeit (Zeile 7) zwischen den verschiedenen YOLO Verisonen zu wechseln, was aufgrund der Performance auf dem Raspberry Pi nötig ist. Auf diesen Punkt wird in Kapitel 4.3.3 eingegangen. In Zeile 13 und 14 wird das vorverarbeitete Bild dem Netzwerk übergeben. Als Rückgabe wird eine Liste der erkannten Objekte erwartet.

```

1 ...
2 # preprocess image
3 blob = cv2.dnn.blobFromImage(img_original, 1 / 255.0, (416,
4 416),
5 swapRB=True, crop=False)
6
7 # load network
8 output_layers, net = load_yolo(tiny=tiny)
9
10 # load coco list
11 class_list = load_coco_names()
12
13 # detect objects
14 net.setInput(blob)
15 outs = net.forward(output_layers)
16 ...

```

Quelltext 4.1: Yolo Modelle

Diese Ergebnisse werden dann in zwei Funktionen (information-cal und information-draw) weiterverarbeitet. Information-cal (Datei ab Zeile 68), überprüft dabei für jedes Objekt in outs, ob es mit einer zufriedenstellenden Confidence (hier 0.5) erkannt wurde. Außerdem werden die Höhe, Breite, x- und y-Position der Bounding Box berechnet und gespeichert. Diese Box wird außerdem mit dem Objekt-Namen angereichert. Daraufhin werden die Ergebnisse der information-draw Funktion übergeben, die dann mithilfe von OpenCV in das Eingabebild eingezeichnet werden. OpenCV bietet dabei die Möglichkeit das Schriftbild zu verändern.

```

1 ...
2 for i in range(len(boxes)):
3
4     if i in indexes:
5
6         x, y, w, h = boxes[i]
7         label = str(class_list[class_ids[i]])
8         full_label = label + ", " + str(round(confidences[i] *
9             100, 2))
10        color = colors[class_ids[i]]

```

```

10     cv2.rectangle(img, (x, y), (x + w, y + h), color,
11         rec_width)
12     cv2.putText(img, full_label, (x, y - 5), font,
13         txt_height,
14         color, text_width)
15     ...

```

Quelltext 4.2: Yolo Erkennung für jeweils ein Bild

Das oben gezeigte beschreibt die Erkennung für jeweils ein Bild. Dies kann auf einen Videostream (z.B. Webcam, hier Pi-Camera) umgeschrieben werden, indem jedes einzelne, von der Kamera aufgezeichneten Frame, diesen Prozess durchläuft.

### 4.3.2 Individuell trainiertes Modell

Aufgrund des Modellmaßstabes des Projekts wird das Auto auf keine Menschen treffen, die die OD auslösen können. Aus diesem Grund wurde versucht ein individuelles YOLO Netzwerk zu trainieren, dass Playmobilmenschen erkennt. Hierfür wurde auf ein Tutorial, sowie ein Google Colab Notebook von Singh zurückgegriffen [32]. Die verwendete Dateistruktur ist unter „<https://1drv.ms/u/s!Alb7pfMD-8hrg50yVtZo-2HU0mjyQQ?e=jRTmqu>“ verfügbar. Im Folgenden sollen die dafür verwendeten Schritte beschrieben werden.

1. Datenbeschaffung: Als Datenquelle wurde zunächst die PlaymoDB [21] nach passenden Bildern durchsucht. Abbildung 4.5 zeigt ein Beispielbild. Hieran wird auch ein mögliches Hemmniss deutlich. Der überwiegende Teil der Bilder ist aus diesem Winkel und dieser Position aufgenommen worden, sodass fraglich bleibt wie erfolgreich anhand der Daten auf neue Bilder aus anderen Winkeln generalisiert werden kann.



Abbildung 4.5: PlaymoDB Bild, Playmobil aus Google Bildersuche  
Quelle: <https://playmodb.org/cgi-bin/showpart.pl?partnum=302000200404>

Alternativ dazu wurden etwa 100 Bilder aus der Google Bildersuche extrahiert. Für das Auffinden wurden die Suchebegriffe "Playmobil Figuren" bzw. "Playmobil Familie" gewählt. Dieser Schritt war notwendig, da mit den von der PlaymoDB bereitgestellten Daten kein zufriedenstellendes YOLO Netzwerk trainiert werden konnte. Mithilfe der neuen Bilder ist es gelungen das Netzwerk Playmobilfiguren erkennen zu lassen. Die verwendeten Bilder sind hier [https://1drv.ms/u/s!Alb7pfMD-8hrg50inCK0\\_dDLyEZ9OA?e=CZDXND](https://1drv.ms/u/s!Alb7pfMD-8hrg50inCK0_dDLyEZ9OA?e=CZDXND) zu finden.

2. Labeling: Ein zentraler (und zeitaufwendiger) Punkt des Trainings ist das Labeling der zuvor beschafften Bilder. Dieses geschieht in unserem Fall über das pip labeling Tool labellImg (siehe Abbildung 4.6). Hierbei werden die Objekte (Playmobilfiguren) mit einer passenden Bounding Box versehen. Diese Informationen werden in einer txt Datei gespeichert im Format: <ClassID> <x-Zentrum> <y-Zentrum> <Breite> <Höhe>.

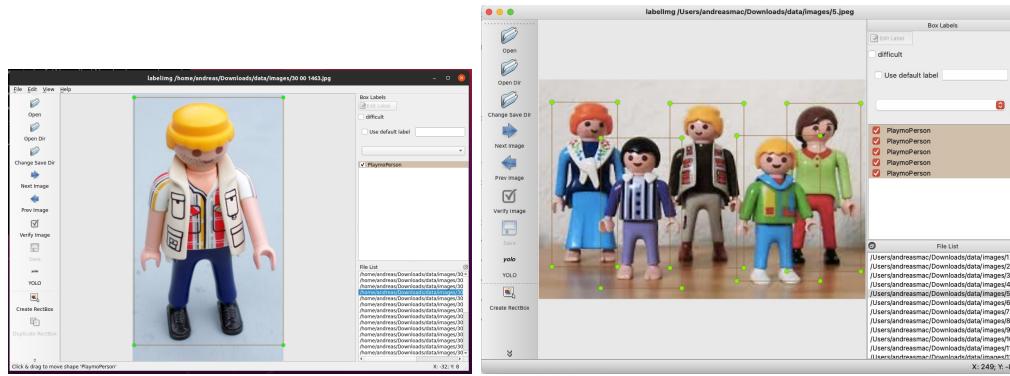


Abbildung 4.6: LabellImg Anwendung (links PlaymoDB, rechts aus Google Bildersuche)

3. Training: Für das Training wird auf ein Tutorial, sowie ein bereitgestelltes Google Colab Notebook zurückgegriffen [24]. Erwähnt seien die nötigen Anpassungen in den cfg Dateien (ob YOLO oder tiny YOLO). Hierfür müssen in den YOLO Layern, sowie dem jeweils vorgelagerten Conv Layer Veränderungen vorgenommen werden.

```

1 ...
2 [convolutional]
3 size=1
4 stride=1
5 pad=1
6 filters=18
7 activation=linear

```

```

8
9     [yolo]
10    mask = 3,4,5
11    anchors= 5.80289956, 11.72601733, 7.61052314, 11.16109429, 11.2596055,
12    classes = 1
13    num = 6
14    jitter=.3
15    ignore_thresh = .7
16    truth_thresh = 1
17    random=1
18    ...

```

Diese betreffen die filters in Zeile 6, diese müssen entsprechend der Formel,

$$filters = (Klassenanzahl + 5) * num/2 \quad (4.1)$$

also:  $filters = (1 + 5) * 6/2 = 6 * 3 = 18$ , gesetzt werden. Außerdem muss in Zeile 12 die Anzahl der Klassen angepasst werden. Nach diesen Anpassungen kann das Colab Notebook ausgeführt werden, welches nach einer definierten Anzahl an Trainingsschritten die Gewichte ausgibt.

### 4.3.3 Performance

Im Generellen kann festgehalten werden, dass ein normales YOLOv3 Netzwerk deutlich bessere Ergebnisse, als ein tinyYOLOv3 Netzwerk liefert. Dies zeigt sich bei einem beispielhaften Durchlauf in Abbildung 4.7. Das tiny Netzwerk erkennt zwar auch Personen im Bild, dies aber nur sehr spärlich im Vergleich zum vollen Netzwerk. Optimalerweise sollte also auf das tiny Netzwerk verzichtet werden.

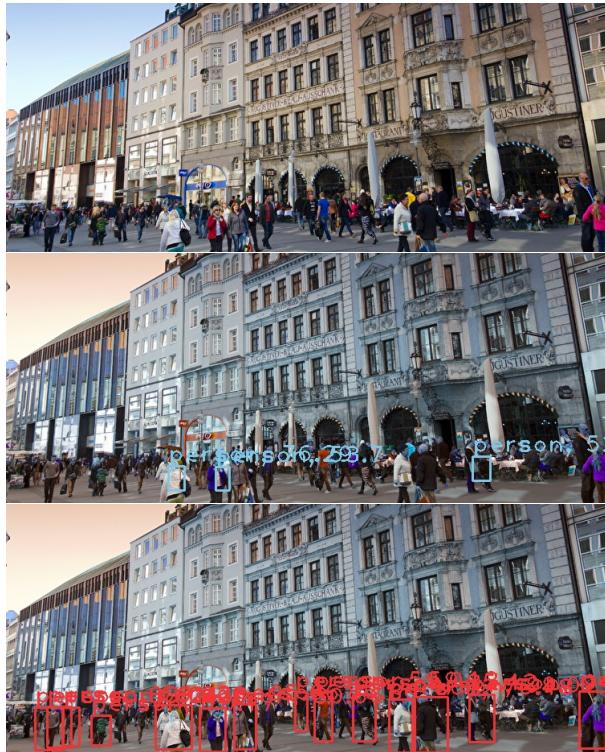


Abbildung 4.7: oben: Original, mitte: tinyYOLO, unten: YOLO

Quelle: <https://images.portal.muenchen.de/000/000/221/848/versions/neuhauser-strasse-detail.jpg>

An diesem Punkt treten jedoch Hardwarelimitationen zu Tage. Während Redmon auf eine Titan X (Nvidia Titan X) GPU [25, S.1] zurückgreifen konnte, und damit Echtzeit Erkennung mittels eines vollen Netzwerks durchführen konnte, wird für dieses Projekt lokal auf einem MacBook Air (2013), sowie im Produktivbetrieb auf einem Raspberry Pi 3 (siehe Sektion 5.1) gearbeitet. Ein normales YOLO Netzwerk läuft auf dieser Hardware nicht performant. Aus diesen Gründen wurde ein tinyYOLO, im speziellen ein custom tinyYOLO, verwendet. Dieses wurde wie in Kapitel 4.3.2 beschrieben trainiert. Mit diesem Netzwerk konnten gute Ergebnisse erzielt werden, wie mittels eines Beispiel Youtube Videos deutlich wurde. (Auffindbar im Github Repository unter [doku/cust-yolo.mp4](#))

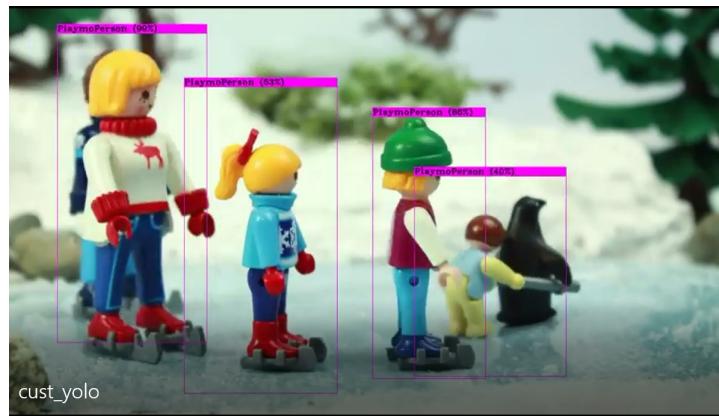


Abbildung 4.8: Test Youtube Video  
Quelle: <https://www.youtube.com/watch?v=vbc9spoCrMlt=199s>

Im Praxisbetrieb wurde jedoch deutlich, dass die Figuren zu spät erkannt werden, sodass ein Bremsen des Autos nicht mehr möglich ist.

## 4.4 Distanzberechnung

Mithilfe von OpenCV ist es möglich den Abstand zu einem erkannten Objekt in einem Bild relativ zur Kamera zu messen. Hierfür wird auf ein Tutorial von Rosebrock zurückgegriffen [1]. Die Berechnungen basieren auf der Dreiecks Ähnlichkeit (triangle similarity). Diese besagt in unserem Fall im Groben, dass wir ausgehend von einem Objekt in einem Foto, dessen Weite und Abstand bekannt sind, die Brennweite (focal length) berechnet werden kann. Von dieser ausgehend kann für ein neues Objekt, dessen Pixelweite bekannt ist, der Abstand zur Kamera berechnet werden. Das neue Objekt bzw. die neue Breite sind im Fall dieses Projektes die Weite der Bounding Box eines bestimmten Objekts. Für dieses Modell wurde als Kalibrierungsmaßstab eine Soundbox mit 16cm Breite und einem 50cm Abstand zur Kamera gewählt. Mithilfe des in Kapitel 2.2 beschriebenen Grayscaleings, sowie einer OpenCV Funktion, die es erlaubt Konturen zu finden und zu extrahieren (`cv2.findContours()`) kann die Pixelweite der Soundbox extrahiert werden.



Abbildung 4.9: Marker (Original links, Kontur rechts)

Die focal length errechnet sich durch:

$$focalLength = (markerPixelWidth * markerDistance) / markerWidth \quad (4.2)$$

Hiervon ausgehend kann die Distanz zu Objekten errechnet werden:

$$distance = (markerWidth * focalLength) / objectWidth \quad (4.3)$$

Zu beachten ist, dass die focal Length Kamera-, sowie Winkelspezifisch ist. Es ist somit auf ein aktuelles Kalibrierungsfoto zu achten.

In unserem Projekt soll mithilfe der Distanzmessung ein „Call to Action“ erreicht werden. In der zugehörigen Funktion check-reation (Z. 106) wird in einer Liste festgelegt bei welchen Klassen die Distanz gemessen werden soll, z.B. Person. Sobald eine Person erkannt wird, wird für diese auch laufend die Distanz errechnet. Im Folgenden kann dann an das Auto ausgegeben werden zu stoppen, falls sich eine Person unterhalb eines bestimmten Abstands zur Kamera (und damit zum Auto) befindet.

# 5 Praktische Umsetzung

## 5.1 Hardware

Für die Umsetzung des Projekts wird ein ferngesteuertes Modellauto und ein Raspberry Pi 3B+ mit Kameramodul verwendet. Auf die genaue Verwendung der genannten Komponenten wird im Weiteren eingegangen.

### 5.1.1 Raspberry Pi

Ein Pi ist ein Minicomputer mit Betriebssystem in Kreditkartengröße. Er verfügt unter anderem über USB-Anschlüsse, einen HDMI-Port, sowie einen Anschluss für ein Kameramodul. Des Weiteren verfügt der Pi über GPIO-Ports die über Programmiersprachen, wie zum Beispiel in diesem Project mit Python und den entsprechenden Bibliotheken, angesteuert werden können. [30]

Für dieses Projekt wird Raspberry Pi OS als Betriebssystem auf einer 8 Gb microSD und eine 10.000 mAh Powerbank zum Betreiben des Pi's genutzt.

Der Pi bildet das Herzstück des SELMA Projekts. Mit Hilfe des Kameramoduls werden Bilder aufgenommen, die durch die Softwarekomponenten Lanedetection und Objectdetection ausgewertet werden, um mit deren Ausgaben das Modellauto zu steuern.

Die Programmierung findet hauptsächlich außerhalb des Pi's statt. Um den Code auf den Pi zu spielen, wird dort das Github Repository des Projekts gecloned und fortlaufend aktualisiert. Für kleinere Anpassungen am Code während Tests mit dem Modellauto, kommt eine USB Maus, Tastatur und ein externer Monitor zum Einsatz.

### 5.1.2 Modellauto

Das Modellauto besteht aus den Komponenten: Lenkservo, Funkfernsteuerung, Empfänger, Motor, Fahrtenregler und Akku. Die Steuerung funktioniert normalerweise über die Funkfernsteuerung, in dem sie Steuersignale an den Empfänger sendet. Dieser wird über

die Verbindung des Fahrtenreglers mit Strom versorgt und versorgt seinerseits den Lenkservo mit Strom. Die Verbindung zu Lenkservo und Fahrtenregler besteht aus jeweils drei Adern. Zwei dienen der Stromversorgung, wie bereits beschrieben, die dritte Ader dient der Ansteuerung der beiden Komponenten. Über diese Adern werden PWM Signale in einer bestimmten Frequenz übermittelt. PWM ist eine digitale Modulationsart, bei der in einer gegebenen Frequenz ein Intervall in bestimmter Dauer übermittelt wird. Im Folgenden ist ein solches Signal visuell dargestellt:

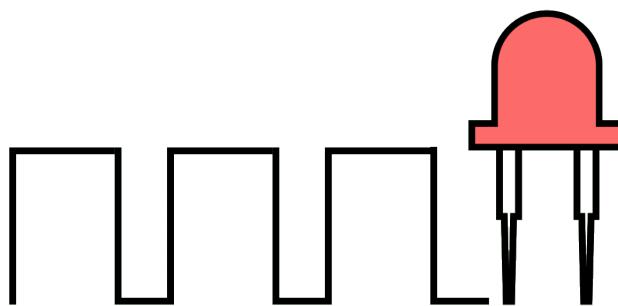


Abbildung 5.1: PWM-Signal

Quelle: <https://www.electronicwings.com/raspberry-pi/raspberry-pi-pwm-generation-using-python-and-c>

Über diese Signale werden die empfangenen Anweisungen der Fernsteuerung an den Lenkservo und den Fahrtenregler weitergegeben. Abhängig von der Intervalldauer wird beschleunigt, beziehungsweise gelenkt.

### 5.1.3 Verkabelung

Um das Modellauto mit dem Pi zu steuern, muss die Übermittlung der PWM Signale vom Pi geschehen. Um dies zu realisieren, wird der Pi zwischen den Empfänger und den Lenkservo beziehungsweise dem Fahrtenregler eingelötet. Um die PWM Signale vom Pi zu übermitteln, muss jeweils für Lenkservo und Fahrtenregler, ein Minuskabel und die dritte Ader, für die Signalübertragung, mit dem Pi anstatt dem Empfänger verbunden sein. Im Folgenden sind die Verbindungen zwischen allen Komponenten visuell dargestellt:

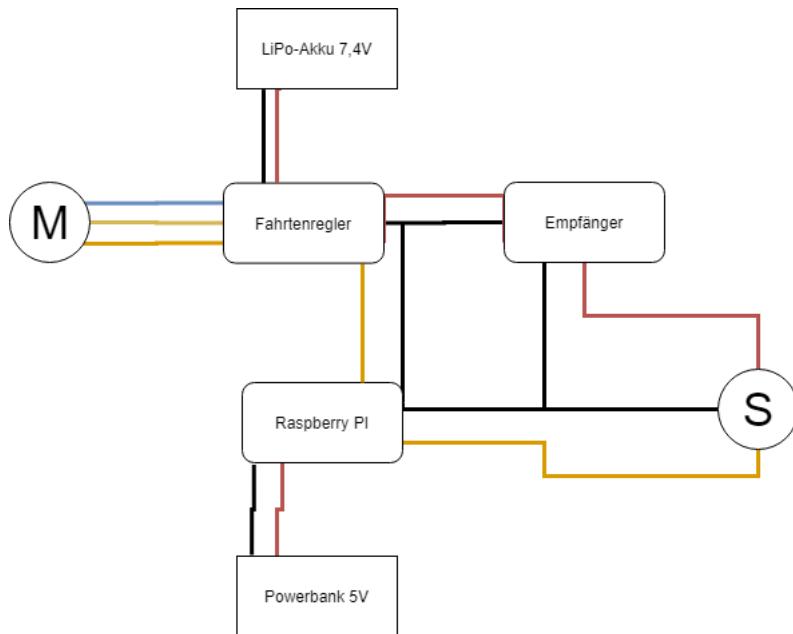


Abbildung 5.2: Graphische Darstellung der Verkabelung

## 5.2 Steuerung des Modellautos

Die Steuerung des Modellautos besteht aus zwei Bestandteilen. Zum Ersten wird die Möglichkeit benötigt, das Modellauto vom Pi anzusteuern. Dies wird im folgenden Abschnitt „Ansteuerung der GPIO-Pins mit Python“ erläutert. Zum Zweiten wird ein Skript benötigt, das die Steuerbefehle anhand der Kamerabilder generiert und mit diesen das Modellauto steuert. Die Erläuterung dieses Vorgehens ist im Abschnitt „Modellautosteuerung mittels Object- und Lanedetection“ zu finden.

### 5.2.1 Ansteuerung der GPIO-Pins mit Python

Für die Umsetzung der Ansteuerung wird die RPi.GPIO Bibliothek und die Pins 33 und 35 für Fahrtenregler und Lenkservo genutzt. Die PWM Signale werden in einer 50 Hertz Frequenz übermittelt. Hierbei muss besonderes Augenmerk auf die korrekte Frequenz gelegt werden, da sonst die Elektronik des Lenkservos beziehungsweise des Fahrtenreglers Schaden nehmen kann. Dies ist bei der Umsetzung des Projektes leider zweimal mit Lenkservos, in Folge einer falschen Frequenzwahl geschehen. Im Weiteren werden die Kernbestandteile der Ansteuerung des Modellautos mit PWM-Signalen erklärt. Die volle Ansteuerung des

Modellautos ist in der Car-Klasse implementiert, welche im Git-Repository unter /AutonomousCar/main/car\_controll/controll\_car.py zu finden ist.

```

1 IO.setmode(IO.BOARD)
2 IO.setup(35,IO.OUT)
3 steering =IO.PWM(35,50)
4 steering.start(0)
```

Quelltext 5.1: Initialisierung der GPIO

Mit den obigen Befehlen wird die Ansteuerung des PWM-Pin 35 mit einer 50 Hertz-Frequenz für die Ansteuerung des Lenkservos initiiert. Die Intervalldauer des Aktivenintervalls (duty cycle) wird dabei zu Beginn auf 0 gesetzt.

```

1 steering.ChangeDutyCycle(10)
2 steering.stop()
```

Quelltext 5.2: Anpassung des PWM-Signal

Mit der Funktion ChangeDutyCycle kann die Dauer des Aktivenintervalls (duty cycle) verändert werden. In diesem Beispiel wird die Zeit des Aktivenintervalls auf 10% der Intervalldauer gesetzt. Dies würde für das Modellauto einen Lenkbefehl nach links entsprechen. Mit der Funktion stop wird die Ansteuerung des entsprechenden GPIO-Pins gestoppt. Mit Hilfe dieser Funktionen wird die Ansteuerung des Modellautos in der bereits erwähnten Car-Klasse umgesetzt.

## 5.2.2 Modellautosteuerung mittels Object- und Lanedetection

Für die Steuerung des Modellautos wird das run\_car.py Skript genutzt. Dieses nutzt die Kamera um ständig Bilder aufzunehmen und diese zur Steuerung auszuwerten. Dazu werden Funktionen aus der Object- und Lanedetection genutzt. Darüber hinaus wird ein Car-Objekt erzeugt, mit dem das Modellauto angesteuert werden kann, wie dies in „Ansteuerung der GPIO-Pins mit Python“ erläutert wurde.

Die Auswertung der Kamerabilder wird in einer While-Schleife durchgeführt. Dazu wird zu Beginn der While-Schleife ein Bild mit der Kamera erstellt. Dieses wird, falls kein Thread zur Objekterkennung läuft, an einen neuen Thread zur Objekterkennung übergeben. Dieser läuft asynchron und blockiert nicht den weiteren Ablauf und die Auswertung zur Steuerung des Autos. Nach dem, falls notwendig, der Objekterkennungsthread gestartet wurde,

wird das Bild mit Hilfe der lane\_detection Funktion aus dem lanedetect\_steer Skript ausgewertet. Die Funktion gibt Anweisungen zur Steuerung des Modellautos zurück. Diese Anweisungen werden genutzt um mit der Steer-Methode der Car-Klasse das Modellauto zu steuern. Da der von uns genutzte Pi zu starke Schwankungen in den PWM-Signalen hat, wurde von einer Umsetzung der Fahrtenregleransteuerung abgesehen. Durch die Schwankungen, ist es nicht möglich eine konstante Geschwindigkeit zu halten, da das Modellauto immer wieder stark beschleunigt. Für die Tests im Außen- und Innenbereich wären hierdurch Schäden nicht auszuschließen. Aus diesem Grund, ist der Output des Objekterkennungs-thread lediglich, ein auf die Konsole ausgegebenes „STOP!“, falls eine Playmobil Figur erkannt wird. Wäre diese Problematik nicht gegeben, würde das Modellauto mit konstanter Geschwindigkeit fahren und mit den Methoden run beziehungsweise stop der Car-Klasse in Abhängigkeit des Objekterkennungsthread-Outputs gesteuert werden.

# 6 Fazit

## 6.1 Spurerkennung

Zusammenfassend ist festzuhalten, dass eine Spurerkennung anhand bildmanipulatorischer Mittel funktioniert und darüber hinaus mit den richtigen Parametereinstellungen sehr Performant ist. Messungen haben ergeben, dass dieses System eine hohe Bearbeitungsgeschwindigkeit hat. Mit der vorliegenden Implementierung dauert die Verarbeitung pro Frame lediglich 0.06 Sekunden. Damit ist eine flüssige Fahrt des Fahrzeugs SELMA möglich.

Die Beschaffenheit der Fahrbahn spielt dabei eine ungeahnt wichtige Rolle. Im Projekt wurde die Lane Detection zuerst in der Wohnung zweier Team Mitglieder getestet. Dafür wurden Fahrspuren mit weisem Klebeband auf dem Boden markiert. Die Spurerkennung hat dabei gute Ergebnisse geliefert, da der Kontrast zwischen Boden und Markierung hoch war. Als dann aber zum ersten mal auf einem öffentlichen Parkplatz getestet wurde, war festzustellen, dass weise Körner im Asphalt bei so geringer Distanz zur Fahrbahn große Schwierigkeiten bergen. Deshalb wurde die Entscheidung getroffen, zwischen zwei Terrains zu unterscheiden und die Spur Erkennung jeweils zu optimieren. Hierfür musste die Datenvorverarbeitung angepasst werden. Das Problem konnte mit einer zusätzlichen Funktion zur Manipulation der Helligkeit und des Kontrastes gelöst werden.

Im Allgemeinen wurden die Anforderungen an die Performance des Projekts SELMA mit der gegebenen Lösung des Problems übertroffen. Die Leistung des Fahrzeugs auf dem Parcours ist, bis auf sehr wenige Fehler, überragend gut.

## 6.2 Objekterkennung

Hinsichtlich der Object Detection ist zu berichten, dass eine Umsetzung mithilfe eines YOLO Netzwerkes möglich ist. Hierbei ist darauf zu verweisen, dass die Verwendung eines vollen YOLO Netzwerkes zu empfehlen wäre, die Hardwareleistung eines Raspberry Pi 3 jedoch nur für ein tiny YOLO Netzwerk ausreichend ist. Im Laufe dieses Projektes wurde ebenso deutlich, dass für ein individuelles Training der YOLO Architektur ein Bildkorpus in ausreichendem Umfang, sowie einer annehmbaren Qualität bereitgestellt werden muss.

Dies schließt also die Verwendung der zunächst präferierten PlaymoDB aus, während das Labeling aufgrund mangelnder Datenquellen per Hand stattfinden muss. Es ist darauf hinzuweisen, dass diese Erkennung bei Playmobil Menschen funktioniert, wenn diese einen relativ geringen Abstand zur Kamera haben, was aufgrund des Winkels der Raspberry Kamera, sowie der Geschwindigkeit in der das Auto sich bewegt nicht umzusetzen war. Verbesserungen können hier erfolgen, indem weitere Datensätze generiert werden, in denen die Playmobil Personen im Hintergrund stehen. Im Zuge der OD wurde deutlich, dass mithilfe der Focal Distance eine Entfernung zu erkannten Objekten berechnet werden kann.

# Literaturverzeichnis

- [1] Adrian Rosebrock. *Find distance from camera to object / marker using Python and OpenCV*. 2015. URL: <https://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/>.
- [2] Abdulelah Alkesaiberi. *Best value for threshold in Canny*. Juli 2020. URL: <https://stackoverflow.com/questions/25125670/best-value-for-threshold-in-canny>.
- [3] AutoPilot Review. *Elon Musk on Cameras vs. LiDAR for Self Driving and Autonomous Cars*. 2019. URL: <https://www.youtube.com/watch?v=HM23sjtk4Q&feature=youtu.be>.
- [4] Soumya Ranjan Behera. *Lane Line Detection*. Sep. 2019. URL: <https://www.kaggle.com/soumya044/lane-line-detection/notebook>.
- [5] J. Canny. „A Computational Approach to Edge Detection“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), S. 679–698. DOI: 10.1109/TPAMI.1986.4767851.
- [6] Darshan Adakane. *Object Detection with OpenCV Python using YOLOv3*. 2019. URL: <https://medium.com/analytics-vidhya/object-detection-with-opencv-python-using-yolov3-481f02c6aa35>.
- [7] Darshan Adakane. *Real Time Object Detection using YOLOv3 with OpenCV and Python*. 2019. URL: <https://medium.com/analytics-vidhya/real-time-object-detection-using-yolov3-with-opencv-and-python-64c985e14786>.
- [8] Dr.-Ing. Thomas Tille. *Automobil-Sensorik 2: Systeme, Technologien und Applikationen*. Springer Verlag, 2018. ISBN: 978-3-662-56309-0. DOI: 10.1007/978-3-662-56310-6. URL: [https://link.springer.com/chapter/10.1007/978-3-662-56310-6\\_2](https://link.springer.com/chapter/10.1007/978-3-662-56310-6_2).
- [9] *Drawing Functions*. Jan. 2021. URL: [https://docs.opencv.org/master/d6/d6e/group\\_\\_imgproc\\_\\_draw.html#ga8c69b68fab5f25e2223b6496aa60dad5](https://docs.opencv.org/master/d6/d6e/group__imgproc__draw.html#ga8c69b68fab5f25e2223b6496aa60dad5).
- [10] Robert Fisher et al. *Image Transforms - Hough Transform*. 2000. URL: <http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm>.
- [11] Ross Girshick. *Fast R-CNN*. URL: <https://arxiv.org/pdf/1504.08083.pdf>.

- [12] Ross Girshick et al. „Region-Based Convolutional Networks for Accurate Object Detection and Segmentation“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.1 (2016), S. 142–158. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2015.2437384.
- [13] Yangqing Jia et al. *Caffe: Convolutional Architecture for Fast Feature Embedding*. URL: <https://arxiv.org/pdf/1408.5093>.
- [14] Ayoosh Kathuria. „What’s new in YOLO v3? - Towards Data Science“. In: *Towards Data Science* (23.04.2018). URL: <https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b>.
- [15] Wei Liu et al. *SSD: Single Shot MultiBox Detector*. 2016. DOI: 10.1007/978-3-319-46448-0{\textunderscore}2. URL: <https://arxiv.org/pdf/1512.02325>.
- [16] M. Manoj krishna et al. „Image classification using Deep learning“. In: *International Journal of Engineering & Technology* 7.2.7 (2018), S. 614. ISSN: 2227-524X. DOI: 10.14419/ijet.v7i2.7.10892.
- [17] Wojciech Mokrzycki und Samko M. „New version of Canny edge detection algorithm“. In: Jan. 2012, S. 533–540.
- [18] *OpenCV Documentation*. 2020. URL: <https://opencv.org/about/>.
- [19] OpenCV Documentation. *OpenCV: Smoothing Images: Image Processing in OpenCV*. 3.01.2021. URL: [https://docs.opencv.org/master/d4/d13/tutorial\\_py\\_filtering.html](https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html).
- [20] *OpenCV: Hough Line Transform*. 2021. URL: [https://docs.opencv.org/3.4/d9/db0/tutorial\\_hough\\_lines.html](https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html).
- [21] PlaymoDB. *PlaymoDB*. 2020. URL: <https://playmodb.org/cgi-bin/category.pl?category=Klicky-Male%20child>.
- [22] Lutz Priese. „Einleitung“. In: *Computer Vision: Einführung in die Verarbeitung und Analyse digitaler Bilder*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, S. 1–9. ISBN: 978-3-662-45129-8. DOI: 10.1007/978-3-662-45129-8\_1. URL: [https://doi.org/10.1007/978-3-662-45129-8\\_1](https://doi.org/10.1007/978-3-662-45129-8_1).
- [23] Qin Zou, Hanwen Jiang, Qiyu Dai, Yuanhao Yue, Long Chen, and Qian Wang. *Robust Lane Detection from Continuous Driving Scenes Using Deep Neural Networks*. Apr. 2020. URL: <https://arxiv.org/pdf/1903.02193.pdf>.

- [24] Rafi. *Train your own tiny YOLO v3 on Google colaboratory with the custom dataset*. 2019. URL: <https://medium.com/@today.rafi/train-your-own-tiny-yolo-v3-on-google-colaboratory-with-the-custom-dataset-2e35db02bf8f>.
- [25] Joseph Redmon und Ali Farhadi. *YOLOv3: An Incremental Improvement*. URL: <https://arxiv.org/pdf/1804.02767.pdf>.
- [26] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. URL: <https://arxiv.org/pdf/1506.02640.pdf>.
- [27] Redmon, Joseph, Fahradi, Ali. *YOLO*. 2020. URL: <https://pjreddie.com/darknet/yolo/>.
- [28] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. URL: <https://arxiv.org/pdf/1506.01497.pdf>.
- [29] Technology Robotix Society. „Grayscale, Binary, and Histogram - COMPUTER VISION & ROBOTICS - Medium“. In: *COMPUTER VISION & ROBOTICS* (2.07.2019). URL: <https://medium.com/image-processing-in-robotics/grayscale-binary-and-histogram-2548ffcee6c5>.
- [30] Michal Šustek et al. „DC motors and servo-motors controlled by Raspberry Pi 2B“. In: *MATEC Web of Conferences* 125 (Jan. 2017), S. 02025. DOI: 10.1051/matecconf/201712502025.
- [31] J. R. R. Uijlings et al. „Selective Search for Object Recognition“. In: *International Journal of Computer Vision* 104.2 (2013), S. 154–171. ISSN: 1573-1405. DOI: 10.1007/s11263-013-0620-5.
- [32] Vijay Singh. *Train your own custom model for Helmet Detection (object detection) using YOLO*. 2018. URL: [https://medium.com/@vijaysingh\\_60587/train-your-own-custom-model-for-helmet-detection-object-detection-using-yolo-f53a48066d7a](https://medium.com/@vijaysingh_60587/train-your-own-custom-model-for-helmet-detection-object-detection-using-yolo-f53a48066d7a).
- [33] Yan Wang et al. *Pseudo-LiDAR from Visual Depth Estimation: Bridging the Gap in 3D Object Detection for Autonomous Driving*. URL: <https://arxiv.org/pdf/1812.07179.pdf>.