



Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Kollisionsvermeidung eines fahrbaren Roboters

-

Implementierung und Training eines neuronalen Netzes mittels Transfer-Learning

Name: Sebastian Richter
Aaron Zielstorff

Matrikelnummer:
572906
567183

Fachbereich: FB1
Studiengang: M. Elektrotechnik
Fachsemester: 2. FS
Fach: MSS5 Computational Intelligence
Dozent: Prof. Dr.-Ing. Steffen Borchers
Abgabe am: 23. September 2022

Inhaltsverzeichnis

1 Einleitung	4
2 Theoretische Grundlagen	5
2.1 Convolutional Neural Networks (CNN)	5
2.1.1 Convolution	6
2.1.2 Pooling	8
2.1.3 Rectification	9
2.1.4 Fully-Connected	9
2.2 AlexNet	10
2.3 Transfer-Learning	12
3 Versuchsaufbau und Inbetriebnahme	13
3.1 Benötigte Komponenten	13
3.2 Hardware-Setup	15
3.3 Software-Setup	16
4 Umsetzung der Kollisionsvermeidung	19
4.1 Ansteuern des Jetbot Roboters	19
4.2 Aufnehmen von Trainingsdaten	20
4.3 Pre-Training des CNN AlexNet	21
4.4 Live-Demo des trainierten CNN	22
5 Ergebnisse und Fazit	24
6 Ausblick - Autonomes Fahren	25
7 Anhang - Jupyter Notebooks	26
Literaturverzeichnis	55
Bücher	55
Artikel	55
Online Quellen	55

Abbildungsverzeichnis

1.1	Illustration Jetbot Roboterfahrzeug	4
2.1	Basisstruktur eines CNN	6
2.2	Unterschiedliche Eingangsbilder mit ähnlichen Merkmalen	6
2.3	Anwendung der Faltung mithilfe eines Filters	7
2.4	Filtertypen und Feature-Maps	8
2.5	Anwendung der Zusammenlegung (max. Pooling)	8
2.6	Rectified Linear Unit	9
2.7	Anwendung der Gleichrichtung (Rectification)	9
2.8	Gewichtungen im Fully-Connected-Layer	10
2.9	Schematischer Aufbau des AlexNet	11
2.10	Visualisierung des Transfer-Learning	12
3.1	Jetson Nano und Weitwinkelkamera	13
3.2	Komponenten Jetbot	14
3.3	Aufbau Jetbot	15
3.4	Jetbot Bedienoberfläche	17
3.5	Jupyter Notebook	18
4.1	Widget-Steuerung	19
4.2	Safety Bubble des Jetbot	21
4.3	Live Demo Jetbot	23
5.1	Vorteile des Transfer Learning	24
6.1	Folgen von Straßenmarkierungen	25

Tabellenverzeichnis

3.1	Materialliste	14
-----	-------------------------	----

1 Einleitung

Diese Arbeit beschäftigt sich mit der Thematik „**Künstliche Intelligenz**“. Künstliche Intelligenzen sind Forschungsgegenstand der Neuroinformatik. (vgl. [6, Seite 3]). Speziell der Zweig der **künstlichen neuronalen Netze** (engl. Artificial Neural Network ANN) spielt immer mehr eine große Rolle für reale Anwendungsfälle. Prominente Beispiele sind unter anderem das autonome Fahren und die Worterkennung bei Sprachassistenten. (vgl. [4, Seite 15]).

Künstliche neuronale Netzwerke (im Folgenden abgekürzt mit KNN) sind Netze aus künstlichen Neuronen. Dabei handelt es sich um ein Modell nach dem biologischen Vorbild einer Nervenzelle. (vgl. [4, Seiten 136, 137]). Eine spezielle Form der neuronalen Netze sind die **Convolutional Neural Networks** (im Folgenden abgekürzt mit CNN). Der Name resultiert aus der Anwendung der Faltung bei der Nutzung dieses Netzwerktyps. Es handelt sich auch hier um ein von biologischen Prozessen inspiriertes Konzept im Bereich des **maschinellen Lernens**. (vgl. [3, Seite 50]).

In dieser Arbeit soll zunächst theoretisch das Konzept und die Funktionsweise der CNN beleuchtet werden. Dabei wird vor allem auch auf **AlexNet** als beispielhaftes CNN eingegangen, da dieses im späteren Verlauf Anwendung findet. Darauf aufbauend wird im Anschluss das **Transfer-Learning** erklärt. Auch dieses soll verwendet werden.

Abschließend wird die Arbeit mit der Anwendung der behandelten Konzepte an einem praktischen Versuch getestet. Hierzu sollen in Echtzeit Bilddaten von einer Kamera auf einem kleinen Roboterfahrzeug mittels CNN und Transfer-Learning ausgewertet werden, so dass das Gefährt sich kollisionfrei im Raum fortbewegen kann. Sämtliche Berechnungen als auch die Steuerung des Roboters werden auf einem **Jetson Nano** Mikrocontroller des Unternehmens Nvidia durchgeführt. Dieses ist bekannt für die Produktion von (Hochleistungs-) Grafikkarten (kurz GPU - Graphical Processing Unit). Der Jetson Controller ist ebenfalls mit einer GPU ausgestattet, die es ermöglicht rechenintensive Operationen, wie z. B. die Berechnung von neuronalen Netzen durchzuführen.

Die Abbildung 1.1 zeigt den Jetson Nano Controller auf dem Roboterfahrzeug „*Jetbot*“ inklusive einer kleinen Weitwinkelkamera.



Abb. 1.1: Illustration des Jetbot Roboterfahrzeugs

2 Theoretische Grundlagen

Im ersten Schritt werden die einzelnen Layer eines CNNs mittels Beispielen erläutert. Anschließend erfolgt die Aufbereitung des AlexNets, welches diese Schichten anwendet und als Basis für die Bildverarbeitung zur Kollisionsvermeidung dienen wird. Im letzten Schritt wird auf das Transfer-Learning, sowie dessen Vor- und Nachteile, aber auch dessen Nutzen für die Arbeit eingegangen.

2.1 Convolutional Neural Networks (CNN)

Ein **CNN** ist ein **künstliches neuronales Netz**, welches aus mehreren Schichten (Layern) besteht und Faltungseigenschaften anwendet [15]. Die verschiedenen Layer werden in vier Kategorien eingeteilt, welche in den nachfolgenden Kapiteln erläutert werden.

- Convolution (Faltung)
- Pooling (Zusammenlegung)
- Rectification (Gleichrichtung).
- Fully-Connected (Vollständig verbunden)

Zur schematischen Übersicht des Aufbaus dient Abbildung 2.1. Da der Rectification-Layer nicht immer Anwendung findet, ist dieser nicht in der Abbildung enthalten.

Das CNN wird durch große Datenmengen, deren Eigenschaften und Klassen bekannt sind, vortrainiert. Das Eingangsmedium wird durch unterschiedliche **Filtertypen** auf bestimmte Merkmale untersucht. Die Ergebnisse werden durch **mathematische Operationen** gewichtet und in s.g. **Feature-Maps** gespeichert [15]. Anschließend erfolgt in der **Klassifizierung**, anhand der Gewichtung, die Zuordnung zu einer vorgegebenen Klasse. Die Zuordnung wird anschließend auf Richtigkeit geprüft, um Korrekturen vorzunehmen. Die Eingangsbilder werden mit kleinen Änderungen (Abbildung 2.2) eingegeben, um weitere Feature-Maps zu erzeugen. Dies erfolgt in Form von Feedbackschleifen (Epochen) [15]. Dieser Prozess wird als „**Trainieren des neuronalen Netzes**“ bezeichnet. Wird das trainierte Netz auf unbekannte, jedoch ähnliche Eingangsmedien angewendet, werden die Filter der Merkmale nicht weiter angepasst. Dieser Vorgang heißt „**Anwendung des neuronalen Netzes**“ [15].

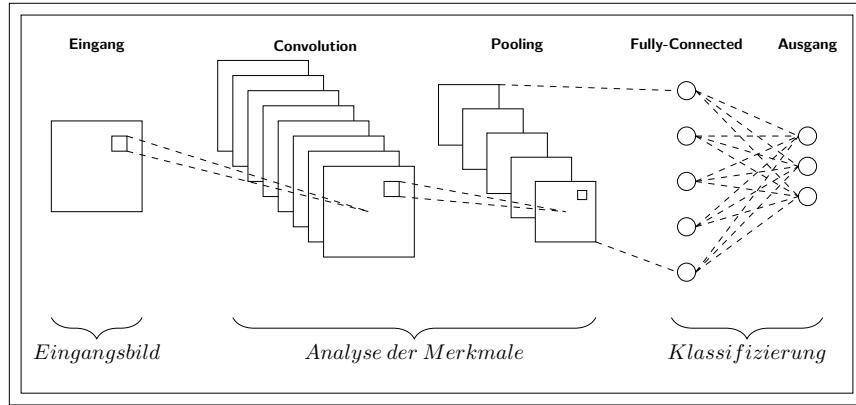


Abb. 2.1: Basisstruktur eines CNN [5]

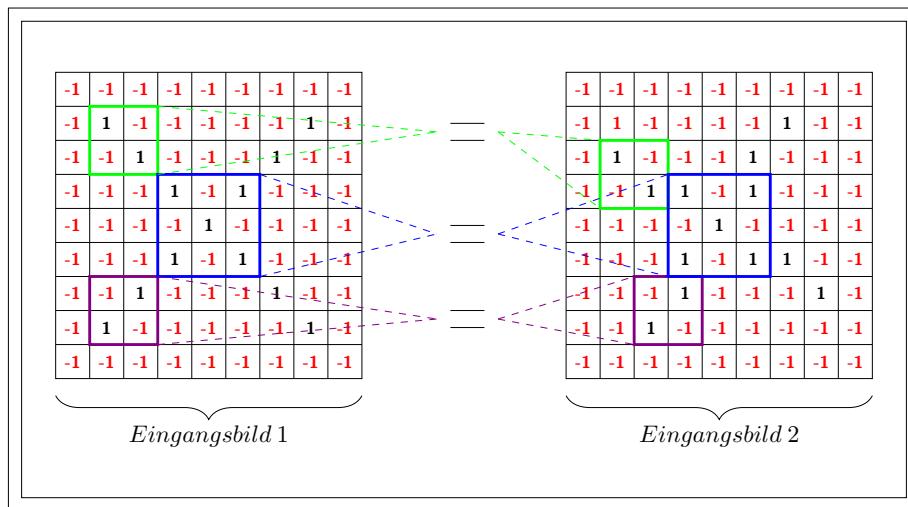


Abb. 2.2: Unterschiedliche Eingangsbilder mit ähnlichen Merkmalen [15]

2.1.1 Convolution

Bei der Faltung werden Filter (Feature) mit **mathematischen Operationen** auf Teilbereiche des Eingangsbildes angewendet. Das Ergebnis der Faltung ist eine **Feature-Map** [15]. Zur Veranschaulichung dient Abbildung 2.3. Der Filter hat beispielhaft eine Größe von 3x3px und wird auf die Teilbereiche im Abstand von 1px angewandt (s. grüne und blaue Markierung im Eingangsbild). Die mathematische Operation gleicht der Multiplikation der einzelnen Werte der Bildpixeln mit den Filterpixeln und der anschließenden **Mittelwertbildung**. Das Ergebnis wird an die jeweilige markierte Stelle in der Feature-Map übernommen [15].

Grüne Markierung:

$$x = \frac{(-1) \cdot 1 + (-1) \cdot (-1) + (-1) \cdot (-1) + (-1) \cdot (-1) + 1 \cdot 1 + (-1) \cdot (-1)}{9} \\ + \frac{(-1) \cdot (-1) + (-1) \cdot (-1) + 1 \cdot 1}{9}$$

$$x \approx 0.77$$

Blaue Markierung:

$$x = \frac{(-1) \cdot 1 + 1 \cdot (-1) + (-1) \cdot (-1) + (-1) \cdot (-1) + (-1) \cdot 1 + 1 \cdot (-1)}{9} \\ + \frac{(-1) \cdot (-1) + (-1) \cdot (-1) + (-1) \cdot 1}{9}$$

$$x \approx -0.11$$

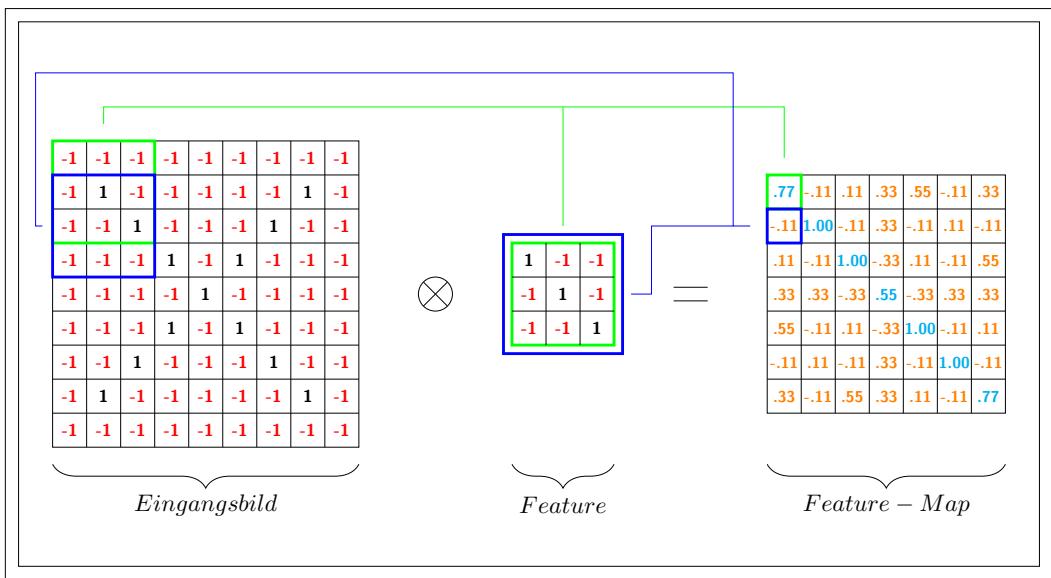


Abb. 2.3: Anwendung der Faltung mithilfe eines Filters [15]

Anhand eines Filters kann keine genaue Auswertung vorgenommen werden. Folglich ist die Anwendung unterschiedlicher Filter notwendig. Jeder Filter erzeugt eine andere Feature-Map, in welcher Auszüge des Originalbilds mit unterschiedlicher Gewichtung erkennbar sind (Abbildung 2.4).

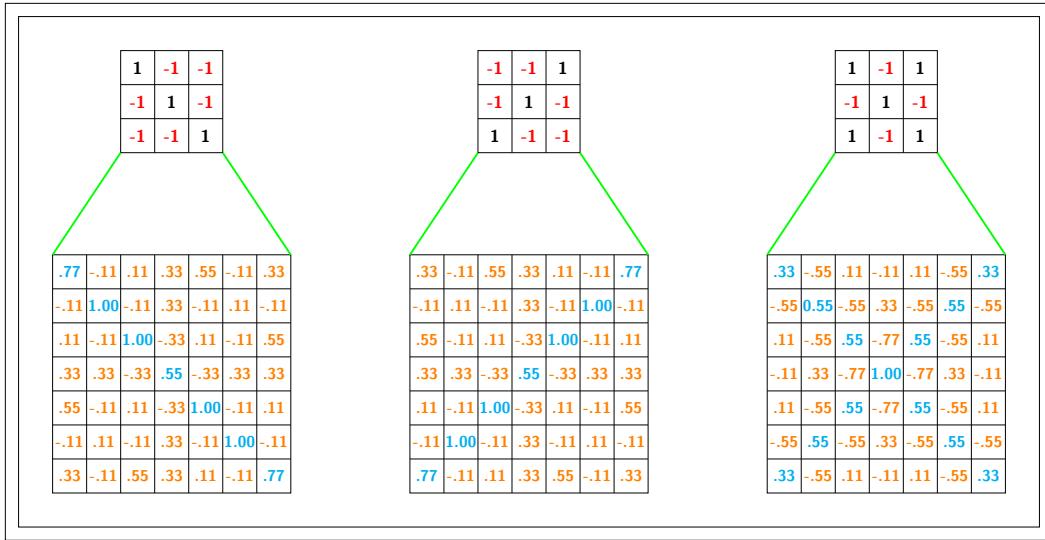


Abb. 2.4: Filtertypen und zugehörige Feature-Maps [15]

2.1.2 Pooling

Beim Maximum Pooling wird in Teilbereichen der Feature-Map nach einem maximalen Wert gesucht. Die Werte werden anschließend zu einer verkleinerten Feature-Map zusammengezogen. Dies hat den Vorteil, dass Speicherplatz gespart und die höchsten Gewichtungen der Merkmale extrahiert werden können [15]. In Abbildung 2.5 wird exemplarisch eine Fenstergröße von 2x2px mit einer Schrittweite von 2px gewählt.

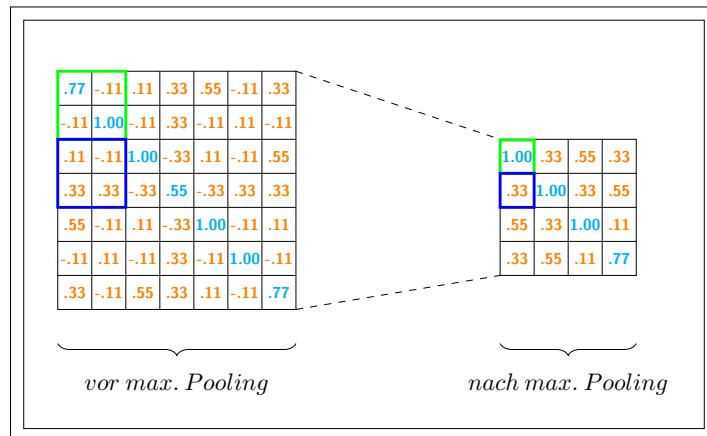


Abb. 2.5: Anwendung der Zusammenlegung (max. Pooling) [15]

2.1.3 Rectification

Im Rectification-Layer werden alle negativen Werte einer Feature-Map zu Null angenommen (Abbildung 2.7). Dieser Prozess reduziert den Rechenaufwand. Zur Anwendung kommt eine lineare Funktion (Rectified Linear Unit (ReLU)) (Abbildung 2.6), die Funktionswerte größer Null unverändert lässt [15].

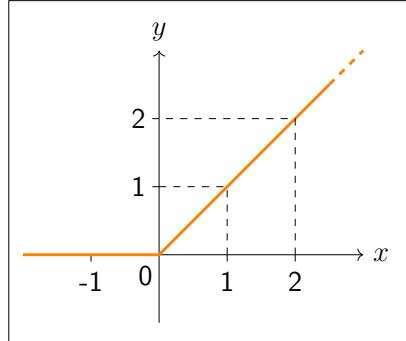


Abb. 2.6: Rectified Linear Unit [15]

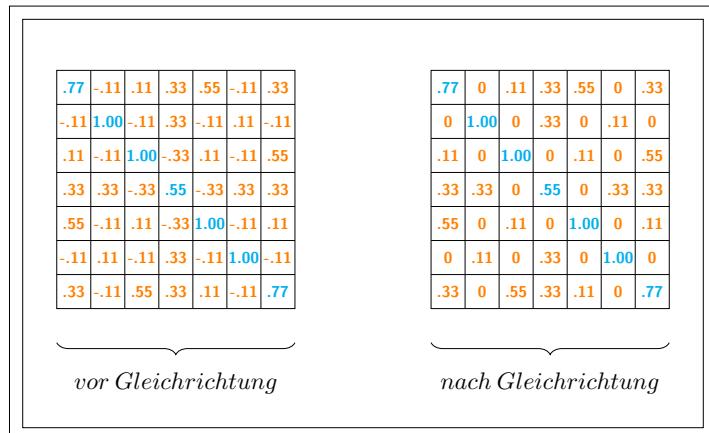


Abb. 2.7: Anwendung der Gleichrichtung (Rectification) [15]

2.1.4 Fully-Connected

Im Fully-Connected-Layer werden die Feature-Maps der vorangegangenen Schicht ausgewertet, um die Wahrscheinlichkeiten der Klassenzugehörigkeiten zu ermitteln [15]. In Abbildung 2.8 werden exemplarisch die fünf größten und fünf kleinsten Gewichtungen gemittelt. Das jeweilige Ergebnis spiegelt die Wahrscheinlichkeit wieder, dass das Eingangsbild

der Klasse „X“ oder „kein X“ zugeordnet werden kann [15].

Wahrscheinlichkeit Klasse „X“:

$$x = \frac{0.90 + 0.87 + 0.96 + 0.89 + 0.94}{5}$$

$$x \approx 0.912 (91.2\%)$$

Wahrscheinlichkeit Klasse „kein X“:

$$x = \frac{0.45 + 0.23 + 0.63 + 0.44 + 0.53}{5}$$

$$x \approx 0.456 (45.6\%)$$

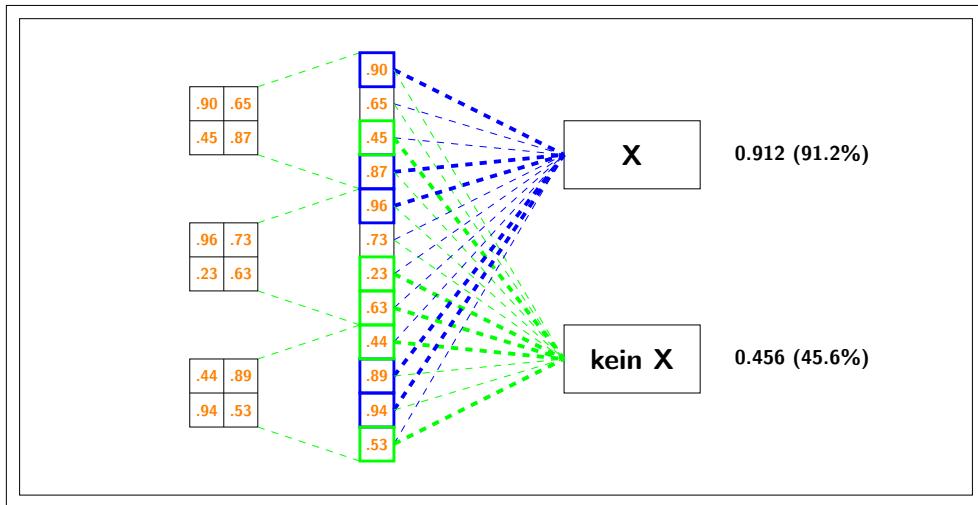


Abb. 2.8: Gewichtungen im Fully-Connected-Layer [15]

2.2 AlexNet

Das AlexNet ist ein Pre-Trained CNN und wurde von Alex Krizhevsky in Zusammenarbeit mit Ilya Sutskever and Geoffrey Hinton entwickelt [12]. Das CNN enthält alle vorher betrachteten Layer-Typen (s. Unterunterabschnitt 2.1.1 bis Unterunterabschnitt 2.1.4).

Das AlexNet wurde 2006 mit 1.2 Mio. Bildern mit einer Größe von jeweils 227x227x3 trainiert und erreichte eine Fehlerrate von 16.4% [2]. Das CNN ermöglichte einfachere und schnellere Fortschritte in der Gesichtserkennung. Mittlerweile wurde das CNN durch das leistungsstärkere GoogleLeNet mit 22 Layern abgelöst.

Das AlexNet besteht aus fünf Convolution- und drei Pooling-Schichten gefolgt von drei Fully-Connected-Layern. Zwischen den Schichten wird teilweise eine ReLU angewendet [10]. Die schematische Struktur ist in Abbildung 2.9 dargestellt.

Das AlexNet wird aufgrund des einfachen Aufbaus im Weiteren zur Bildverarbeitung zur Vermeidung von Kollisionen verwendet.

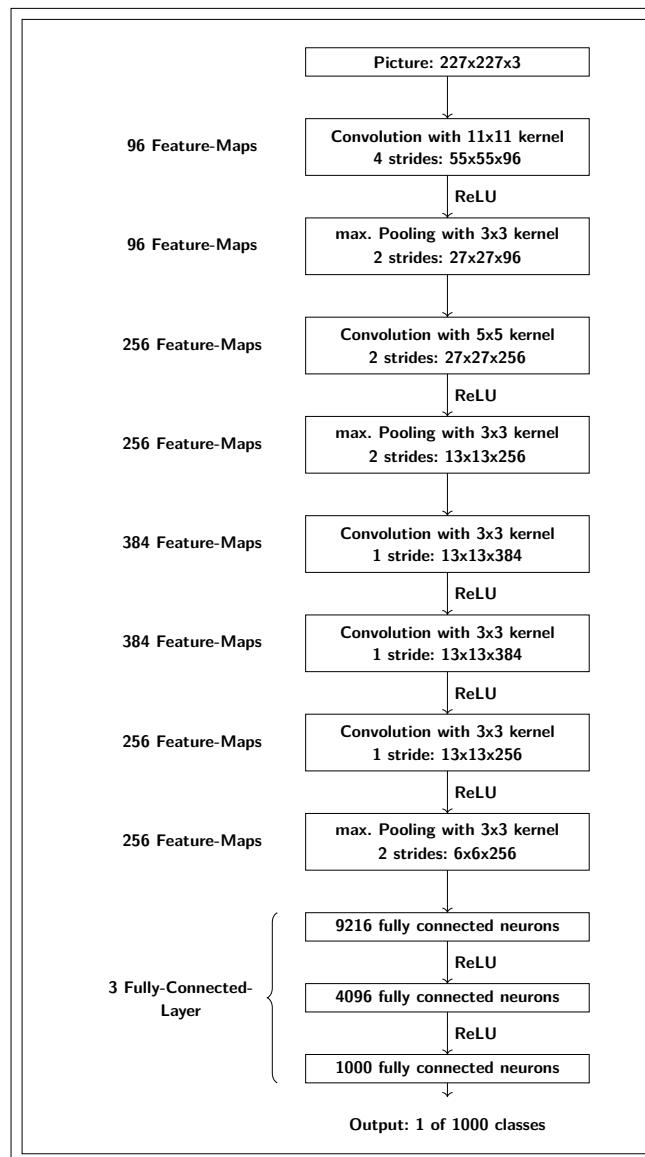


Abb. 2.9: Schematischer Aufbau des AlexNet [12]

2.3 Transfer-Learning

Um das AlexNet aus Unterabschnitt 2.2 sinnvoll nutzen zu können, wird die Maschine-Learning-Technik des **Transfer-Learning** angewandt. Beim Transfer-Learning wird ein bereits **vortrainiertes neuronales Netz** (z.B. CNN oder ResNet) auf ein **ähnliches Problem** angewendet (z.B: Bild- und Textverarbeitung) [13]. Die Anwendung dessen ist dann sinnvoll, wenn nur ein kleiner eigener Datensatz mit wenigen Klassen zur Verfügung steht. Durch den Transfer werden sämtliche Skills und Eigenschaften übernommen und auf das eigene Problem adaptiert (Abbildung 2.10). Dies reduziert den eigenen Ressourceneinsatz, ermöglicht eine schnellere Erstellung und Anwendung auf ein Problem und erhöht die Modellqualität [13].

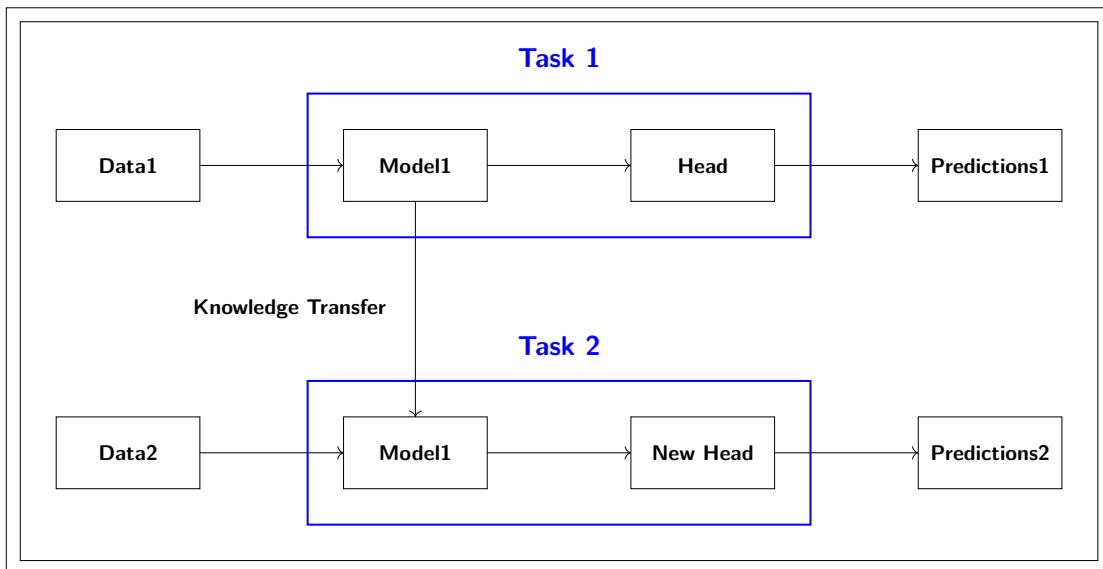


Abb. 2.10: Visualisierung des Transfer-Learning [9]

Das Transfer-Learning wird in die Bereiche **Pre-Training** und **Fine-Tuning** unterschieden. Beim Pre-Training werden die Feature-Maps des neuronalen Netzes unverändert übernommen und lediglich die Zuordnung der Klassen (Predictions) an die Klassen des neuen Problems angepasst und trainiert [13].

Beim Fine-Tuning werden zusätzlich die Feature-Maps entsprechend auf das spezifische Problem angepasst. Hierzu werden größere eigene Datenmengen benötigt [13].

Aufgrund des geringen eigenen Datensatzes wird das Pre-Training des AlexNets bevorzugt angewendet.

3 Versuchsaufbau und Inbetriebnahme

Bevor das Roboterfahrzeug (im Folgenden bezeichnet mit „Jetbot“) programmiert und getestet werden kann auf seine Fähigkeiten sich autonom mittels CNN und Transfer Learning fortzubewegen und dabei jegliche Kollisionen zu vermeiden, muss dieses zunächst gebaut und in Betrieb genommen werden. Dieser Abschnitt wird einen kurzen Überblick über die verwendete Hardware geben und kurz beschreiben, wie der Jetbot zum Programmieren aufgesetzt wird.

3.1 Benötigte Komponenten

Tabelle 3.1 zeigt eine Liste aller benötigten Komponenten und Materialien für den Roboter. Von besonderer Wichtigkeit für die Anwendung Neuronaler Netze in diesem Versuch ist zum einen das Gehirn des Jetbot, der **Jetson Nano Mikrocontroller** von Nvidia. Dabei handelt es sich konkret um einen auf ARM basierenden Computer, der dafür entwickelt wurde mittels seiner integrierten GPU (Graphical Processing Unit) mehrere neuronale Netze parallel zu berechnen. Er ist ausgestattet mit einer Quad-core ARM Cortex-A57 MPCore CPU, einer NVIDIA Maxwell GPU mit 128 CUDA cores und GB 64-bit LPDDR4, 1600MHz 25.6 GB/s RAM. Die zweite wichtige Komponente ist die Kamera des Roboters. Dabei handelt es sich um eine 8MP 160° **FOV Kamera** mit IMX219 Sensor und 3280x2464 Pixeln Auflösung, welche für Gesichtserkennung, Objektklassifizierung, und Echtzeitmonitoring entwickelt wurde.



Abb. 3.1: Jetson Nano Mikrocontroller und 160° Weitwinkelkamera

Nr.	Bauteil	Anzahl	Bemerkung
1	Jetson Nano	1	4 GB RAM Variante
2	Mikro SD Karte	1	mind. 64 GB
3	Karosserie	1	—
4	Kamera-Befestigung	1	—
5	Abstandshalter für Kamera aus Acryl	1	—
6	Kamera	1	IMX219-160 Weitwinkel
7	WLAN-Stick	1	RTL8121 Chipsatz
8	Erweiterungsboard	1	für Jetson Nano
9	Motor	2	„TT“-Bauform
10	Rad	2	60mm Durchmesser
11	Kugelrolle	2	1" Durchmesser
12	Steckeradapter EU	1	—
13	Ladegerät	1	12.6 V
14	Gamepad	1	kabellos
15	Montagewerkzeug	2	Schraubenzieher
16	6-Pin Kabel	1	9 cm
17	Schrauben	38	M2 und M3 Gewinde
18	Lüfter	1	Bauform 4010
19	SD-Kartenlesegerät	1	—

Tab. 3.1: Auflistung der benötigten Bauteile für den Aufbau des Jetbot



Abb. 3.2: Darstellung aller verwendeten Komponenten für die Montage des Jetbots

3.2 Hardware-Setup

Zunächst werden die Motoren auf der Grundfläche des Gehäuses verschraubt. Dieses kann nachfolgen geschlossen werden mit den restlichen Karosserieteilen. Es bietet sich an im nächsten schritt die Räder und Kugelrollen zu montieren, so dass das Gefährt aufrecht stehen kann. Darauf werden die Akkus in dem Erweiterungsboard installiert sowie Abstandshalter an den Ecken angebracht. Das Erweiterungsboard kann dann auf der Karosserie angebracht werden mit vier M2 Schrauben. Auf den im vorherigen Schritt montierten Abstandshaltern wird der Jetson Nano Controller befestigt. Über das 6-Pin Kabel wird anschließend das Erweiterungsboard mit dem Jetson Nano verbunden. Im Anschluss wird der Lüfter auf dem Kühlkörper angebracht und das Lüfterkabel in den PWM-Anschluss gesteckt. Im letzten Schritt wird die Kamerabefestigung am Gehäuse angebracht, der Abstandshalter aus Acryl an an dieser angeschraubt und Abschließend die Kamera montiert. Abbildung 3.3 zeigt den fertigen Aufbau des Jetbot Roboterfahrzeugs.

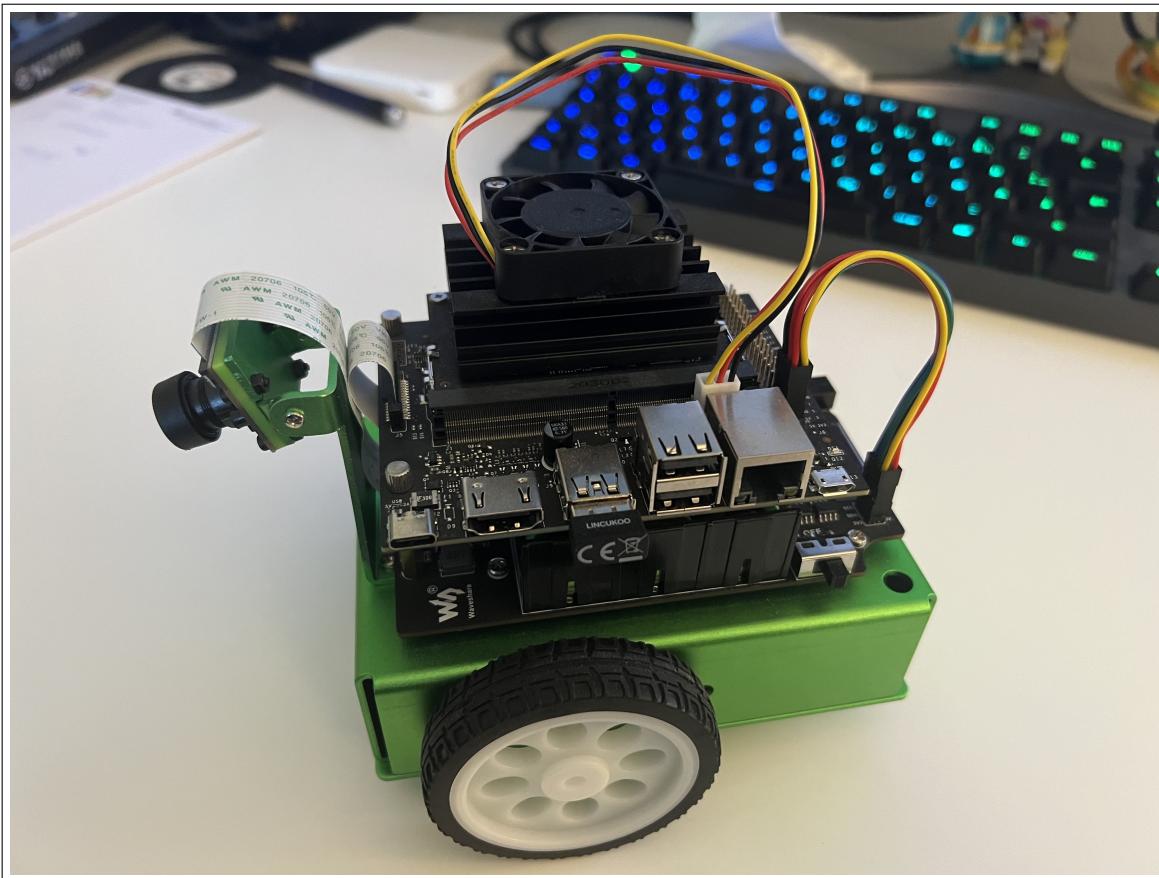


Abb. 3.3: Aufbau des Jetbot-Roboterfahrzeugs

3.3 Software-Setup

Der erste Schritt ist es sich ein von Nvidia für den Jetson Nano bereitgestelltes Image auf die SD-Karte herunterzuladen. Dieses beinhaltet Ubuntu Linux als Betriebssystem, sämtliche notwendigen Treiber für die Betriebsmittel wie z. B. die Kamera und besitzt bereits eine Installation für Python inklusive Jupyter und PyTorch. Worum es sich dabei handelt wird in Abschnitt 4 geklärt.

Die folgenden Schritte sind notwendig, um das Image auf die SD-Karte zu spielen:

1. SD-Karte über Lesegerät an beliebigen PC anschließen.
2. Mit Etcher das heruntergeladene Image auswählen und auf die SD-Karte übertragen.
3. SD-Karte in den Jetson Nano einstecken.

Der nächste Schritt kann abweichen je nach gewählter Methode. Grundsätzlich ist das Ziel den Jetson Nano mit einem WLAN-Netzwerk zu verbinden. Dazu kann entweder ein Monitor an den HDMI-Port, sowie Maus und Tastatur per USB angeschlossen werden. Nach dem Booten des Systems ist es dann möglich über die Netzwerkeinstellungen eine Verbindung zum WLAN herzustellen. Alternativ ist es möglich über z. B. Putty unter Windows oder über das Terminal in Linux oder MACOS auf den Jetson Nano zuzugreifen. Anschließend wird die Verbindung hergestellt über das Kommando

```
sudo nmcli device wifi connect <SSID> password <PASSWORD>.
```

Der Mikrocontroller verbindet sich nun nach dem Einschalten automatisch mit dem ausgewählten Netzwerk. Nachfolgend kann der Jetbot über den Browser eines beliebigen Endgerätes genutzt werden. Dazu sind folgende Schritte notwendig:

1. Jetbot einschalten über den Power-Schalter
2. Warten, bis der Bootvorgang abgeschlossen ist
3. Die IP-Adresse am piOLED-Display ablesen
4. Über den Browser eines beliebigen Gerätes im selben Netzwerk navigieren zu
http://<jetbot_ip_address>:8888
5. Einloggen mit dem Passwort jetbot

Nachdem über den Browser erfolgreich eine Verbindung hergestellt wurde, wird die Ansicht aus Abbildung 3.4 gezeigt.

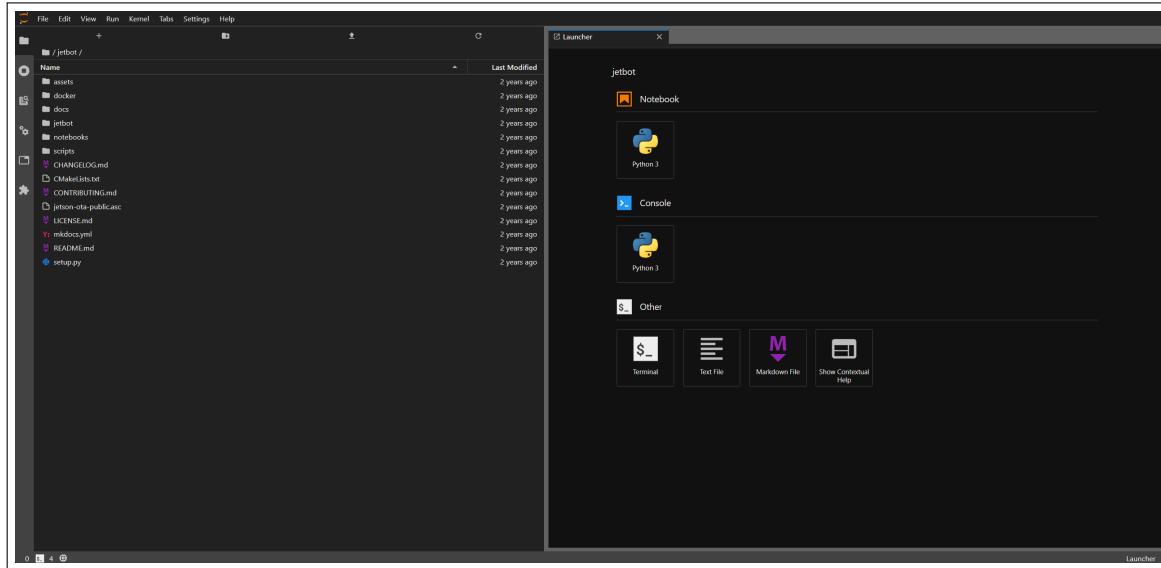


Abb. 3.4: Bedienoberfläche des Jetbots über JupyterLab

Zu sehen ist die Oberfläche JupyterLab. Dabei handelt es sich um ein Interface für Jupyter Notebooks. Solch ein Notebook ist grundsätzlich ein aus dem Web aufrufbares Python-Programm. Python meint hier die Programmiersprache in der Version 3.X. Das besondere an einem Jupyter Notebook ist, dass der Code in diesem nicht zwangsläufig in der gegebenen Reihenfolge ausgeführt werden muss. Weiterhin werden bereits berechnete Ergebnisse gespeichert und können zu jedem Zeitpunkt weiter genutzt werden, ohne das ein erneutes Ausführen nötig ist. Das ist besonders hilfreich bei dem Trainieren von neuronalen Netzen, da ein Code-Segment mitunter Stunden oder Tage braucht, um abgearbeitet zu werden. Müsste dies jedes Mal auf ein neues geschehen, würde das viel Zeit in Anspruch nehmen. Abbildung 3.5 zeigt ein beispielhaftes Notebook. Als weiterer Vorteil ist zu erkennen, dass ebenso Text wie Code eingebunden werden kann. Somit bietet es sich an die Dokumentation bzw. Beschreibung des Programms in der selben Datei vorzunehmen. Mit Ausblick auf die Programme in den umgesetzten Notebooks für den Jetbot ist zu erwähnen, dass dies dort ebenso vorgenommen wurde. Sämtliche Jupyter Notebooks für die Umsetzung der Kollisionsvermeidung sind im Anhang zu finden.

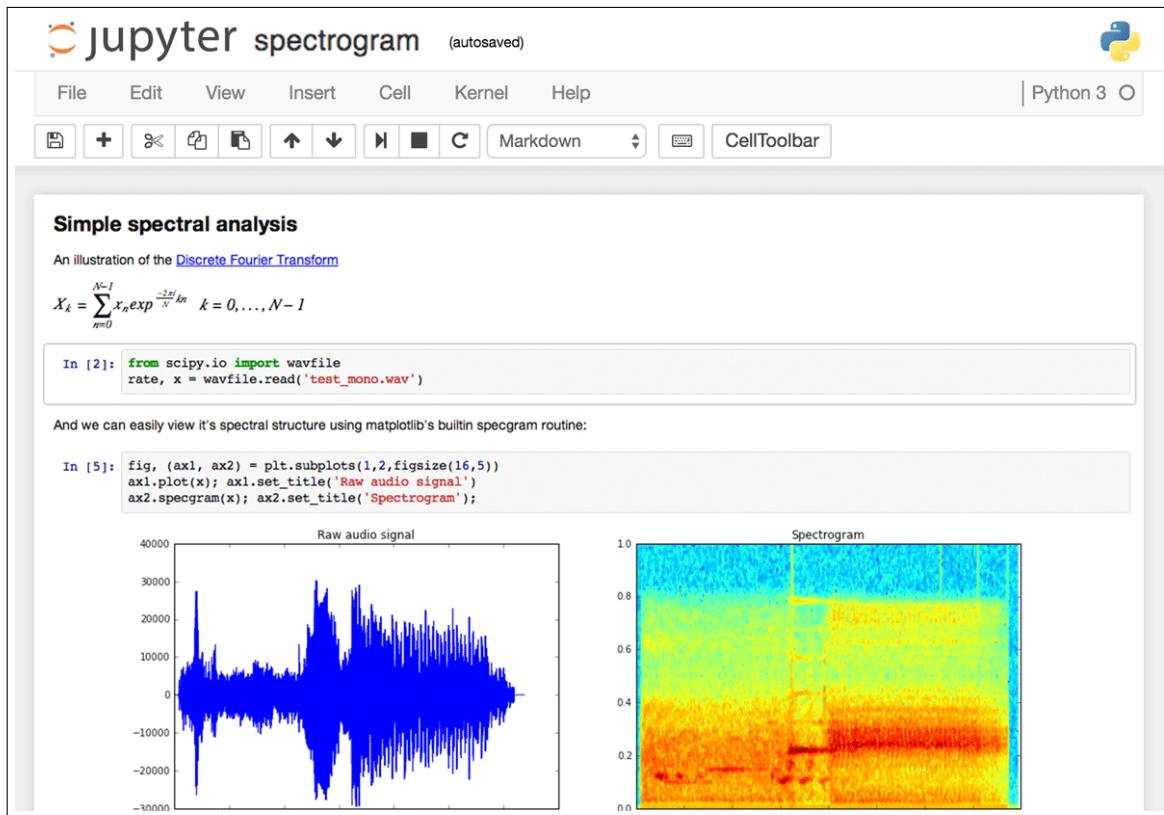


Abb. 3.5: Beispielhafte Abbildung eines Jupyter Notebooks mit Python-Code und Beschreibung in Textform

4 Umsetzung der Kollisionsvermeidung

Dieser Abschnitt gliedert sich in vier Unterabschnitte. Der erste beschäftigt sich damit, wie aus einem Jupyter Notebook (in der Programmiersprache Python) der Jetbot bzw. dessen Motoren angesteuert werden können, so dass sich dieser bewegt. Darauf folgt im nächsten Unterabschnitt das sammeln von Daten, konkret Bildern, um das CNN später zu trainieren. Mittels dieser Trainingsdaten wird im nächsten Unterabschnitt ein Modell angelernt mittels Transfer-Learning, welches im letzten Unterabschnitt in einer Live-Demo Anwendung findet.

Alle aufgezeigten Inhalte finden sich in Code-Form dokumentiert im Anhang wieder und können dort im Detail nachvollzogen werden.

4.1 Ansteuern des Jetbot Roboters

Um mit der Programmierung des Jetbots zu beginnen, muss zunächst die von der Nvidia-Jetbot-Community bereitgestellte Klasse „Robot“ importiert werden. Diese ist Teil des Jetbot-Package, welches in Python eingebunden werden kann. Dies ist bereits geschehen, da für sie Inbetriebnahme (siehe Abschnitt 3) ein vorkonfiguriertes Image verwendet wurde. Mit der Klasse können die Motoren des Roboters angesteuert werden.

Eine hilfreiche Ergänzung sind die sogenannten „traitlets“, über welche es möglich ist Widgets zu implementieren, mit denen man in einer grafischen Oberfläche im Browser den Jetbot Roboter steuern kann. Über diese ist es auch möglich Funktionen mit Events zu verknüpfen. So kann dann z. B. über einen Button-Druck eine Vorwärtsbewegung ausgelöst werden.

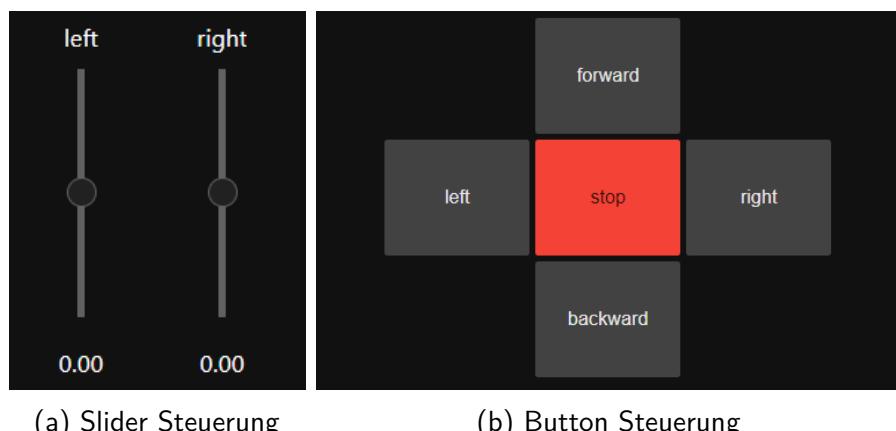


Abb. 4.1: Widgets zum Steuern des Jetbots mithilfe von traitlets

Eine dritte grundlegende Funktion ist der „Killswitch“. Dabei handelt es sich konkret um eine Sicherheitsfunktion, die dafür sorgt, dass der Roboter in seiner Bewegung stoppt, wenn er die Verbindung zur JupyterLab Web-Oberfläche verliert.

4.2 Aufnehmen von Trainingsdaten

Nun wo die Möglichkeit besteht den Roboter per Code manuell zu fahren, ist der nächste Schritt, dass dieser sich von alleine ohne menschliches Einwirken fortbewegen kann. Schwierig ist dabei die Anforderung, dass sämtliche Bewegungen kollisionsfrei stattfinden sollen. Dazu muss der Jetbot bzw. dessen CNN mit Trainingsdaten angelernt werden. Damit der Roboter in mehreren Epochen effektiv trainiert werden kann müssen diese Daten zunächst aufgenommen werden.

Der Ansatz für das kollisionsfreie Bewegen ist in seinem Konzept sehr simpel. Ziel ist es eine virtuelle „Safety Bubble“ um den Roboter zu erstellen. In dieser kann der Jetbot sich frei im Kreis drehen, ohne dass er mit Objekten kollidiert oder von einem Vorsprung herunterfällt. Nicht berücksichtigt wird in diesem Ansatz, dass sich natürlich auch Objekte außerhalb des Sichtfeldes in die sichere Zone hinein bewegen können. Er kann sich also ausschließlich nicht von selbst durch seine eigenen Bewegungen in eine „gefährliche Situation“ begeben.

Konkret umgesetzt wird der Ansatz über den einzigen aber höchst effektiven Sensor des Roboters, die Weitwinkelkamera. Zunächst wird der Roboter manuell in Szenarien platziert, in denen seine Sicherheitsblase verletzt wird. Diese werden als „blocked“ gelabelt. Dazu wird ein Bild gespeichert von dem, was der Roboter sieht, zusammen mit dem Label.

Zweitens wird der Roboter manuell in Szenarien platziert, in denen es sicher ist sich ein Stück vorwärts zu bewegen. Diese Szenarien werden als „free“ gelabelt. Ebenfalls wird ein Bild zusammen mit dem Label abgespeichert.

Nachdem genügen Trainingsdaten aufgenommen wurden, können diese auf einen mit einer GPU ausgestatteten Rechner geladen werden, wo das neuronale Netzwerk trainiert wird vorherzusagen, ob die Sicherheitsblase des Roboters verletzt wurde anhand der Live-Bilder, die dieser aufnimmt.

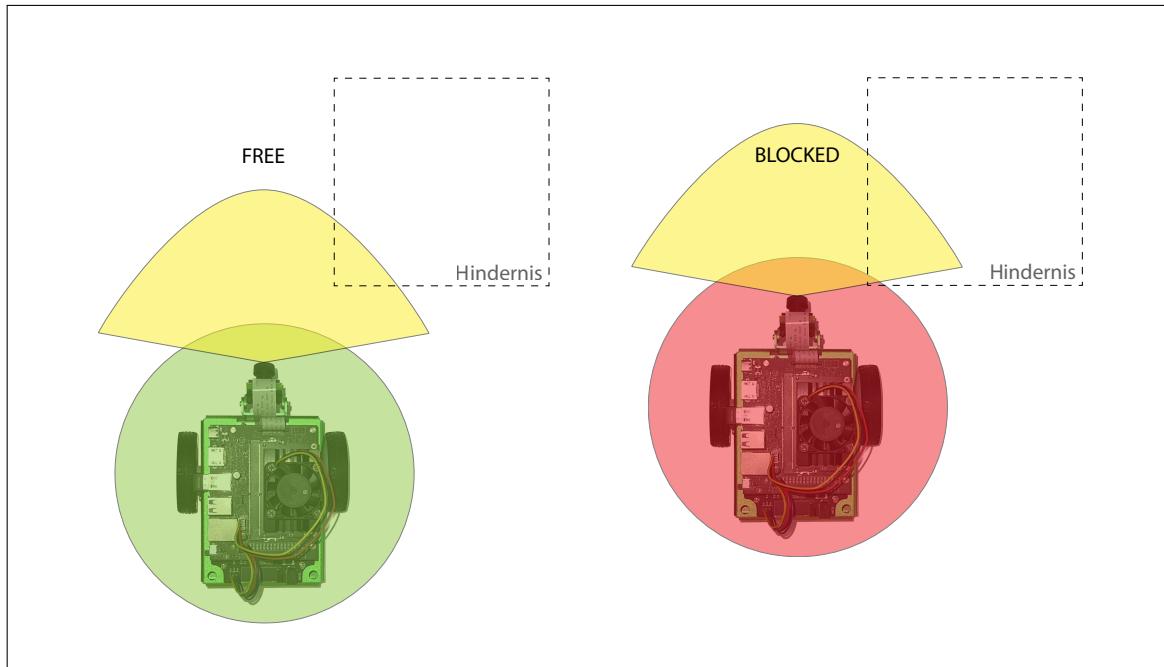


Abb. 4.2: Sicherheitsblase des Jetbot Roboterfahrzeugs im free und blocked Szenario

4.3 Pre-Training des CNN AlexNet

Im nächsten Schritt wird das neuronale Netz trainiert, so dass ein möglichst ideales Modell erstellt werden kann, welches im realen Betrieb des Roboters Einsatz findet. Für das Trainieren des Modells wird die Python Bibliothek PyTorch verwendet. Dabei handelt es sich um eine Sammlung von Funktionen und Klassen, die für „Deep Learning“ entwickelt wurden.

Zunächst werden die in zwei Klassen unterteilten Bilder auf einen Rechner transferiert, welcher über einen Grafikprozessor (GPU) verfügt. Das könnte zum einen der Jetbot selbst sein oder ein separater Computer. Je nach gewähltem CNN und dessen Anzahl an Layern ergibt es jedoch Sinn, einen Leistungsstarken Computer mit einer schnellen Grafikkarte zu verwenden. Wie bereits Eingangs erwähnt wurde, soll das CNN AlexNet verwendet werden, welches moderate Anforderungen an die Rechenleistung stellt. Somit ist es grundsätzlich möglich bei kleiner Anzahl an Epochen (z. B. 30 Epochen) das Trainieren auf dem Jetson Nano Mikrocontroller selbst durchzuführen.

Bevor die Bilddaten in Kombination mit dem AlexNet verwendet werden können, müssen diese an dessen Vorgaben angepasst werden. Dazu zählt zum einen die Anpassung der Bildgröße auf eine Auflösung von 224x224 Pixeln, zum anderen das Umwandeln der Daten in Tensoren, welche im gleichen Schritt normalisiert werden. Ein Tensor meint dabei eine Anordnung von Zahlen entlang n Achsen. Die Zahl n heißt die Stufe des Tensors.

Als Nächstes wird der Datensatz in einen Trainings- und einen Testsatz aufgeteilt. Der Testdatensatz wird verwendet, um die Genauigkeit des trainierten Modells zu überprüfen. Nachdem alle Vorbereitungen getroffen wurden, kann nun das eigentliche neuronale Netz definiert werden. Das „torchvision“ Paket (aus der PyTorch Bibliothek) bietet bereits eine Sammlung von vortrainierten Modellen, von welchen das AlexNet ausgewählt wurde. Dieses Modell, welches bereits mit Millionen von Bildern trainiert wurde, kann mit Hilfe des Transfer Learning aus Unterabschnitt 2.3 auf den kleinen Datensatz an Bildern angewendet werden, der im vorherigen Unterabschnitt aufgenommen wurde. Wichtige Merkmale, die beim ursprünglichen Training des vortrainierten Modells gelernt wurden, können damit für die neue Aufgabe (das kollisionsfreie Fahren) wiederverwendet werden.

Das CNN wird in 30 Epochen trainiert, wobei aus jeder Epoche das Modell mit der besten Leistung abgespeichert wird. Pro Epoche wird einmal der komplette Datensatz an Bildern durchgearbeitet.

Nach Beendigung aller Epochen wurde ein möglichst optimales Modell errechnet, welches im nächsten Unterabschnitt im Live-Betrieb eingesetzt wird.

4.4 Live-Demo des trainierten CNN

Im letzten Unterabschnitt der Umsetzung wird das trainierte Modell eingesetzt, um dem Jetbot die Fähigkeit zu geben zu erkennen, ob ein Szenario als free oder blocked bewertet werden muss, so dass beim Fahren sämtliche Kollisionen vermieden werden können.

Dazu wird das Modell auf den Jetson Nano geladen. Im nächsten Schritt muss dafür gesorgt werden, dass die Live-Kameradaten wie bereits die Trainingsbilder an die Anforderungen des AlexNet angepasst werden. Dieser Prozess wird als Vorverarbeitung (engl. Preprocessing) bezeichnet. Dabei werden über das Python Paket opencvCV die Bilddaten von BGR zu RGB konvertiert, die Daten normalisiert und dann von der CPU auf die GPU transferiert. Die Umsetzung des Preprocessing erfolgt in einer Funktion, die später aufgerufen werden kann, wenn ein neues Bild verarbeitet werden muss.

Weiterhin muss wie auch schon in Unterabschnitt 4.1 die Roboter-Klasse importiert werden, so dass die Motoren des Jetbots angesteuert werden können.

Ist dies geschehen kann die Funktion „update“ implementiert werden, welche jedes mal aufgerufen wird, wenn die Kamera ein neues Bild aufnimmt. In dieser werden drei Aufgaben der Reihe nach ausgeführt:

1. Vorverarbeitung des Kamerabildes
2. Ausführung des neuronalen Netzes
3. Wenn das CNN blocked ausgibt, nach links fahren,
und wenn es free ausgibt, dann vorwärts fahren.

Der letzte Schritt ist es diese Funktion mit der Weitwinkelkamera zu verbinden. Dazu wird eine observe-Funktion (Beobachter-Funktion) mit dem Value-traitlet (siehe Unterabschnitt 4.1 der Kamera verbunden, welche dann die update-Funktion aufruft, wenn der Value (Wert) der Kamera eine Änderung erfährt.



Abb. 4.3: Kamera Aufnahmen des Jetbots während des kollisionsfreien Fahrens

Der Jetbot kann nach bedarf wieder gestoppt werden über einen unobserve-Aufruf und die Stopp-Routine der Roboterklasse. Alle implementierten Programme und Funktionen können im Anhang nachgelesen werden. Dort befinden sich sämtliche Jupyter Notebooks.

5 Ergebnisse und Fazit

Das gesetzte Ziel, ein kleines Roboterfahrzeug zu befähigen über CNN's frei im Raum zu fahren, ohne dabei mit Hindernissen zu kollidieren, konnte erreicht werden. Dabei wurden als Grundlage die Fragen beantwortet, was ein Convolutional neural Network (CNN) ist, wie diese aufgebaut sind und warum Transfer Learning eine gute Wahl ist, um schnell mit wenig Trainingsdaten ein Modell bezüglich bestimmter Fähigkeiten anzulernen.

Jedoch muss festgehalten werden, dass es nur in den meisten Fällen möglich war Kollisionen zu vermeiden. Besonders schlecht beleuchtete Umgebungen oder unbekannte Gebiete in der Wohnung sorgten für erhöhte Fehlerraten, wo Kollisionen nicht vermieden werden konnten. Das Konzept scheint also in diesem simplen Ansatz nicht auf ein realen Verkehrsteilnehmer übertragbar zu sein. Dafür müsste das Modell deutlich intensiver mit mehr Bildern aber auch mehr Klassen trainiert werden, um Situationen bzw. Szenarien besser differenzieren und einschätzen zu können.

Dennoch konnte nachgewiesen werden, dass es mit wenigen Zeilen Code bereits möglich ist eine fähige künstliche Intelligenz zu implementieren, die einfache Aufgaben zuverlässig umsetzen kann.

Weiterhin hat sich gezeigt, dass die Nutzung von CNN's mit mehr Layern (z. B. ResNet50) dafür sorgt, dass die antrainierten Modelle akkurate werden. Jedoch erhöhen sich damit auch die Anforderungen an die Hardware, insbesondere die GPU. Es konnte festgestellt werden, dass die Reaktionszeit des Jetbots sichtbar langsamer war, desto komplexer das genutzte neuronale Netz ist. Das AlexNet hat sich als Sweetspot

erwiesen mit dem gewählten Jetson Nano. Es gibts jedoch bereits heute schon sehr kompakte und deutlich effizientere Mikrocontroller (meist von Nvidia) die eine tausendfache Leistung im Vergleich zu der genutzten Hardware haben.

Das Pre-Training hat sich hingegen als eine kleinere Herausforderung dargestellt, da dieses auf einem separaten Computer stattfinden konnte, welcher mit einer Hochleistungsgrafikkarte (Nvidia RTX 3090) ausgestattet ist.

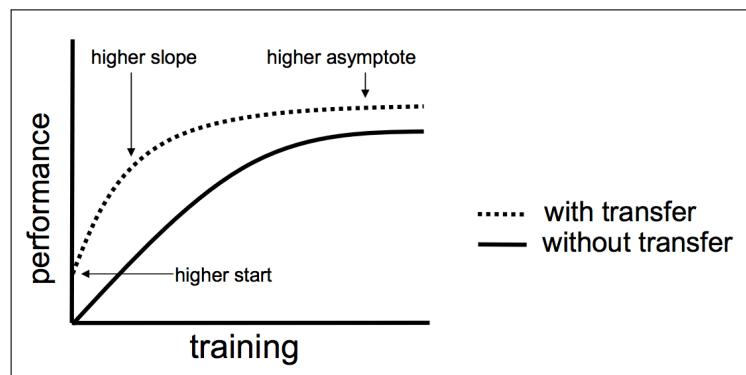


Abb. 5.1: Vorteile des Transfer Learning [7]

6 Ausblick - Autonomes Fahren

Wie bereits bei der Analyse der Ergebnisse festgestellt wurde, ist das Erkennen von Hindernissen mit anschließendem Ausweichen nur ein erster kleiner Schritt auf dem Weg zum autonomen Fahren. Und selbst diese einfach anmutende Aufgabe scheint nie Perfekt umsetzbar zu sein. Das Ziel wird es in Zukunft werden, über mehr Trainingsdaten, komplexere neuronale Netze und schnellere Hardware das Fahren sicherer zu gestalten.

Da die Kollisionsvermeidung am Ende jedoch leichter umzusetzen war, als Anfangs erwartet, findet sich in diesem Ausblick auch eine weitere Live-Demo wieder. Konkret konnte zusätzlich die Fähigkeit umgesetzt werden einer Straße bzw. vielmehr den Straßenmarkierungen zu folgen. Dazu wurden wie auch schon in Abschnitt 4 drei Phasen durchlaufen:

1. Datensammlung
2. Trainieren des CNN
3. Einsatz des trainierten Modells

Der wesentliche Unterschied im Vergleich zur Umsetzung der Fähigkeit zur Kollisionsvermeidung bestand dabei vor allem in der Aufnahme und Klassifizierung der Daten. Der Jetbot wurde dafür auf einer Klemmbaustein-Straße an verschiedenen Positionen platziert. An den Positionen wurden erneut Bilder aufgenommen, diesmal jedoch mit einer zusätzlichen grünen Markierung, welche dem neuronalen Netz zeigen soll, wohin der Roboter sich bewegen sollte, wenn er sich in einer vergleichbaren Position befindet.

Auch hier war es möglich den Jetbot auf einer kleinen Demo-Strecke im Kreis fahren zu lassen.

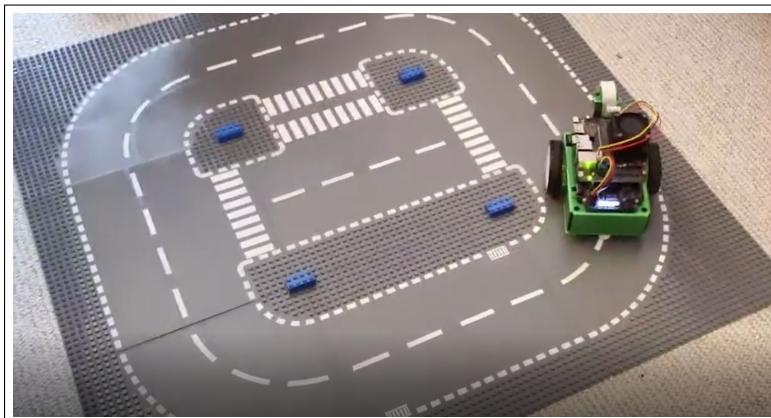


Abb. 6.1: Testversuch zum autonomen Fahren unter Vorgabe von Straßenmarkierungen

Codebeispiele in Form von Jupyter Notebooks finden sich auch hierfür im Anhang wieder.

7 Anhang - Jupyter Notebooks

Da der Anhang eine gewisse Länge besitzt und in mehrere Abschnitte unterteilt werden kann, dient diese Seite als eine Art Inhaltsverzeichnis für ausschließlich den Anhang. Zu finden sind im Folgenden sämtliche Jupyter Notebooks zu den im Fließtext beschriebenen Funktionen und Fähigkeiten des Jetbot Roboterfahrzeugs.

Grundlegende Funktionen:

[Steuern des Jetbot](#)

Kollisionsvermeidung:

[Datensammlung](#)

[Trainieren des CNN](#)

[Einsatz des trainierten Modells](#)

Spurverfolgung:

[Datensammlung](#)

[Trainieren des CNN](#)

[Einsatz des trainierten Modells](#)

grundlegende_bewegungen

September 25, 2022

1 Grundlegende Bewegungen

In diesem Notebook werden die Grundlagen der Steuerung des Jetbots behandelt.

1.0.1 Importieren der Roboter-Klasse

Um mit der Programmierung des Jetbots zu beginnen muss die Klasse `Robot` importiert werden. Diese Klasse erlaubt es die Motoren des Roboters einfach zu steuern. Sie ist in dem Paket `jetbot` enthalten.

```
[ ]: from jetbot import Robot
```

Nachdem die Klasse `Robot` importiert wurde kann die Klassen-*Instanz* wie folgt initialisiert werden:

```
[ ]: robot = Robot()
```

1.0.2 Steuern des Roboters

Nun wo die `Robot`-Instanz erstellt wurde unter dem Namen "robot" kann die Instanz verwendet werden um den Roboter zu steuern. Um den Roboter mit 30% seiner maximalen Geschwindigkeit gegen den Uhrzeigersinn drehen zu lassen kann folgendes Codebeispiel aufgerufen werden:

```
[ ]: robot.left(speed=0.3)
```

Um den Roboter wieder zu stoppen muss die `stop`-Methode aufgerufen werden.

```
[ ]: robot.stop()
```

Falls der Roboter nur für eine bestimmte Zeit aktiv sein soll kann das Python-Paket `time` importiert und verwendet werden.

```
[ ]: import time
```

Dieses Paket definiert die Funktion `sleep`, die die Codeausführung für die angegebene Anzahl von Sekunden blockiert bevor der nächste Befehl ausgeführt wird. Das folgende Codebeispiel lässt den Roboter für eine halbe Sekunde nach links drehen.

```
[ ]: robot.left(0.3)
      time.sleep(0.5)
      robot.stop()
```

1.0.3 Motoren einzeln ansteuern

Bis jetzt konnten die Motoren nur über die Befehle `left`, `right`, usw. angesteuert werden. Es besteht jedoch auch die Möglichkeit die Geschwindigkeit jedes Motors einzeln zu setzen. Grundsätzlich existieren dafür zwei Möglichkeiten.

Die erste Möglichkeit ist es die Methode `set_motors` aufzurufen. Um zum Beispiel eine Sekunde lang einen Linksbogen zu fahren könnten der linke Motor auf 30% und der rechte Motor auf 60% eingestellt werden:

```
[ ]: robot.set_motors(0.3, 0.6)
      time.sleep(1.0)
      robot.stop()
```

Alternativ existiert eine zweite Möglichkeit das gleiche Ergebnis zu erzielen.

Die Klasse `Robot` hat zwei Attribute mit den Namen `left_motor` und `right_motor`, die jeden Motor einzeln repräsentieren. Diese Attribute sind Instanzen der Klasse `Motor`, die jeweils ein `value`-Attribut besitzen. Dieses `value`-Attribut ist ein `traitlet` das `events` erzeugt, wenn ihm ein neuer Wert zugewiesen wird. In der `Motor` Klasse wird eine Funktion hinzugefügt, die die Motorbefehle aktualisiert, sobald sich der Wert ändert.

Um also genau das Gleiche wie oben zu erreichen könnte folgender Code ausgeführt werden:

```
[ ]: robot.left_motor.value = 0.3
      robot.right_motor.value = 0.6
      time.sleep(1.0)
      robot.left_motor.value = 0.0
      robot.right_motor.value = 0.0
```

You should see the robot move in the same exact way!

1.0.4 Verbinden der Motoren zu traitlets

Eine hilfreiche Fähigkeit dieser `Traitlets` ist, dass sie auch mit anderen Traitlets verknüpfen werden können. Das ist sehr praktisch, da Jupyter Notebooks es erlauben grafische `Widgets` zu erstellen, die besagte Traitlets verwenden. Das bedeutet, dass die Motoren mit `Widgets` verknüpft werden können, um sie vom Browser aus zu steuern oder einfach nur Daten zu Visualisieren.

Als Beispiel für diese Fähigkeit dienen zwei Schieberegler, über die die Geschwindigkeit der Motoren individuell eingestellt werden kann.

```
[ ]: import ipywidgets.widgets as widgets
      from IPython.display import display

      # create two sliders with range [-1.0, 1.0]
      left_slider = widgets.FloatSlider(description='left', min=-1.0, max=1.0, step=0.01, orientation='vertical')
      right_slider = widgets.FloatSlider(description='right', min=-1.0, max=1.0, step=0.01, orientation='vertical')
```

```
# create a horizontal box container to place the sliders next to each other
slider_container = widgets.HBox([left_slider, right_slider])

# display the container in this cell's output
display(slider_container)
```

Bis jetzt haben die Schieberegler noch keinen Effekt auf den Jetbot. Diese müssen zunächst mit den Motoren verbunden werden. Dies geschieht mit der Funktion `link` aus dem `traitlets`-Paket.

```
[ ]: import traitlets

left_link = traitlets.link((left_slider, 'value'), (robot.left_motor, 'value'))
right_link = traitlets.link((right_slider, 'value'), (robot.right_motor, □
    ↴'value'))
```

Werden die Regler nun vorsichtig bewegt ist zu erkennen, dass der zugehörige Motor sich entsprechend der Position des Reglers dreht.

Die erstellte `link`-Funktion ist eine bidirektionale Verbindung. Das bedeutet, dass wenn die Motorwerte an einer anderen Stelle gesetzt werden, sich die Schieberegler entsprechend auch anpassen. Dafür kann folgendes Codebeispiel ausgeführt werden:

```
[ ]: robot.forward(0.3)
time.sleep(1.0)
robot.stop()
```

Es ist zu erkennen, dass sich die Regler ohne Einwirken des Nutzers bewegen. Die Verbindung kann über die `unlink`-Methode wieder entfernt werden.

```
[ ]: left_link.unlink()
right_link.unlink()
```

Sollte keine *bidirektionale* Verbindung erwünscht sein, um z.B. die Motorwerte nur zu visualisieren, kann die `dlink`-Funktion genutzt werden. Die linke Eingabe ist die `Quelle` und der rechte Eingang das `Target`.

```
[ ]: left_link = traitlets.dlink((robot.left_motor, 'value'), (left_slider, 'value'))
right_link = traitlets.dlink((robot.right_motor, 'value'), (right_slider, □
    ↴'value'))
```

Werden nun die Schieberegler bewegt, ändern sich die Werte der Motoren nicht. Werden die Werte der Motoren jedoch an einer anderen Stelle angepasst, bewegen sich die Schieberegler entsprechend.

1.0.5 Funktionen an Ereignisse binden

Ein anderer Weg `traitlets` zu benutzen ist es Funktionen (wie `forward`) an Ereignisse zu binden. Diese Funktionen werden aufgerufen, sobald sich der Wert des Objekts ändert und erhalten Informationen über die Änderung wie den `old`- und `new`-Wert.

Beispielhaft werden einige Buttons erstellt, die den Roboter steuern sollen.

```
[ ]: # create buttons
button_layout = widgets.Layout(width='100px', height='80px', align_self='center')
stop_button = widgets.Button(description='stop', button_style='danger', layout=button_layout)
forward_button = widgets.Button(description='forward', layout=button_layout)
backward_button = widgets.Button(description='backward', layout=button_layout)
left_button = widgets.Button(description='left', layout=button_layout)
right_button = widgets.Button(description='right', layout=button_layout)

# display buttons
middle_box = widgets.HBox([left_button, stop_button, right_button], layout=widgets.Layout(align_self='center'))
controls_box = widgets.VBox([forward_button, middle_box, backward_button])
display(controls_box)
```

Auch hier müssen die Motoren zunächst verbunden werden. Dazu werden Funktionen erstellt, die mit den `on_click`-Ereignissen der Buttons verknüpft werden.

```
[ ]: def stop(change):
    robot.stop()

def step_forward(change):
    robot.forward(0.4)
    time.sleep(0.5)
    robot.stop()

def step_backward(change):
    robot.backward(0.4)
    time.sleep(0.5)
    robot.stop()

def step_left(change):
    robot.left(0.3)
    time.sleep(0.5)
    robot.stop()

def step_right(change):
    robot.right(0.3)
    time.sleep(0.5)
    robot.stop()
```

Nachdem die Funktionen definiert wurden, können diese nun mit den `on_click`-Ereignissen der Buttons verknüpft werden.

```
[ ]: # link buttons to actions
stop_button.on_click(stop)
```

```

forward_button.on_click(step_forward)
backward_button.on_click(step_backward)
left_button.on_click(step_left)
right_button.on_click(step_right)

```

Nun kann der Jetbot per Button-Klick gesteuert werden!

1.0.6 Heartbeat Killswitch

Das letzte Beispiel zeigt, wie ein ‘Heartbeat’ umgesetzt werden kann, um den Roboter zu stoppen. Dabei handelt es sich um einen leichten Weg um zu prüfen, ob die Verbindung zum Roboter noch aktiv ist. Über den Schieberegler kann die Zykluszeit des Heartbeats (in Sekunden) angepasst werden. Kann keine Kommunikation zwischen Jetbot und Computer innerhalb zwei Heartbeats abgeschlossen werden, wird das ‘status’-Attribut des Heartbeats auf `dead` gesetzt. In dem Moment wo die Verbindung wieder hergestellt wird, wechselt das `status`-Attribut wieder zu `alive`.

```

[ ]: from jetbot import Heartbeat

heartbeat = Heartbeat()

# this function will be called when heartbeat 'alive' status changes
def handle_heartbeat_status(change):
    if change['new'] == Heartbeat.Status.dead:
        robot.stop()

heartbeat.observe(handle_heartbeat_status, names='status')

period_slider = widgets.FloatSlider(description='period', min=0.001, max=0.5,
                                     step=0.01, value=0.5)
traitlets.dlink((period_slider, 'value'), (heartbeat, 'period'))

display(period_slider, heartbeat.pulseout)

```

Beispielhaft kann der unten stehende Codeschnipsel ausgeführt werden, während der Schieberegler nach unten geschoben wird.

```

[ ]: robot.left(0.2)

# now lower the `period` slider above until the network heartbeat can't be satisfied

```

datensammlung_kollisionsvermeidung

September 25, 2022

1 Kollisionsvermeidung - Datensammlung

Im nächsten Schritt soll es darum gehen den Jetbot von selbst fahren zu lassen.

Dabei handelt es sich um eine schwierige Aufgabe, die jedoch in kleinere Teilaufgaben unterteilt werden kann. Eine der wichtigsten Aufgaben bzw. Fähigkeiten, die der Roboter besitzen muss ist, dass er sich nicht selbst in gefährliche Situationen begeben kann. Diese Fähigkeit wird nachfolgend als *Kollisionsvermeidung* bezeichnet.

Die folgenden Notebooks beschäftigen sich mit der Umsetzung dieser Fähigkeit mithilfe von Neuronalen Netzen und der Kamera des Roboters als Sensor.

Der Ansatz ist dabei eine virtuelle “safety bubble” um den Roboter herum zu erstellen. Innerhalb dieser Blase kann dieser sich frei im Kreis drehen, ohne dass er mit Hindernissen kollidiert oder von einem Vorsprung herunterfällt.

Konkret würd dafür wie folgt vorgegangen:

Als erstes wird der Jetbot in Szenarien platziert, wo seine “safety bubble” verletzt wird. Dazu werden Bilder aufgenommen zusammen mit dem Label **blocked**.

Als zweites wird er in Szenarien platziert, wo er sich nach vorne bewegen könnte, ohne dass er mit etwas Kollidiert. Es werden ebenfalls Bilder aufgenommen, diesmal jedoch mit dem Label **free**.

Nachdem genügend gelabelte Bilder aufgenommen wurden (ca. 100 sollten fürs erste genügen), werden diese auf einen Computer mit einer schnellen GPU geladen, um dort das neuronale Netzwerk zu *trainieren*, so dass es anhand der aufgenommenen Bilder entscheiden kann, ob die safety bubble des Jetbots verletzt wurde. Damit wird es möglich sein am Ende die Fähigkeit der Kollisionsvermeidung zu implementieren.

1.0.1 Live-Bild der Kamera anzeigen

Zunächst wird die Kamera initialisiert und ihr Bild angezeigt.

Das neuronale Netzwerk erwartet Bilder mit der Auflösung von 224x224 Pixeln als Eingabe. Dafür wird die Auflösung der Kamera entsprechend dieser Anforderung gesetzt, was ebenfalls dafür sorgt, dass die Dateigröße minimiert wird. Es gibt grundsätzlich auch Szenarien, wo es von Vorteil wäre die volle Auflösung zu nutzen und erst zu einem späteren Zeitpunkt die Bilder zu komprimieren. Dies wird hier jedoch nicht behandelt.

```
[ ]: import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display
from jetbot import Camera, bgr8_to_jpeg

camera = Camera.instance(width=224, height=224)

image = widgets.Image(format='jpeg', width=224, height=224) # this width and
#height doesn't necessarily have to match the camera

camera_link = traitlets.dlink((camera, 'value'), (image, 'value'),
#transform=bgr8_to_jpeg)

display(image)
```

Als nächsten werden Verzeichnisse angelegt, in denen die Bild-Daten gespeichert werden. Es wird ein Verzeichnis `dataset` angelegt, welches zwei Unterordner `free` und `blocked` enthält. In diesen werden die Bilder für die jeweiligen Szenarien gespeichert.

```
[ ]: import os

blocked_dir = 'dataset/blocked'
free_dir = 'dataset/free'

# we have this "try/except" statement because these next functions can throw an
# error if the directories exist already
try:
    os.makedirs(free_dir)
    os.makedirs(blocked_dir)
except FileExistsError:
    print('Directories not created because they already exist')
```

Als nächstes werden Buttons angelegt, mit denen die Bilder mitsamt Label gespeichert werden können. Außerdem werden Textfelder angelegt, die anzeigen, wie viele Bilder für die jeweiligen Kategorien bereits gespeichert wurden. Dies ist hilfreich, um sicherzustellen, dass etwa gleich viele Bilder für die Kategorien `free` und `blocked` gespeichert wurden. Außerdem hilft es zu wissen, wie viele Bilder insgesamt gespeichert wurden.

```
[ ]: button_layout = widgets.Layout(width='128px', height='64px')
free_button = widgets.Button(description='add free', button_style='success',
#layout=button_layout)
blocked_button = widgets.Button(description='add blocked', #layout=button_layout)
#button_style='danger', layout=button_layout)
free_count = widgets.IntText(layout=button_layout, value=len(os.
#listdir(free_dir)))
blocked_count = widgets.IntText(layout=button_layout, value=len(os.
#listdir(blocked_dir)))
```

```
display(widgets.HBox([free_count, free_button]))
display(widgets.HBox([blocked_count, blocked_button]))
```

Aktuell haben die Buttons noch keine Funktion. Dafür müssen die Funktionen zum speichern der Bilder der jeweiligen Kategorie erst mit dem `on_click` Event der Buttons verbunden werden. Die Bilder werden dabei aus dem `Image` Widget gespeichert, da diese bereits im komprimierten JPEG Format vorliegen.

Um sicherzugehen, dass keine Dateinamen doppelt vergeben werden (auch nicht auf verschiedenen Computern), wird das `uuid` Paket in Python verwendet, welches die `uuid1` Methode definiert, um einen eindeutigen Identifikator zu generieren. Dieser eindeutige Identifikator wird aus Informationen wie der aktuellen Zeit und der Maschinenadresse generiert.

```
[ ]: from uuid import uuid1

def save_snapshot(directory):
    image_path = os.path.join(directory, str(uuid1()) + '.jpg')
    with open(image_path, 'wb') as f:
        f.write(image.value)

def save_free():
    global free_dir, free_count
    save_snapshot(free_dir)
    free_count.value = len(os.listdir(free_dir))

def save_blocked():
    global blocked_dir, blocked_count
    save_snapshot(blocked_dir)
    blocked_count.value = len(os.listdir(blocked_dir))

# attach the callbacks, we use a 'lambda' function to ignore the
# parameter that the on_click event would provide to our function
# because we don't need it.
free_button.on_click(lambda x: save_free())
blocked_button.on_click(lambda x: save_blocked())
```

Nun können die Buttons Bilder in den `free` und `blocked` Ordnern abspeichern.

Als nächstes kann mit dem Aufnehmen von Bildern begonnen werden:

1. Jetbot in einem Szenario platzieren, wo er blockiert ist und `add blocked` drücken
2. Roboter in einem Szenario platzieren, wo er frei ist und den Button `add free` betätigen
3. 1 und 2 wiederholen, bis genügend Bilder gespeichert wurden

Tipps für das Labeln der Daten:

1. Verschiedene Ausrichtungen des Roboters aufnehmen
2. Verschiedene Lichtverhältnisse aufnehmen
3. Verschiedene Typen von Kollisionen aufnehmen (z.B. Wände, Vorsprünge, Gegenstände)

4. Verschiedene Texturen aufnehmen (z.B. glatte Oberflächen, rauhe Oberflächen, etc.)

Grundsätzlich kann gesagt werden, um so mehr Daten aufgenommen wurden für verschiedene Szenarien, desto besser ist das kollisionsvermeidende Verhalten des Roboters. Dabei ist es vor allem wichtig eine *Varianz* in den Daten zu haben (wie oben beschrieben) und nicht nur eine große Menge an Bildern.

```
[ ]: display(image)
display(widgets.HBox([free_count, free_button]))
display(widgets.HBox([blocked_count, blocked_button]))
```

Abschließend muss die Kameraverbindung wieder geschlossen werden, damit diese später in einem anderen Notebook wieder genutzt werden kann.

```
[ ]: camera.stop()
```

1.1 Wie geht es weiter?

Wurden genügend Daten gesammelt, können diese auf einen Computer mit einer Leistungsstarken GPU kopiert werden. Dafür kann das folgende *Terminal* Kommando aufgerufen werden, um dataset Ordner in eine *zip*-Datei umzuwandeln.

Der ! Prefix gibt an, dass die Zelle als *shell* (oder *terminal*) Kommando ausgeführt werden soll.

Die -r Flag im zip Kommando indiziert *recursive*, dass alle untergeordneten Dateien mit berücksichtigt werden, die -q Flag indiziert *quiet*, damit das zip Kommando keine Ausgabe in das Terminal auslöst.

```
[ ]: !zip -r -q dataset.zip dataset
```

Die *dataset.zip* Datei kann nun heruntergeladen und auf den Zielrechner kopiert werden. You should download the zip file using the Jupyter Lab file browser by right clicking and selecting **Download**.

Im nächsten Schritt und gleichzeitig auch nächsten Notebook wird dann das neuronale Netz trainiert.

modell_training_kollisionsvermeidung

September 25, 2022

1 Kollisionsvermeidung - Modell-Training

In diesem Notebook wird ein Bildklassifikator darauf trainiert die zwei Klassen `free` und `blocket` zur Vermeidung von Kollisionen zu erkennen. Dazu wird die Deep-Learning-Bibliothek *PyTorch* verwendet.

```
[ ]: import torch
      import torch.optim as optim
      import torch.nn.functional as F
      import torchvision
      import torchvision.datasets as datasets
      import torchvision.models as models
      import torchvision.transforms as transforms
```

1.0.1 Datensatz hochladen und entpacken

Bevor begonnen werden kann wird die `dataset.zip` Datei aus dem `datensammlung.ipynb` Notebook vom Roboter hochgeladen.

Der folgende Befehl entpackt die Zip-Datei:

```
[ ]: !unzip -q dataset.zip
```

Es sollte nun ein `dataset` Ordner erscheinen.

1.0.2 Datensatz-Instanz erstellen

Als nächstes wird die `ImageFolder` Datensatz-Klasse aus dem `torchvision.datasets` Paket genutzt. Weiterhin werden sogenannte `transform` aus dem `torchvision.transforms` Paket verwendet, um die Daten für das Trainieren vorzubereiten.

```
[ ]: dataset = datasets.ImageFolder(
      'dataset',
      transforms.Compose([
          transforms.ColorJitter(0.1, 0.1, 0.1, 0.1),
          transforms.Resize((224, 224)),
          transforms.ToTensor(),
          transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
```

```
)
```

1.0.3 Aufteilen des Datensatzes in Trainings- und Testdaten

Als nächstes wird der Datensatz in *training* und *test* Sets unterteilt. Die Test-Sets werden genutzt um die Genauigkeit/Performance des antrainierten Modells zu verifizieren.

```
[ ]: train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset) - 50, 50])
```

1.0.4 Data Loaders zum Laden der Daten in Paketen

Es werden zwei `DataLoader` Instanzen erzeugt, welche Fähigkeiten zum durchmischen von Daten bereitstellen, sowie *batches* von Bildern erzeugen, die dann parallel an das Modell übergeben werden können mittels mehrere Worker.

```
[ ]: train_loader = torch.utils.data.DataLoader(  
    train_dataset,  
    batch_size=8,  
    shuffle=True,  
    num_workers=0  
)  
  
test_loader = torch.utils.data.DataLoader(  
    test_dataset,  
    batch_size=8,  
    shuffle=True,  
    num_workers=0  
)
```

1.0.5 Definieren des neuronalen Netzes

Nun wird das neuronale Netz deklariert, welches im Folgenden genutzt werden soll. Das *torchvision* Paket besitzt bereits eine sammlung von vortrainierten Modellen, die genutzt werden können.

Über das *Transfer Learning* kann ein vortrainiertes Modell weiterverwertet werden für eine neue Aufgabe, für weelche deutlich weniger Daten zur Verfügung stehen.

Wichtige Eigenschaften, die beim initialen Training des vortrainierten Modells erlernt wurden, können für die neue Aufgabe weiterverwertet werden. Als neuronales Netzwerk wird anschließend das `alexnet`-Modell genutzt werden.

```
[ ]: model = models.alexnet(pretrained=True)
```

Das `alexnet`-Modell wurde ursprünglich mit einem Datensatz aus 1000 Klassen-Labeln trainiert, der nun genutze Datensatz besitz jedoch nur zwei Label. Folglich wird die letzte Layer mit einer neuen, untrainierten Layer ersetzt, welche nur zwei Ausgaben besitzt.

```
[ ]: model.classifier[6] = torch.nn.Linear(model.classifier[6].in_features, 2)
```

Abschließend wird das Modell auf die GPU verschoben, um dort berechnet zu werden

cuda sind die Rechenkerne einer Nvidia GPU

```
[ ]: device = torch.device('cuda')
model = model.to(device)
```

1.0.6 Trainieren des neuronalen Netzes

Über den unter stehenden Code wird das neuronale Netz in 30 Epochen trainiert, wobei das beste Modell nach jeder Epoche gespeichert wird.

Eine Epoche ist ein Durchlauf durch den gesamten Trainingsdatensatz

```
[ ]: NUM_EPOCHS = 30
BEST_MODEL_PATH = 'best_model.pth'
best_accuracy = 0.0

optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

for epoch in range(NUM_EPOCHS):

    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.cross_entropy(outputs, labels)
        loss.backward()
        optimizer.step()

    test_error_count = 0.0
    for images, labels in iter(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        test_error_count += float(torch.sum(torch.abs(labels - outputs.
→argmax(1)))))

    test_accuracy = 1.0 - float(test_error_count) / float(len(test_dataset))
    print('%d: %f' % (epoch, test_accuracy))
    if test_accuracy > best_accuracy:
        torch.save(model.state_dict(), BEST_MODEL_PATH)
        best_accuracy = test_accuracy
```

Wurde der Vorgang komplett abgeschlossen, sollte eine `best_model.pth` Datei im Jupyter Lab Dateibrowser erscheinen, welche anschließend heruntergeladen werden kann.

live_einsatz_kollisionsvermeidung

September 25, 2022

1 Kollisionsvermeidung - Live Demo

In diesem Notebook wird das trainierte Model eingesetzt um festzustellen, ob der Roboter `free` oder `blocked` ist, um die Fähigkeit der Kollisionsvermeidung zu implementieren.

1.1 Laden des trainierten Modells

Die Datei `best_model.pth` muss nun wieder auf den Jetbot geladen und in den Ordner dieses Notebooks kopiert werden.

Der folgende Codeblock initialisiert das PyTorch Modell. Es fällt die Ähnlichkeit zum Training auf.

```
[ ]: import torch
      import torchvision

      model = torchvision.models.alexnet(pretrained=False)
      model.classifier[6] = torch.nn.Linear(model.classifier[6].in_features, 2)
```

Als nächstes werden die trainierten Weights aus der hochgeladenen `best_model.pth` Datei geladen.

```
[ ]: model.load_state_dict(torch.load('best_model.pth'))
```

Aktuell befinden sich diese noch auf dem Speicher der CPU. Um sie auf die GPU zu übertragen, muss der folgende Codeblock ausgeführt werden.

```
[ ]: device = torch.device('cuda')
      model = model.to(device)
```

1.1.1 Erstellen der Vorverarbeitungs-Funktion (Preprocessing)

Das Modell wurde nun geladen, jedoch existiert noch ein kleines Problem. Das Format des trainierten modells stimmt nicht *exakt* mit dem der Kamera überein. Um das zu korrigieren sind die folgenden *preprocessing* Bildverarbeitungsschritte nötig:

1. Konvertieren von BGR zu RGB
2. Konvertieren von HWC Layout zum CHW Layout
3. Normalisieren unter der Nutzung der selben Parameter wie im Training (die Kamera gibt Werte zwischen [0, 255] zurück, die geladenen Bilder haben jedoch einen Wertebereich von [0, 1]. Folglich muss um 255.0 skaliert werden)
4. Transferieren der Daten vom CPU Speicher (RAM) zum GPU Speicher (VRAM)

5. Paketgrößen hinzufügen (Batchgröße)

```
[ ]: import cv2
import numpy as np

mean = 255.0 * np.array([0.485, 0.456, 0.406])
stdev = 255.0 * np.array([0.229, 0.224, 0.225])

normalize = torchvision.transforms.Normalize(mean, stdev)

def preprocess(camera_value):
    global device, normalize
    x = camera_value
    x = cv2.cvtColor(x, cv2.COLOR_BGR2RGB)
    x = x.transpose((2, 0, 1))
    x = torch.from_numpy(x).float()
    x = normalize(x)
    x = x.to(device)
    x = x[None, ...]
    return x
```

Nun wurde die preprocessing-Funktion erstellt, die die oben genannten Schritte ausführt, um das Kamera-Format an das des trainierten Modells anzupassen.

Im nächsten Schritt soll das Kamerabild wieder angezeigt werden. Weiterhin wird ein Schieberegler erstellt, der anzeigen soll, wie hoch die Wahrscheinlichkeit ist, dass der Roboter `blocked` ist. Außerdem wirds ein Regler implementiert, über den die Geschwindigkeit des Jetbot eingestellt werden kann.

```
[ ]: import traitlets
from IPython.display import display
import ipywidgets.widgets as widgets
from jetbot import Camera, bgr8_to_jpeg

camera = Camera.instance(width=224, height=224)
image = widgets.Image(format='jpeg', width=224, height=224)
blocked_slider = widgets.FloatSlider(description='blocked', min=0.0, max=1.0, orientation='vertical')
speed_slider = widgets.FloatSlider(description='speed', min=0.0, max=0.5, value=0.0, step=0.01, orientation='horizontal')

camera_link = traitlets.dlink((camera, 'value'), (image, 'value'), transform=bgr8_to_jpeg)

display(widgets.VBox([widgets.HBox([image, blocked_slider]), speed_slider]))
```

Weiterhin muss wieder die `robot` Instanz erstellt werden, die für die Steuerung der Motoren benötigt wird.

```
[ ]: from jetbot import Robot
      robot = Robot()
```

Darauf wird die Funktion implementiert, die immer dann aufgerufen wird, wenn eine Änderung des Kamerawertes (Kamerabildes) stattfindet. Diese Funktion schließt folgende Schritte ein:

1. Pre-process des Kamerabildes
2. Aufrufen/Ausführen des neuronalen Netzes
3. Wenn das neuronale netzt `blocked` als Ergebnis liefert, dann soll der Roboter sich nach links drehen, ansonsten soll er geradeaus fahren.

```
[ ]: import torch.nn.functional as F
      import time

      def update(change):
          global blocked_slider, robot
          x = change['new']
          x = preprocess(x)
          y = model(x)

          # we apply the `softmax` function to normalize the output vector so it sums to 1 (which makes it a probability distribution)
          y = F.softmax(y, dim=1)

          prob_blocked = float(y.flatten()[0])

          blocked_slider.value = prob_blocked

          if prob_blocked < 0.5:
              robot.forward(speed_slider.value)
          else:
              robot.left(speed_slider.value)

          time.sleep(0.001)

      update({'new': camera.value}) # we call the function once to initialize
```

Nachdem die Ausführ-Funktion erstellt wurde, muss diese nun an die Kamera verknüpft werden, um die Verarbeitung zu ermöglichen.

Dies kann über die `observe` Funktion erreicht werden.

ACHTUNG: der Jetbot wird sich nun von alleine bewegen!

```
[ ]: camera.observe(update, names='value') # this attaches the 'update' function to the 'value' traitlet of our camera
```

Soll der Roboter wieder gestoppt werden, kann dies über die `unobserve` Funktion erreicht werden.

```
[ ]: import time  
  
camera.unobserve(update, names='value')  
  
time.sleep(0.1) # add a small sleep to make sure frames have finished  
    ↵processing  
  
robot.stop()
```

Falls das Kamerabild nicht dauerhaft im Browser angezeigt werden soll:

```
[ ]: camera_link.unlink() # don't stream to browser (will still run camera)
```

Um das Kamerabild wieder im Browser anzuzeigen:

```
[ ]: camera_link.link() # stream to browser (wont run camera)
```

Am Ende der Ausführung sollte die Kameraverbindung wieder getrennt werden, damit diese erneut in einem anderen Notebook verwendet werden kann.

```
[ ]: camera.stop()
```

datensammlung_spurverfolgung

September 25, 2022

1 Spurverfolgung - Datensammlung

Auch hier werden wieder die folgenden Schritte durchgeführt:

1. Datensammlung
2. Training
3. Live-Einsatz

Im Unterschied zu der vorherigen Fähigkeit wird hier nicht mehr die Klassifizierung eingesetzt, sondern die **Regression**. Diese wird verwendet um dem Jetbot zu ermöglichen eine Spur zu verfolgen (bzw. einen beliebigen Pfad oder Zielpunkt).

1. Sammeln von Daten für verschiedene Positionen des Jetbots auf dem Pfad (verschiedene Abstände zum Mittelpunkt, verschiedene Winkel, etc.)

Wie auch schon bei der Kollisionsvermeidung gilt: Datenvarianz ist entscheidend!

2. Live-Bild der Kamera anzeigen
3. ‘Grünen Punkt’, welcher der Zielrichtung des Roboters entspricht, auf dem Bild platzieren
4. X und Y Werte des grünen Punktes zusammen mit dem Bild der Kamera des Roboters speichern

Dannach wird im Trainings-Notebook ein neuronales Netzwerk trainiert, welches die X und Y Werte des Labels vorhersagen kann. Im Live-Demo-Notebook werden die X und Y Werte verwendet um einen ungefähren Steuerwert zu berechnen.

Wie wird der Zielpunkt richtig platziert?

1. Auf das Livebild der Kamera gucken
2. Überlegen welchen Pfad der Roboter auf dem Bild fahren sollte (Entfernung abschätzen, nach der der Roboter von der Straße abkommen würde)
3. Zielpunkt so weit wie möglich in die Ferne platzieren, so dass der Roboter möglichst lange geradeaus fahren kann, bevor er von der Straße abkommt.

Beispiel: Bleibt die Strecke lange gerade, kann der Zielpunkt weit in die Ferne gesetzt werden. Liegt eine scharfe Kurve bevor, muss der Zielpunkt nah am Roboter platziert werden, da er sonst nicht mehr die Spur halten würde.

Tut das neuronale Netz das was ihm angelernt wurde, versichern die Label-Richtlinien folgendes:

1. Der Roboter kann sicher in Richtung eines Ziels fahren (ohne dabei von der Straße abzukommen)
2. Der Zielpunkt bewegt sich immer weiter entlang der zu fahrenden Strecke

Man kann sich das Ergebnis so ähnlich vorstellen wie das Reiten eines Esels mit einer Karotte an einer Angel. Der wesentliche Unterschied ist, dass das neuronale Netz vorgibt, wo sich die Karotte befindet.

1.0.1 Einbinden der Bibliotheken

Von entscheidender Rolle ist hier die Bibliothek OpenCV mit der die Kamerabilder sowohl dargestellt als auch mit Labels angespeichert werden können. Bibliotheken wie uuid und datetime werden für die Namensgebung der Bilddateien verwendet.

```
[ ]: # IPython Libraries for display and widgets
import ipywidgets
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display

# Camera and Motor Interface for JetBot
from jetbot import Robot, Camera, bgr8_to_jpeg

# Basic Python packages for image annotation
from uuid import uuid1
import os
import json
import glob
import datetime
import numpy as np
import cv2
import time
```

1.0.2 Datensammlung

Erneut soll das Kamerabild wieder angezeigt werden. Diesmal wird jedoch ein spezielles ipywidget verwendet (`jupyter_clickable_image_widget`), welches es ermöglicht auf das Bild zu klicken und die Koordinaten des Labels zu speichern.

Es wird die Kamera Klasse des Jetbot verwendet, um die CSI MIPI Kamera zu aktivieren. Das neuronale Netzwerk nimmt ein Bild mit den Maßen 224x224 Pixel als Eingabe entgegen. Die Kamera wird auf diese Größe gesetzt, um die Dateigröße des Datensatzes zu minimieren. In einigen Szenarien kann es besser sein, Daten in einer größeren Bildgröße zu sammeln und später auf die gewünschte Größe zu skalieren.

Der folgende Codeblock zeigt das Livebild der Kamera auf welches geklickt werden kann, um ein Label zu platzieren. Daneben wird das letzte Bild angezeigt, auf welchem ein grüner Kreis zu sehen ist, wo zuvor geklickt wurde. Darunter wird die Anzahl der gespeicherten Bilder angezeigt.

Wird links auf das Livebild geklickt, dann wird ein Bild inklusive Label im `dataset_xy` abgespeichert mit dem Namen

`xy_<x value>_<y value>_<uuid>.jpg`

Beim Trainieren werden die Bilder geladen und die X und Y Werte aus dem Dateinamen extrahiert.

Dabei sind <x value> und <y value> die Koordinaten **in Pixeln** (gezählt von der linken oberen Ecke).

```
[ ]: from jupyter_clickable_image_widget import ClickableImageWidget

DATASET_DIR = 'dataset_xy'

# we have this "try/except" statement because these next functions can throw an
error if the directories exist already

try:
    os.makedirs(DATASET_DIR)
except FileExistsError:
    print('Directories not created because they already exist')

camera = Camera()

# create image preview
camera_widget = ClickableImageWidget(width=camera.width, height=camera.height)
snapshot_widget = ipywidgets.Image(width=camera.width, height=camera.height)
traitlets.dlink((camera, 'value'), (camera_widget, 'value'),
transform=bgr8_to_jpeg)

# create widgets
count_widget = ipywidgets.IntText(description='count')
# manually update counts at initialization
count_widget.value = len(glob.glob(os.path.join(DATASET_DIR, '*.jpg')))

def save_snapshot(_, content, msg):
    if content['event'] == 'click':
        data = content['eventData']
        x = data['offsetX']
        y = data['offsetY']

        # save to disk
#dataset.save_entry(category_widget.value, camera.value, x, y)
        uuid = 'xy_%03d_%03d_%s' % (x, y, uuid1())
        image_path = os.path.join(DATASET_DIR, uuid + '.jpg')
        with open(image_path, 'wb') as f:
            f.write(camera_widget.value)

        # display saved snapshot
        snapshot = camera.value.copy()
        snapshot = cv2.circle(snapshot, (x, y), 8, (0, 255, 0), 3)
        snapshot_widget.value = bgr8_to_jpeg(snapshot)
        count_widget.value = len(glob.glob(os.path.join(DATASET_DIR, '*.jpg')))

camera_widget.on_msg(save_snapshot)
```

```

data_collection_widget = ipywidgets.VBox([
    ipywidgets.HBox([camera_widget, snapshot_widget]),
    count_widget
])

display(data_collection_widget)

```

Wie bereits in anderen Notebooks muss am Ende wieder die Kamera freigegeben werden (Verbindung trennen).

[]: camera.stop()

1.0.3 Next

Auch hier gilt wieder: Wurden genügend Daten gesammelt, so können diese wieder zum Trainieren auf einen Leistungsstarken Computer übertragen werden.

Sollte der Wunsch bestehen das Trainieren auf dem Jetbot selbst durchzuführen, so kann dieser Schritt übersprungen werden. Es sollte sich jedoch auf eine lange Berechnungszeit eingestellt werden.

Der folgende Codeblock komprimiert die Bilddaten wieder in eine zip Datei, welche dann auf einen Leistungsstarken Computer übertragen werden kann.

[]:

```

def timestr():
    return str(datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S'))

!zip -r -q road_following_{DATASET_DIR}_{timestr()}.zip {DATASET_DIR}
```

Es sollte nun eine Datei angelegt worden sein mit dem Namen road_following_<Date&Time>.zip. Diese kann heruntergeladen werden.

modell_training_spurverfolgung

September 25, 2022

1 Spurverfolgung - Modell-Training

In diesem Notebook wird ein neuronales Netzwerk trainiert ein Eingabebild entgegenzunehmen und eine Menge von x, y-Werten wieder auszugeben, die einer Zielposition (des Roboters) entsprechen.

Dafür wird das PyTorch Deep Learning Framework verwendet, um ein ResNet18-Modell für die Spurverfolgung zu trainieren.

```
[ ]: import torch
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision.transforms as transforms
import glob
import PIL.Image
import os
import numpy as np
```

1.0.1 Downloaden und Extrahieren des Datensatzes

Die `road_following_<Date&Time>.zip` muss zunächst entpackt werden.

Dazu dient folgender Befehl:

```
[ ]: !unzip -q road_following.zip
```

Es sollte nun ein Ordner mit dem Namen `dataset_all` erstellt worden sein.

1.0.2 Datensatz-Instanz erstellen

Hier wird eine eigene `torch.utils.data.Dataset` Implementation erstellt, welche die `__len__` und `__getitem__` Funktionen implementiert. Diese Klasse ist dafür verantwortlich Bilder zu laden und die x, y-Werte aus den Bilddateinamen zu parsen. Da die Klasse `torch.utils.data.Dataset` implementiert wurde, können alle torch-Daten-Utilities verwendet werden.

Außerdem wurden einige Transformationen hard decoded (wie z.B. ein Farb-Jitter) des Datensatzes. Außerdem wurden zufällige horizontale Spiegelungen der Bilder durchgeführt, falls es sich bei der

Strecke um einen nicht-symmetrischen Pfad handelt (z.B. eine Straße, auf der der Roboter auf der rechten Seite fahren muss).

```
[ ]: def get_x(path, width):
    """Gets the x value from the image filename"""
    return (float(int(path.split("_")[1])) - width/2) / (width/2)

def get_y(path, height):
    """Gets the y value from the image filename"""
    return (float(int(path.split("_")[2])) - height/2) / (height/2)

class XYDataset(torch.utils.data.Dataset):

    def __init__(self, directory, random_hflips=False):
        self.directory = directory
        self.random_hflips = random_hflips
        self.image_paths = glob.glob(os.path.join(self.directory, '*.jpg'))
        self.color_jitter = transforms.ColorJitter(0.3, 0.3, 0.3, 0.3)

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]

        image = PIL.Image.open(image_path)
        width, height = image.size
        x = float(get_x(os.path.basename(image_path), width))
        y = float(get_y(os.path.basename(image_path), height))

        if float(np.random.rand(1)) > 0.5:
            image = transforms.functional.hflip(image)
            x = -x

        image = self.color_jitter(image)
        image = transforms.functional.resize(image, (224, 224))
        image = transforms.functional.to_tensor(image)
        image = image.numpy()[:, :, ::-1].copy()
        image = torch.from_numpy(image)
        image = transforms.functional.normalize(image, [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

    return image, torch.tensor([x, y]).float()

dataset = XYDataset('dataset_xy', random_hflips=False)
```

1.0.3 Aufteilen des Datensatzes in Trainings- und Validierungsdatensatz

Nachdem der Datensatz eingelesen wurde wird er auch hier in einen Trainings- und Validierungsdatensatz aufgeteilt. In diesem Beispiel wird der Datensatz in 90%-10% aufgeteilt. Der Validierungsdatensatz wird verwendet um die Genauigkeit des trainierten Modells zu überprüfen.

```
[ ]: test_percent = 0.1
num_test = int(test_percent * len(dataset))
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [len(dataset) - num_test, num_test])
```

1.0.4 Erstellen von DataLoader-Instanzen zum Laden von Daten in Batches

Es wird ein `DataLoader` erstellt, um die Daten in Batches zu laden, diese zu mischen und das Laden in mehreren (Unter-) Prozessen zu ermöglichen. In diesem Beispiel wird eine Batch-Größe von 64 verwendet. Die Batch-Größe sollte auf die verfügbare GPU-Speichergröße abgestimmt sein, da sie die Genauigkeit des Modells beeinflusst.

```
[ ]: train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=8,
    shuffle=True,
    num_workers=0
)

test_loader = torch.utils.data.DataLoader(
    test_dataset,
    batch_size=8,
    shuffle=True,
    num_workers=0
)
```

1.0.5 Deklarieren des neuronalen Netzes

Es wird das ResNet-18 Modell aus PyTorch TorchVision verwendet.

Auch hier findet das Transfer Learning wieder Anwendung.

```
[ ]: model = models.resnet18(pretrained=True)
```

Das ResNet Modell hat eine fully connected (fc) final-Layer mit 512 `in_features` und nach dem Angewendeten Training (unter Nutzung von Regression) mit einem `out_features`.

Abschließend wir das Modell zur Berechnung auf die GPU verschoben.

```
[ ]: model.fc = torch.nn.Linear(512, 2)
device = torch.device('cuda')
model = model.to(device)
```

1.0.6 Regressionstraining

Es wird für 50 Epochen trainiert und das beste Modell wird abgespeichert, falls der Verlust im vergleich zur vorherigen Epoche verringert werden konnte.

```
[ ]: NUM_EPOCHS = 70
BEST_MODEL_PATH = 'best_steering_model_xy.pth'
best_loss = 1e9

optimizer = optim.Adam(model.parameters())

for epoch in range(NUM_EPOCHS):

    model.train()
    train_loss = 0.0
    for images, labels in iter(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        train_loss += float(loss)
        loss.backward()
        optimizer.step()
    train_loss /= len(train_loader)

    model.eval()
    test_loss = 0.0
    for images, labels in iter(test_loader):
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        loss = F.mse_loss(outputs, labels)
        test_loss += float(loss)
    test_loss /= len(test_loader)

    print('%f, %f' % (train_loss, test_loss))
    if test_loss < best_loss:
        torch.save(model.state_dict(), BEST_MODEL_PATH)
        best_loss = test_loss
```

Ist das Modell fertig trainiert, so wird es in der `best_steering_model_xy.pth` Datei abgelegt, welche wieder zurück auf den Jetbot geladen werden kann.

live_einsatz_spurverfolgung

September 25, 2022

1 Spurverfolgung - Live Demo

In diesem Notebook wird das trainierte Modell genutzt um den Jetbot auf einer (Klemmbaustein-) Straße fahren zu lassen.

1.0.1 Laden des trainierten Modells

Nun muss die `best_steering_model_xy.pth` Datei wieder auf den Jetbot hochgeladen werden in den Ordner dieses Notebooks.

Der folgende Code initialisiert das PyTorch Modell.

```
[ ]: import torchvision
      import torch

      model = torchvision.models.resnet18(pretrained=False)
      model.fc = torch.nn.Linear(512, 2)
```

Als nächstes werden die trainierten Weights aus der `best_steering_model_xy.pth` Datei hochgeladen.

```
[ ]: model.load_state_dict(torch.load('best_steering_model_xy.pth'))
```

Aktuell sind die hochgeladenen Weights noch im Speicher der CPU hinterlegt. Diese müssen zunächst in den VRAM der GPU transferiert werden.

```
[ ]: device = torch.device('cuda')
      model = model.to(device)
      model = model.eval().half()
```

1.0.2 Erstellen der Proprocessing Funktion

Wie bereits bei der Kollisionsvermeidung muss auch hier ein Proprocessing der Bilddaten stattfinden. Dazu werden die folgenden Schritte ausgeführt:

1. Kovertieren vom HWC Layout zum CHW Layout
2. Normalisieren unter der Nutzung der selben Parameter wie im Training (die Kamera gibt Werte zwischen [0, 255] zurück, die geladenen Bilder haben jedoch einen Wertebereich von [0, 1]. Folglich muss um 255.0 skaliert werden)
3. Transferieren der Daten vom CPU Speicher (RAM) zum GPU Speicher (VRAM)

4. Paketgrößen hinzufügen (Batchgröße)

```
[ ]: import torchvision.transforms as transforms
import torch.nn.functional as F
import cv2
import PIL.Image
import numpy as np

mean = torch.Tensor([0.485, 0.456, 0.406]).cuda().half()
std = torch.Tensor([0.229, 0.224, 0.225]).cuda().half()

def preprocess(image):
    image = PIL.Image.fromarray(image)
    image = transforms.functional.to_tensor(image).to(device).half()
    image.sub_(mean[:, None, None]).div_(std[:, None, None])
    return image[None, ...]
```

Nun wurde die preprocessing-Funktion erstellt, die die oben genannten Schritte ausführt, um das Kamera-Format an das des trainierten Modells anzupassen.

Im nächsten Schritt soll das Kamerabild wieder angezeigt werden.

```
[ ]: from IPython.display import display
import ipywidgets
import traitlets
from jetbot import Camera, bgr8_to_jpeg

camera = Camera()

image_widget = ipywidgets.Image()

traitlets.dlink((camera, 'value'), (image_widget, 'value'),  
    transform=bgr8_to_jpeg)

display(image_widget)
```

Weiterhin muss wieder die `robot` Instanz erstellt werden, die für die Steuerung der Motoren benötigt wird.

```
[ ]: from jetbot import Robot

robot = Robot()
```

Als nächstes werden Schieberegler, über die der Roboter konfiguriert werden kann, implementiert.
> Anmerkung: Die Regler wurden nach bestem Wissen und Wissen vorkonfiguriert. Konkrete Einstellungen hängen in der Praxis jedoch stark von der Umgebung und den Trainingsdaten ab.

1. Geschwindigkeitseinstellung (`speed_gain_slider`): Damit der Jetbot losfährt muss der `speed_gain_slider` erhöht werden

2. Lenk-Verstärkung (steering_gain_slider): Falls der Jetbot hin- und her wackeln sollte muss der `steering_gain_slider` verringert werden, bis dieser sich flüssig fortbewegt
3. Vorsteuerung der Lenkung (steering_bias_slider): Sollte sich der Jetbot permanent zu sehr in eine Richtung drehen, kann der `steering_bias_slider` verwendet werden um die Lenkung zu korrigieren (das gilt sowohl für eine Korrektur der Motoren als auch der Kameraposition)

Anmerkung: Es ist hilfreich bei einer niedrigen Geschwindigkeit mit den Reglern "rumzuspielen" um die optimalen Einstellungen zu finden.

```
[ ]: speed_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, description='speed gain')
steering_gain_slider = ipywidgets.FloatSlider(min=0.0, max=1.0, step=0.01, value=0.2, description='steering gain')
steering_dgain_slider = ipywidgets.FloatSlider(min=0.0, max=0.5, step=0.001, value=0.0, description='steering kd')
steering_bias_slider = ipywidgets.FloatSlider(min=-0.3, max=0.3, step=0.01, value=0.0, description='steering bias')

display(speed_gain_slider, steering_gain_slider, steering_dgain_slider, steering_bias_slider)
```

Als nächstes werden Schieberegler angelegt, die die Vorhersagen des Modells bezüglich des x- und y-Wertes anzeigen sollen.

Der Link-Schieberegler zeigt den geschätzten Lenkwert an.

```
[ ]: x_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='x')
y_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical', description='y')
steering_slider = ipywidgets.FloatSlider(min=-1.0, max=1.0, description='steering')
speed_slider = ipywidgets.FloatSlider(min=0, max=1.0, orientation='vertical', description='speed')

display(ipywidgets.HBox([y_slider, speed_slider]))
display(x_slider, steering_slider)
```

Darauf wird die Funktion implementiert, die immer dann aufgerufen wird, wenn eine Änderung des Kamerawertes (Kamerabildes) stattfindet. Diese Funktion schließt folgende Schritte ein:

1. Pre-process des Kamerabildes
2. Aufrufen/Ausführen des neuronalen Netzes
3. Berechnung der ungefähren Lenkwerte
4. Steuerung der Motoren über PD-Regler

```
[ ]: angle = 0.0
angle_last = 0.0

def execute(change):
```

```

global angle, angle_last
image = change['new']
xy = model(preprocess(image)).detach().float().cpu().numpy().flatten()
x = xy[0]
y = (0.5 - xy[1]) / 2.0

x_slider.value = x
y_slider.value = y

speed_slider.value = speed_gain_slider.value

angle = np.arctan2(x, y)
pid = angle * steering_gain_slider.value + (angle - angle_last) *_
steering_dgain_slider.value
angle_last = angle

steering_slider.value = pid + steering_bias_slider.value

robot.left_motor.value = max(min(speed_slider.value + steering_slider._.
value, 1.0), 0.0)
robot.right_motor.value = max(min(speed_slider.value - steering_slider._.
value, 1.0), 0.0)

execute({'new': camera.value})

```

Nachdem die Ausführ-Funktion erstellt wurde, muss diese nun an die Kamera verknüpft werden, um die Verarbeitung zu ermöglichen.

Dies kann über die `observe` Funktion erreicht werden.

ACHTUNG: der Jetbot wird sich nun von alleine bewegen!

```
[ ]: camera.observe(execute, names='value')
```

Soll der Roboter wieder gestoppt werden, kann dies über die `unobserve` Funktion erreicht werden im unten stehenden Codeblock.

```
[ ]: import time

camera.unobserve(execute, names='value')

time.sleep(0.1) # add a small sleep to make sure frames have finished_
processing

robot.stop()
```

Als letztes wird wieder die Verbindung zur Kamera getrennt

```
[ ]: camera.stop()
```

Literaturverzeichnis

Bücher

- [1] T. Amaratunga, *Deep Learning on Windows*. Apress, Dez. 2020, Kapitel Transfer Learning, 338 Seiten, ISBN: 978-1-4842-6431-7. Adresse: https://www.ebook.de/de/product/41243234/thimira_amaratunga_deep_learning_on_windows.html.
- [2] P. Kalaiarasi und P. E. Rani, *Advances in Smart System Technologies*. Springer-Verlag GmbH, Aug. 2020, Kapitel A Comparative Analysis of AlexNet and GoogLeNet with a Simple DCNN for Face Recognition, 836 Seiten, ISBN: 978-981-15-5029-4. Adresse: https://www.ebook.de/de/product/39551333/advances_in_smart_system_technologies.html (siehe Seite 10).
- [3] K. B. Prakash, Y. V. R. Nagapawan und G. R. Kanagachidambaresan, *Programming with TensorFlow*. Springer International Publishing, Jan. 2021, Kapitel Convolutional Neural Networks, 190 Seiten, Convolutional Neural Networks, ISBN: 978-3-030-57077-4. Adresse: https://www.ebook.de/de/product/41247357/programming_with_tensorflow.html (siehe Seite 4).
- [4] Z. A. Styczynski, K. Rudion und A. Naumann, *Einführung in Expertensysteme*. Springer-Verlag GmbH, Mai 2017, 250 Seiten, ISBN: 9783662531723. Adresse: https://www.ebook.de/de/product/29262042/zbigniew_a_styczynski_krzysztof_rudion_andre_naumann_einfuehrung_in_expertensysteme.html (siehe Seite 4).

Artikel

- [5] P. D.-I. S. Borchers-Tigasson, „Summary and Remarks on CNN,“ *Computational Intelligence*, 2022 (siehe Seite 6).
- [6] U. Schäffer und J. Weber, „Künstliche Intelligenz,“ *Controlling Management Review*, Jahrgang 65, Nummer 2, Seiten 3–3, Feb. 2021. DOI: [10.1007/s12176-021-0370-0](https://doi.org/10.1007/s12176-021-0370-0) (siehe Seite 4).

Online Quellen

- [7] J. Brownlee. „A Gentle Introduction to Transfer Learning for Deep Learning.“ (Dez. 2017), Adresse: <https://machinelearningmastery.com/transfer-learning-for-deep-learning/> (besucht am 24.09.2022) (siehe Seite 24).
- [8] A. Regensburg. „Hochschullogo HTW Berlin.“ (Juli 2018), Adresse: https://www.akademieregensburg.de/files/akademie/img/hochschulen_logos/18.jpg (besucht am 25.08.2022).

- [9] S. Sonwane. „Transfer Learning From Pre-Trained Model for Image Recognition.“ (Mai 2020), Adresse: <https://sagarsonwane230797.medium.com/transfer-learning-from-pre-trained-model-for-image-facial-recognition-8b0c2038d5f0> (besucht am 19.09.2022) (siehe Seite 12).
- [10] S. Vasudevan. „10. AlexNet - CNN Explained and Implemented.“ (Apr. 2020), Adresse: <https://www.youtube.com/watch?v=8GheVe2UmUM> (besucht am 19.09.2022) (siehe Seite 11).
- [11] Wikimedia. „Logo HTW Berlin.“ (Okt. 2014), Adresse: https://upload.wikimedia.org/wikipedia/commons/7/7e/LogoHTW_Berlin.svg (besucht am 25.08.2022).
- [12] Wikipedia. „AlexNet.“ (Sep. 2022), Adresse: <https://en.wikipedia.org/wiki/AlexNet> (besucht am 19.09.2022) (siehe Seiten 10, 11).
- [13] F. Woelki. „Was ist das Transfer Learning? | Künstliche Intelligenz.“ (Feb. 2020), Adresse: https://www.youtube.com/watch?v=K_csnXsNN5Q (besucht am 19.09.2022) (siehe Seite 12).
- [14] L. Wuttke. „Transfer Learning: Grundlagen und Einsatzgebiete.“ (), Adresse: <https://datasolut.com/was-ist-transfer-learning/> (besucht am 19.09.2022).
- [15] U. Würtz. „CNNs Convolutional Neural Networks Basiswissen.“ (Dez. 2019), Adresse: <https://www.youtube.com/watch?v=OV0KXyYpEZY> (besucht am 19.09.2022) (siehe Seiten 5–10).