



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**

<b>1 Einleitung</b>	<b>4</b>
<b>2 Theoretische Grundlagen</b>	<b>5</b>
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
<b>3 Versuchsaufbau und Inbetriebnahme</b>	<b>13</b>
3.1 Benötigte Komponenten . . . . .	13
3.2 Hardware-Setup . . . . .	15
3.3 Software-Setup . . . . .	16
<b>4 Umsetzung der Kollisionsvermeidung</b>	<b>19</b>
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
<b>5 Ergebnisse und Fazit</b>	<b>24</b>
<b>6 Ausblick - Autonomes Fahren</b>	<b>25</b>
<b>7 Anhang - Jupyter Notebooks</b>	<b>26</b>
<b>Literaturverzeichnis</b>	<b>32</b>
Bücher . . . . .	32
Artikel . . . . .	32
Online Quellen . . . . .	32

## 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].

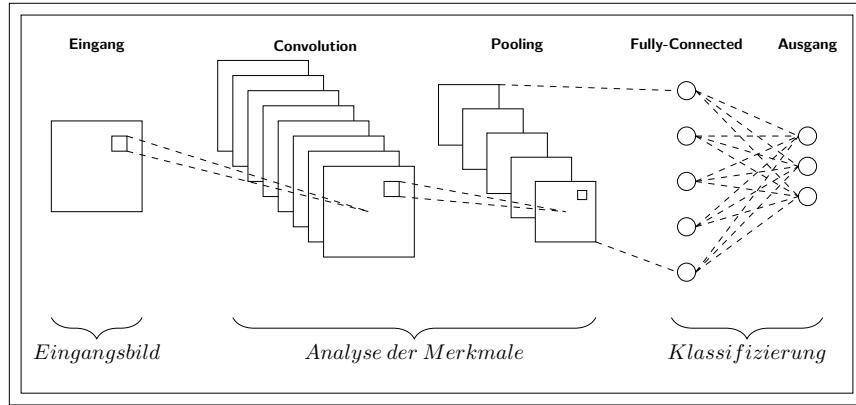


Abb. 2.1: Basisstruktur eines CNN [5]

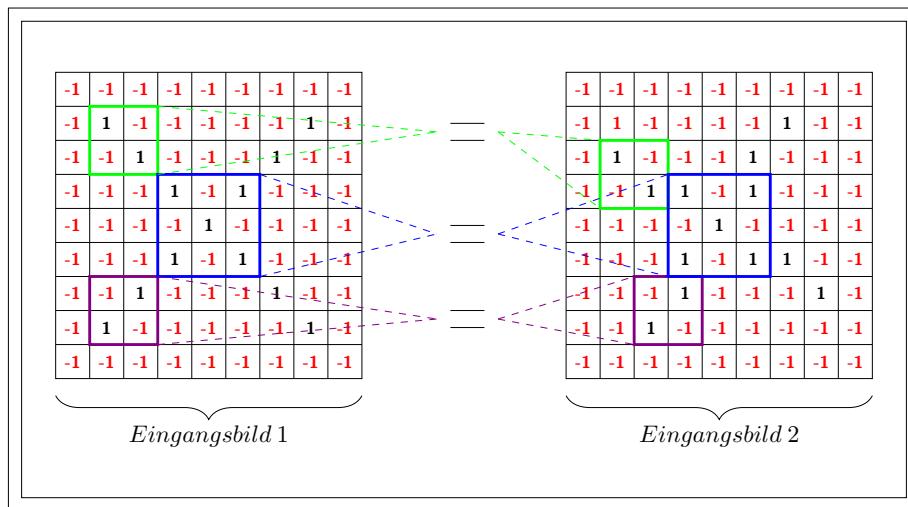


Abb. 2.2: Unterschiedliche Eingangsbilder mit ähnlichen Merkmalen [15]

### 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:

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

$$x \approx -0.11$$

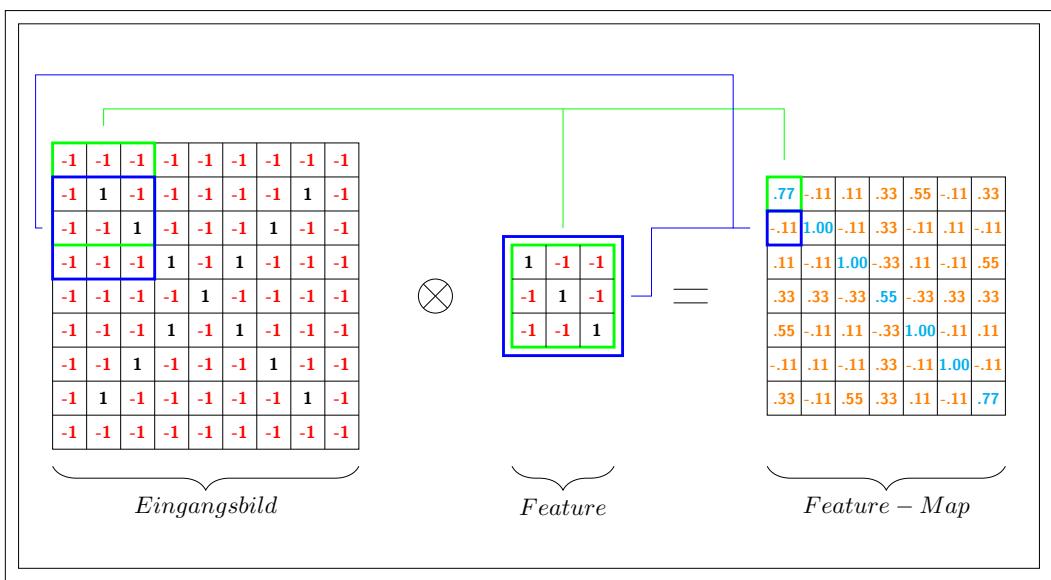


Abb. 2.3: Anwendung der Faltung mithilfe eines Filters [15]

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).

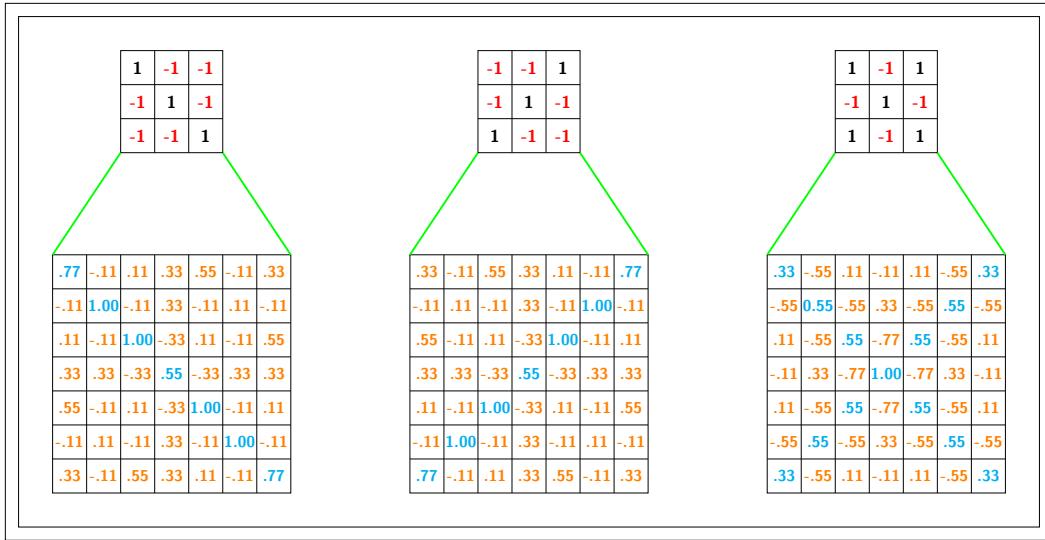


Abb. 2.4: Filtertypen und zugehörige Feature-Maps [15]

### 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.

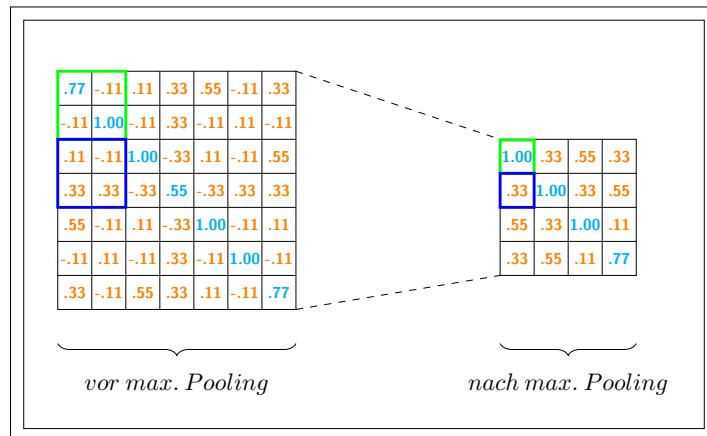


Abb. 2.5: Anwendung der Zusammenlegung (max. Pooling) [15]

### 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].

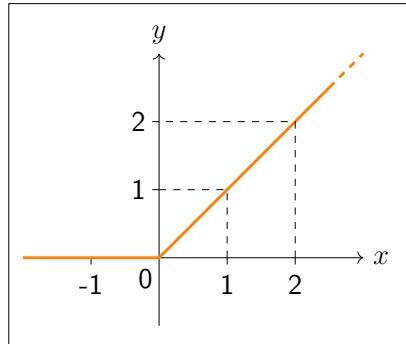


Abb. 2.6: Rectified Linear Unit [15]

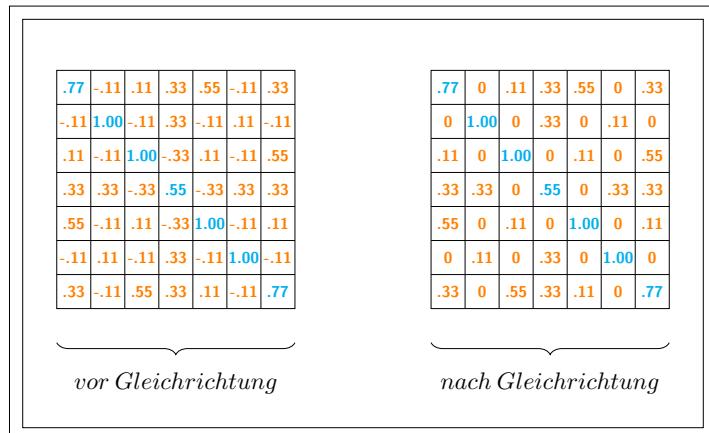


Abb. 2.7: Anwendung der Gleichrichtung (Rectification) [15]

### 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“:

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

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

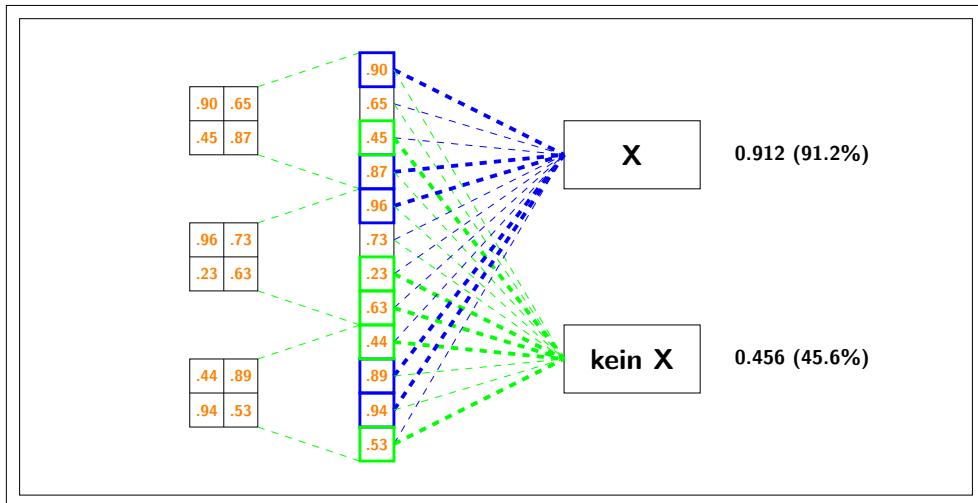


Abb. 2.8: Gewichtungen im Fully-Connected-Layer [15]

## 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.

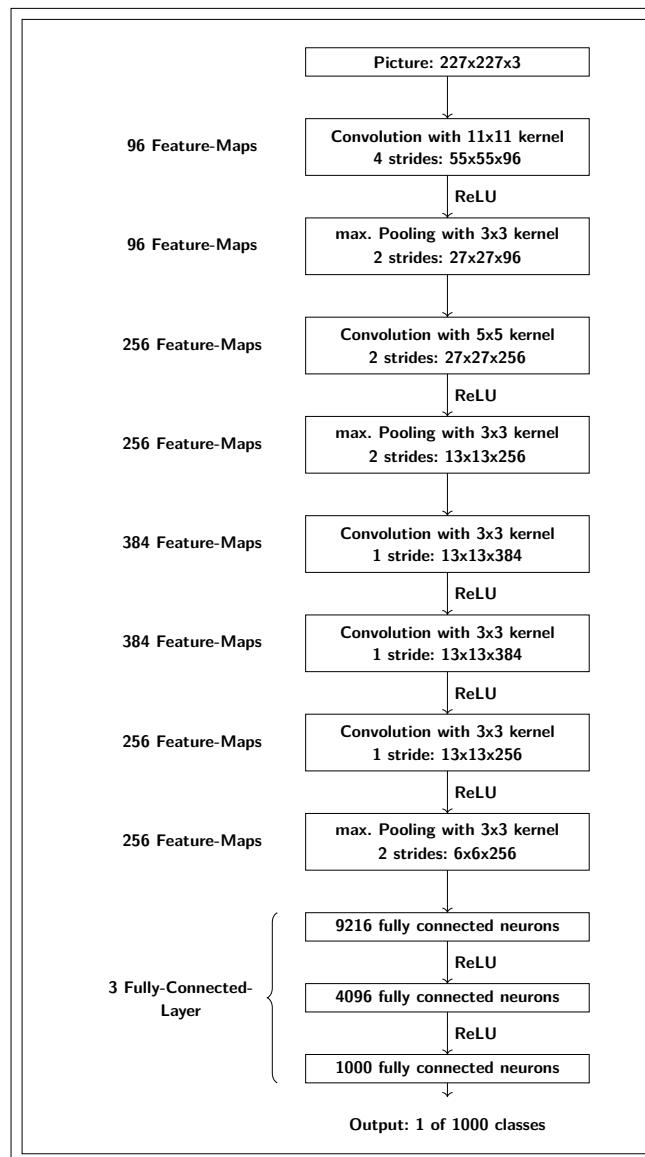


Abb. 2.9: Schematischer Aufbau des AlexNet [12]

## 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].

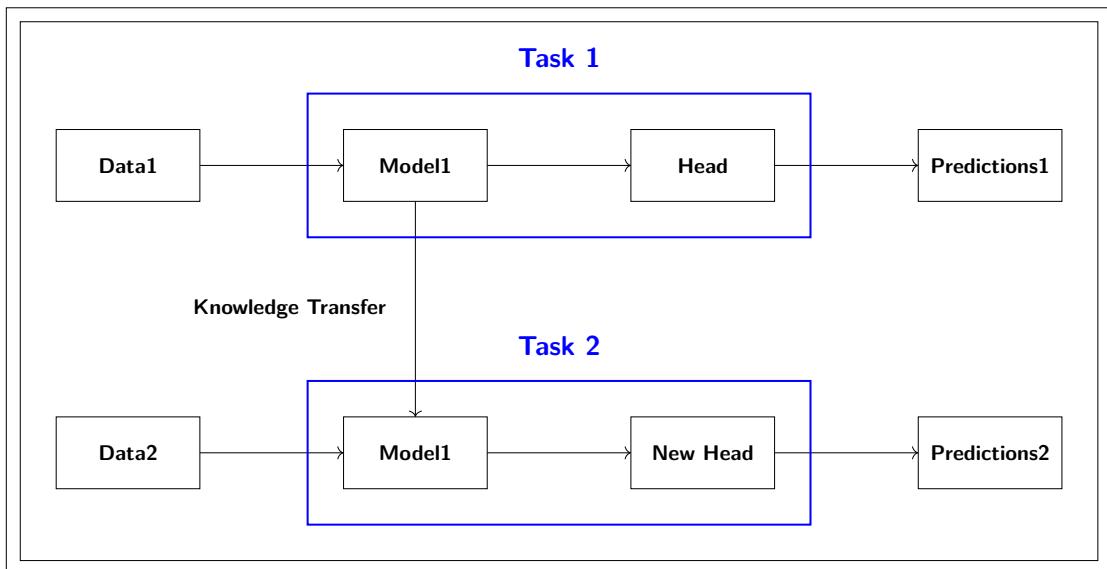


Abb. 2.10: Visualisierung des Transfer-Learning [9]

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.

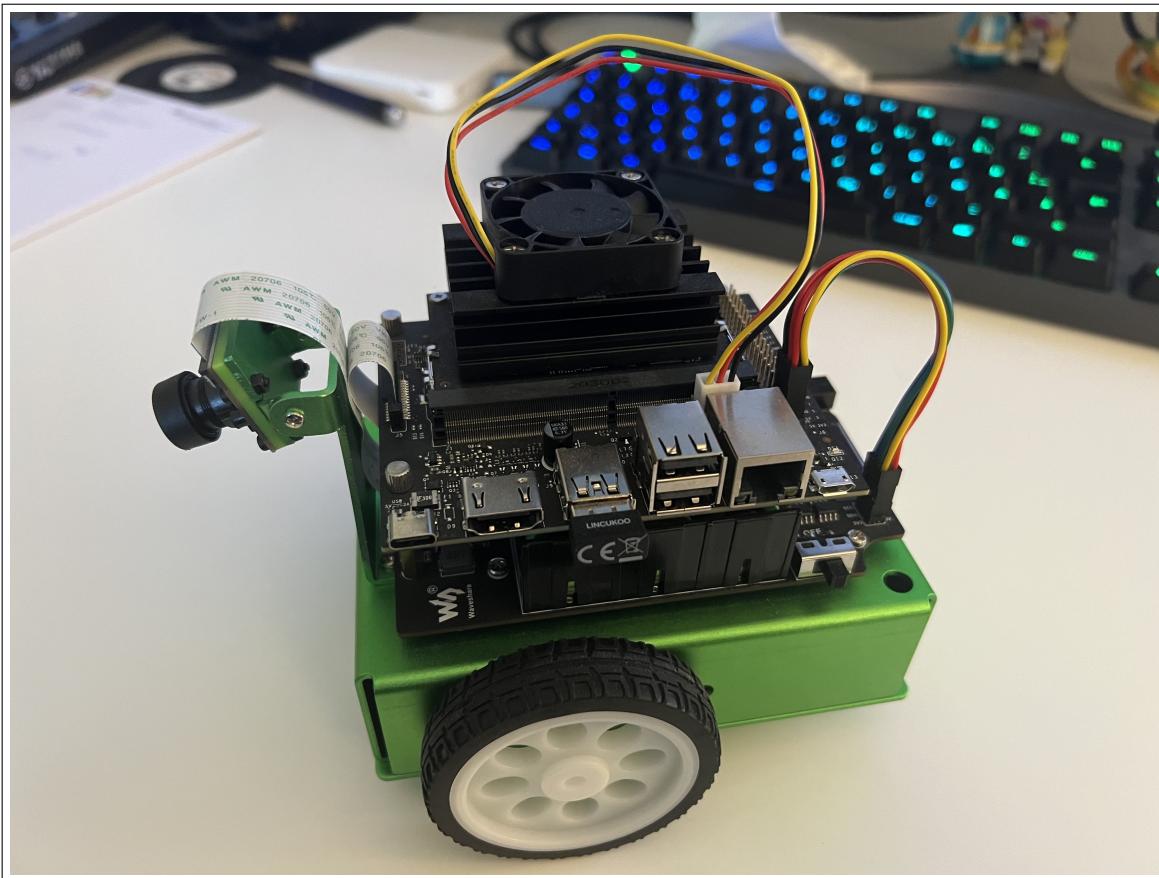


Abb. 3.3: Aufbau des Jetbot-Roboterfahrzeugs

### 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](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.

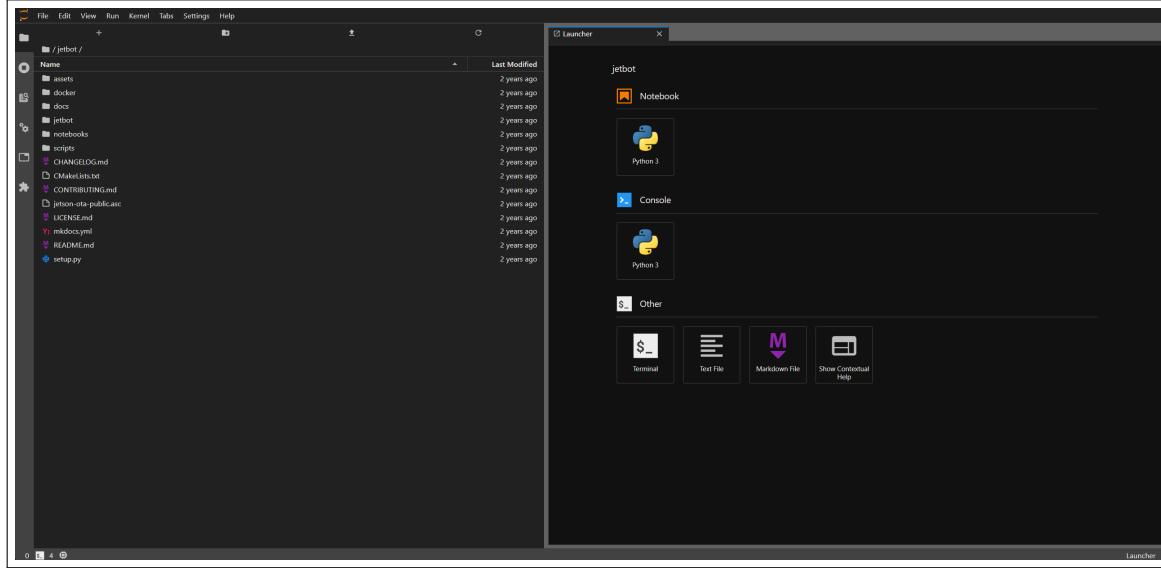


Abb. 3.4: Bedienoberfläche des Jetbots über JupyterLab

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.

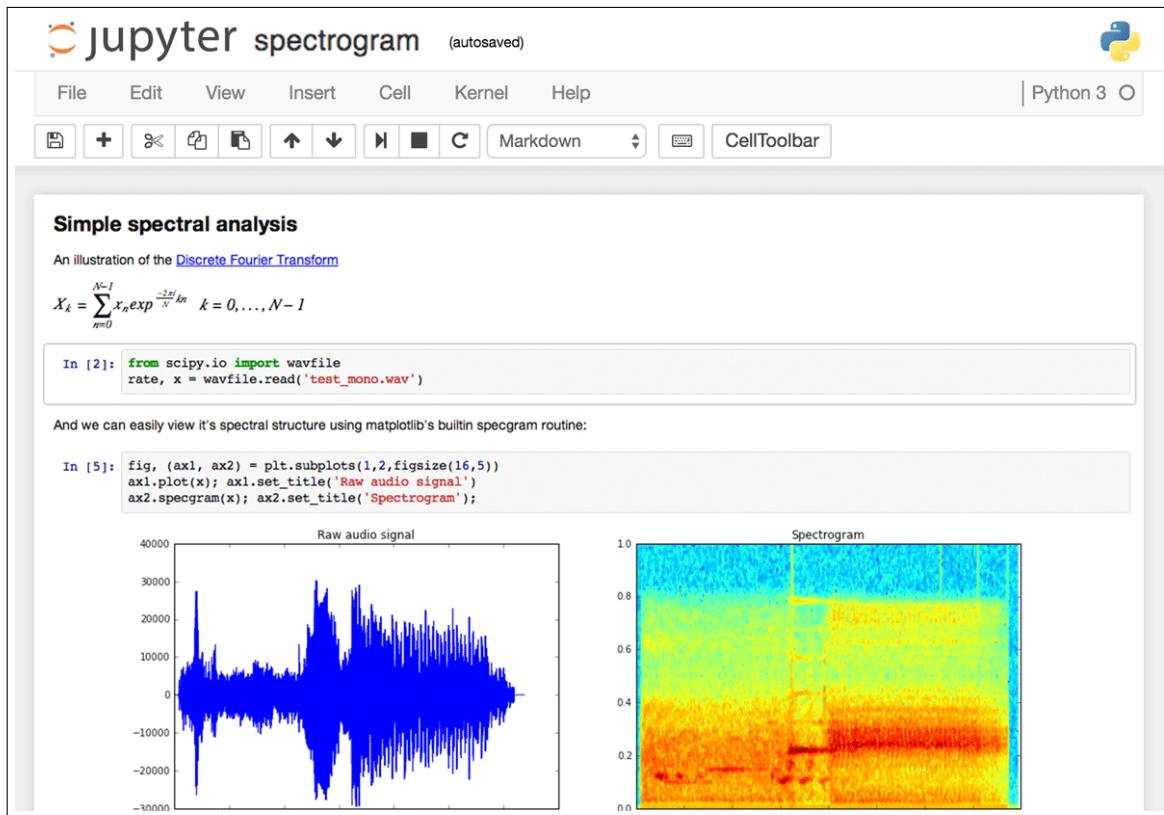


Abb. 3.5: Beispielhafte Abbildung eines Jupyter Notebooks mit Python-Code und Beschreibung in Textform

## 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.

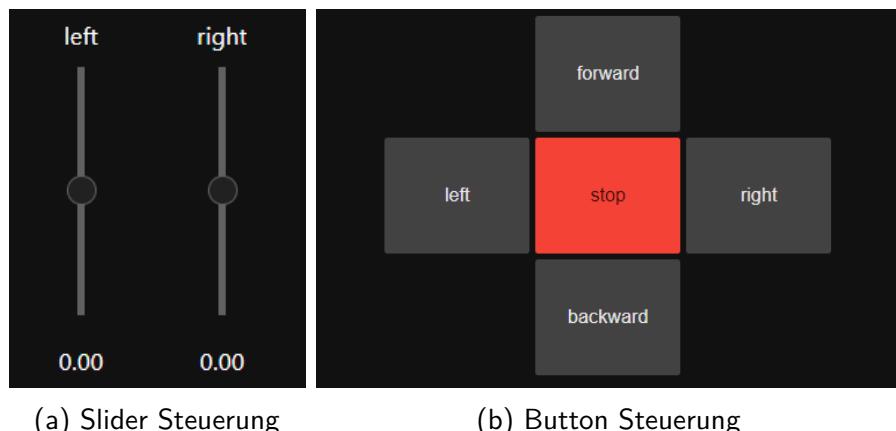


Abb. 4.1: Widgets zum Steuern des Jetbots mithilfe von traitlets

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.

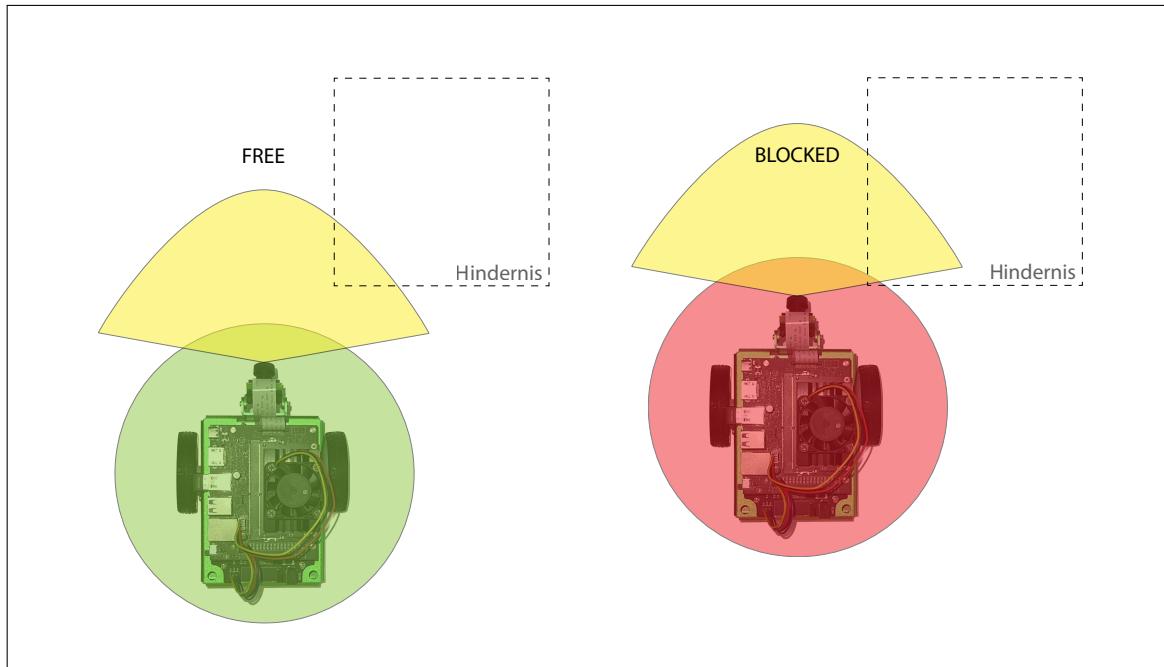


Abb. 4.2: Sicherheitsblase des Jetbot Roboterfahrzeugs im free und blocked Szenario

### 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.

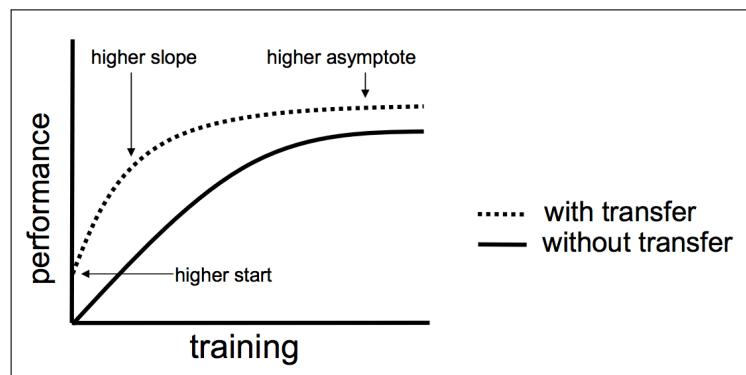


Abb. 5.1: Vorteile des Transfer Learning [7]

## 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.

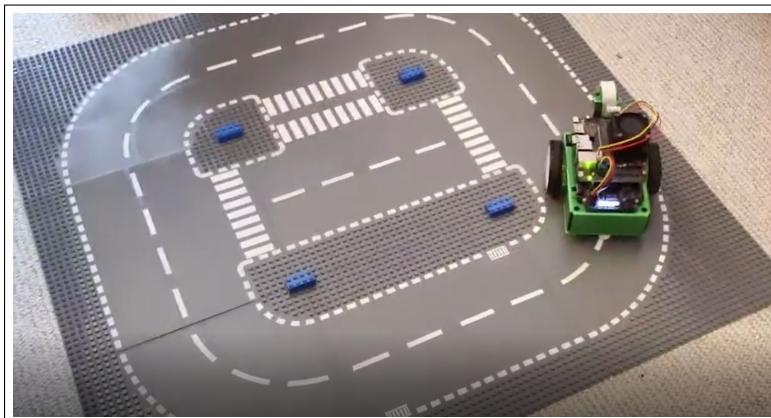


Abb. 6.1: Testversuch zum autonomen Fahren unter Vorgabe von Straßenmarkierungen

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 24, 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

```

## 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](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](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](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](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](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](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](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).