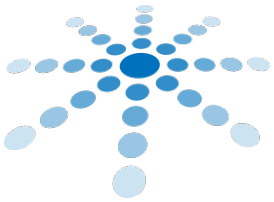


Hochschule Aalen

Hochschule Aalen

Fakultät Mechatronik

# Autonomes Fahren
# Objekterkennung

*Marco Burkhardt*

Supervised By
Herr Prof. Dr.-Ing. Juergen Baur
Herr Stefan Bäuerle

17. Juni 2021

# Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen.

Datum: _____ Unterschrift: _____

# Zusammenfassung / Abstract


### Zusammenfassung

In dieser Arbeit wird mit einem Künstlich Neuronalen Netzwerk (KNN) ein Objekterkennungsalgorithmus mit Matlab entworfen, der in Echtzeit Fahrzeuge erkennt. Der Algorithmus wird auf einen Jetson Nano von Nvidia installiert und dort auf der Graphik Prozessor Unit (GPU) ausgeführt. Werden Objekte erkannt, wird die Information einer Verarbeitungspipeline übergeben. Die Pipeline manipuliert die gewonnenen Daten in gewünschter Weiße. Am Ende sollen die Informationen über die Objekte über das Control Area Network (CAN) versendet werden. Da der Jetson Nano nicht über einen eigenen CAN Controller verfügt, wird eine Möglichkeit besprochen, wie die Kommunikation über den Bus realisiert werden kann.

Im ersten Teil wird besprochen, was ein Künstlich Neuronales Netwerk ist und warum es sich für die Objekterkennung eignet. Im zweiten Teil wird der Jetson Nano vorgestellt, will heißen die Hardware (Developement-Board, Platine). Im dritten Teil wird die Erstellung und die Installation des Objekterkennungsalgorithmus vorgestellt. Im vierten Teil wird die Verarbeitungspipeline besprochen, was diese ist und wie sie sich ins Projekt einfügt. Im fünften und letzten Teil werden die einzelnen Schritte der Arbeit nochmals in einem größeren Zusammenhang betrachten. Es wird die schlußendliche Versendung der Nachrichten via CAN erläutert. Weiterhin wird ein Ausblick gegeben, wie das Projekt weiter geführt werden kann und welche Punkte im Laufe der Arbeit auftauchten, die verbessert werden können.

### Abstract

This paper is about designing a real time algorithm recognizing objects with the aid of artificial neural networks in matlab. The algorithm is installed on a Jetson Nano from Nvidia and executed there on the graphics processor unit (GPU). If objects are recognized, the information is passed to a processing pipeline. The pipeline manipulates the data obtained as desired. At the end, the information about the objects should be sent via the Control Area Network (CAN). Since the Jetson Nano does not have its own CAN controller, a possibility is discussed how communication can be implemented via the bus.

The first part discusses what an artificial neural network is and why it is suitable for object recognition. In the second part the Jetson Nano is presented, that is to say the hardware (development board, circuit board). The third part introduces the creation and installation of the object recognition algorithm. In the fourth part, the processing pipeline is discussed, what it is and how it fits into the project. In the fifth and last part, the individual steps of the work will be considered again in a larger context. The final transmission of the messages via CAN is explained. Furthermore, an outlook is given of how the project can be continued and which points emerged in the course of the work that can be improved.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

**RGB** Rot Gelb Blau

**RISC** Reduced Instruction Set Computer

**IDE** Integrated Developement Enviroment

**FPS** Frames Per Second

**GPU** Graphic Process Unit

**CPU** Command Prozess Unit

**NVCC** NVidia Cuda Compiler

**KI** Künstliche Intelligenz

**AI** Artificial Intelligence

**KNN** Künstlich Neuronale Netze

**CNN** Convolutional Neural Network

**CAN** Control Area Network

# 1 Einführung Künstliche Intelligenz

Damit ein Fahrzeug autonom, d. h. ohne menschliches Zutun, im Straßenverkehr reagieren kann, muss es Fähig sein, die verschiedenen Verkehrsteilnehmer zu identifizieren. Hierbei wird sich einem Teilgebiet der angewandten Informatik bedient — der Künstliche Intelligenz (KI), auch Artificial Intelligence (AI) genannt.

In diesem Kapitel wird folgendes behandelt:

- Was ist Künstliche Intelligenz
- Modellbildung Neuronaler Netze
- Technische Umsetzung
- übersicht: Convolutional Neueral Networks (CNN)
- Bisheriger Stand im Projekt — Status Quo

## 1.1 Was ist Künstliche Intelligenz

*Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better.*[1]

KI ist eine Teildisziplin der angewandten Informatik, in der es um automatisierte Problemlösungen geht. Die Ursprünge lagen im Ansatz, allein Logik zu verwenden. Nachteil dabei: hohe Komplexität für Abdeckung aller Fälle. Es wurde als neuer Ansatz das menschliche Gehirn als Grundlage verwendet. Menschen reagieren auf eine sich verändernde Umwelt. Wir lernen aus Beispielen. Es erfolgt keine Abarbeitung von Schlussregeln wie "WENN ..., DANN ...". Man stelle sich das Halten des Gleichgewichts beim Fahrradfahren oder das Schreiben auf einem Blatt Papier vor. Die Reaktionszeiten für eine sequenzielle Abarbeitung wäre zu hoch. Beim Fahrradbeispiel wäre man schon längst vor der Abarbeitung aller Regeln umgeflogen. Das selbe gilt analog für den Fall des Schreibens, bspw. eines Briefes. Es würde einfach viel zu lange dauern und die Regeln wären viel zu komplex.

Unser Gehirn funktioniert anders. Es passt sich den Situationen an und greift dabei auf Erfahrungen von schon erlebten Situationen zurück. Das wollte man auch für Maschinen — dynamische Anpassung auf sich ändernde Randbedingungen. Das ist auch der Grund, warum künstliche Netze antrainiert werden müssen. Sie sollen auf Situationen reagieren und dabei auf "Erfahrung von schon erlebten" zurückgreifen. Das "Schon Erlebte" stellt dabei das Trainingsmaterial dar, mit dem das Netzwerk antrainiert wird. Dennoch, im Allgemeinen ist zu entscheiden:

[1] [Bos20]

- Existiert ein Algorithmus, dann ist dieser vorzuziehen.
- Ist Erfahrung vorhanden, ein Problem anhand von Regeln zu beschreiben, ist dies vorzuziehen.
- Wenn die ersten beiden Ansätze nicht zum Erfolg führen oder nicht führten, dann kann der Einsatz von KI zielführend sein, wenn genügend Daten vorhanden sind, aus denen gelernt werden kann.

Im Straßenverkehr ändern sich zu jedem Zeitpunkt die Randbedingungen. Autos biegen ab, bremsen, überholen. Es gibt Verkehrsschilder, Ampelanlagen, Fußgänger- und Fahrradwege und mehr. Menschen greifen im Straßenverkehr auf ihre Erfahrung zurück. Möchte man als Ziel ansetzen, dass Fahrzeuge in der Lage sein sollen, im Verkehr ohne menschliches Zutun sich zu bewegen, ist eines ersichtlich. Die KI ist prädestiniert für den Einsatz im Bereich *autonomes Fahren*.

## 1.2 Künstlich Neuronales Netz

Künstlich Neuronale Netze (KNNs) versuchen das menschliche Gehirn nachzubilden. Wie unsere Nervenzellen trainierbar sind und Steuerungsaufgaben übernehmen, so möchte man auch, dass Computerprogramme ähnlich befähigt sein sollen, in analoger Weiße auf neue Situationen aus schon erlebten Beispielen zu reagieren. Man bedient sich hierbei dem Gehirn als biologischen Bauplan Abb. 1.1.

Die Dendriten nehmen das Signal auf, leiten es an den Zellkörper weiter. Dort findet eine Gewichtung der Informationen statt. Die gewichtete Gesamtinformation geht über das Axon weiter und verteilt sich auf die Synapsen, welche jeweils mit weiteren Dendriten anderer Neuronen verbunden sind. Ein einzelnes Neuron wird in Form eines Softwarebausteins nachgebildet. Ein Neuron $j$ besteht aus:

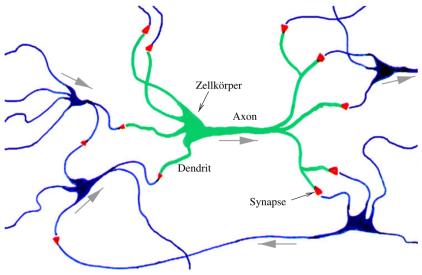



Abbildung 1.1: Menschliches Neuron; [Bos20]

- $k$ gewichteten Eingängen $W_{kj}$
- aus der Summe $net_j$ der jeweils einzelnen Produkten der Gewichte $W_{kj}$ und dem Wert des Eingangs $O_k$. $net_j$ stellt damit die Information zusammen, die aus dem Netz in das Neuron eingehen.
- der Aktivität des Neurons, d. h. wie stark der potenzielle Einfluss von $net_j$ auf andere Neuronen sein kann. $\Theta_j$ ist ein *Hyper Parameter*.
- der Ausgabe — Verbindung zu anderen Neuronen

Abb. 1.2 entspricht der Beschreibung.

Viele Neuronen sind so miteinander verbunden und bilden zusammen ein *Neuronales Netzwerk*. Als technische Modellierung existieren verschiedene Varianten, von denen aber hier nur die *forwärts gerichtete* Variante interessiert.

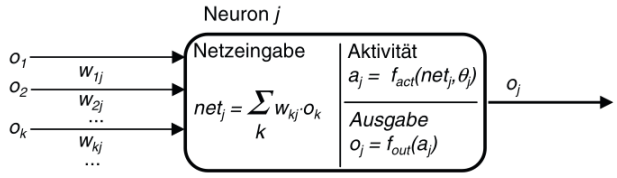
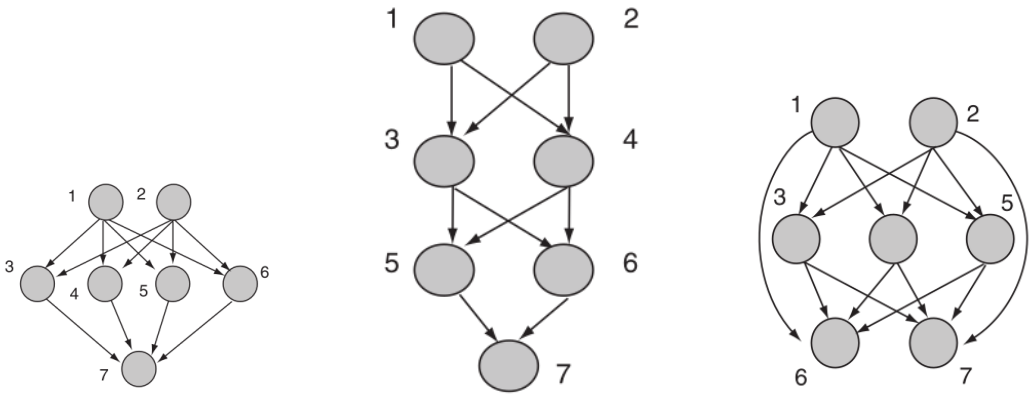



Abbildung 1.2: Modellierung eines einzelnen Neurons — Künstliches Neuron; [Bos20]



(a) (b) (c)

Abbildung 1.3: Fully Connected, Forward Looking Networks: a) Neuronen der Hidden Layer gehen in die Breite; b) Hidden Layer gleichbleibende Anzahl an Neuronen; c) Hintere Neuronen greifen direkt auf die unteren Schichten zu

Abb. 1.3 zeigt verschiedene Ausführungen eines *Feed Forward Neural Networks* oder auch *Fully Connected Neural Network*. Das entspricht der klassischen Vorstellung eines KNNs. Jedes Neuron ist mit jedem vorherigen und jedem nachfolgenden Neuron verbunden.

Die erste Schicht bildet den Eingang. Dort treffen Signale ein, z. B. in Form eines Bildes. Es geht weiter in die zweite Schicht. Tiefer liegende Schichten werden auch als *hidden layer* bezeichnet, da sie von außen, d. h. von den Schnittstellen, nicht zu sehen sind. Die letzte Schicht ist die Schnittstelle nach Außen. Jedes Neuron gibt eine Wahrscheinlichkeit über den repräsentierenden *Classifier* aus. Zum Beispiel als Anwendungsfall in der Objekterkennung, ob Objekt von der Klasse "Afrikanischer Elefant" ist. Existieren mehrere Neuronen in der letzten Schicht, kann das Netz Aussagen über mehrere Classifier machen.

Es gibt noch wesentlich mehr Architekturen / Topologien, wie ein KNN aufgebaut sein kann. Jede hat ihre Vor- und Nachteile und somit eigene Anwendungsgebiete. In der Objekterkennung / Bildverarbeitung wird eine andere Variante als die *Fully Connected* Variante eingesetzt — das CNN.

## 1.3 übersicht CNN

Im Gegensatz zum *Fully Connected Network* sind die Neuronen des CNNs nicht mit jedem Neuron der übergeordneten Schicht verbunden. Die Verbindung entsteht als Faltungsoperation mit einem Filter bestimmter Größe (Kernel-Size). Der Filter "wandert" über das Bild

und verknüpft so jeden Bildbereich mit den darüberliegenden Neuronen. Jedes Neuron ist somit mit einem Filter verknüpft. Die Ergebnisse der Faltung entsprechen den Gewichten, mit denen das Neuron mit der darunter liegende Schicht verbunden ist. Allgemein gibt es mehrere Filter pro Schicht; dementsprechend besteht jede Schicht aus mehreren Neuronen. In Abb. 1.4 ist das Vorgehen aufskizziert.

Die Parameter der Filter haben anfangs zufällige Werte. Die Werte werden im Trainingsprozess "gelernt" oder "antrainiert". Die Faltung (*der Filter*), wird zur Reduktion (*Feature Extraction*) der inneren Schichten (*Hidden Layers*) angewandt.

Man kann für die inneren Schichten ein schon vor trainiertes, beliebiges CNNs verwenden. Einzige Voraussetzung ist, das Netzwerk darf bspw. nicht mit dem Verhalten von Käferarten vortrainiert sein, wenn doch das Ziel ist, Objekte zu erkennen. Vorteil eines vortrainierten Netzwerks (*Pre-Trained Network*) ist, es muss weniger Zeit für das Lernen seines Anwendungsgebietes aufgewendet werden und die *Feature Extraction* ist schon aufgebaut. Es gibt hierbei zwei wesentliche Punkte zu beachten:

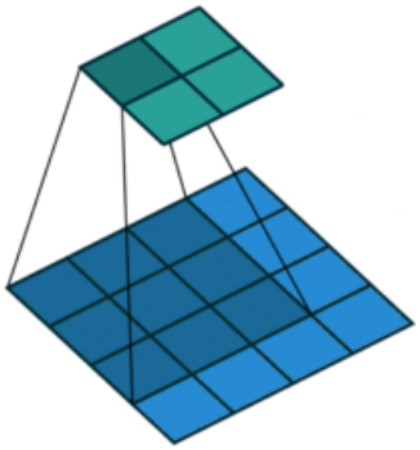


Abbildung 1.4: Jedes Neuron ist mit einer Faltungsoperation (Filter) mit dem darüber liegenden Neuron verbunden.

- Es muss "nur" der Eingang auf die korrekte Größe des Bildes angepasst werden.
- Die letzte Schicht muss ein *YOLO-Layer* sein mit den gewünschten Ausgangsneuronen für jede Klasse an zu detektierenden Objekten.

Die Vorgängergruppe hat dieses Vorgehen angewandt und ein schon vortrainiertes Netzwerk als Grundlage verwendet. Vorteil: Lernprozess kostet weniger Zeit und weniger Trainingsmaterial wird benötig; d. h. es kann mehr Energie / Zeit an die Anpassung des konkreten Anwendungsfalls aufgewand werden.

In Abb. 1.5 ist die Struktur des YOLO-Netzwerks aufskizziert. Im Unterschied zu den vorherigen Betrachtungen:

1. Die Hidden Layers bestehen intern jeweils wieder selbst aus einer Struktur speziell aufeinanderfolgenden Arten von Layern.
2. Die letzten Schichten müssen *fully connected* Yolo Layer sein.

**Zu Erstens**:

- *Faltunsschicht* ist für die Reduktion des Layergröße zum einen und zum anderen für die Feature Extraction (nur die wichtigen Details sind für weitere Betrachtung wichtig) verantwortlich.

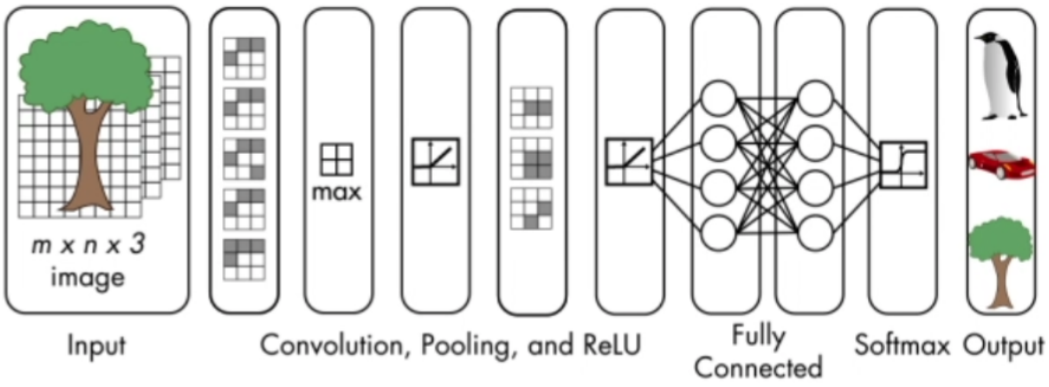



Abbildung 1.5: Beschreibung von links nach rechts: Input (der Situation angepasst, im Projekt: Bild), pretrained CNN mit Faltungs- und Aktivierungsschicht, YOLO-Layer (auf Bedürfnisse angepasst, da pro zu erkennende Klasse ein Ausgangsneuron, im Projekt: ein Neuron (Fahrzeug))

- *Normalisierungsschicht* ist für die Beschleunigung des Trainingsprozesses verantwortlich.
- *Aktivierungsschicht* entscheided darüber, ob ein Neuron "gezündet" wird (geht in die aktuelle Berechnung ein) oder nicht.

**Zu Zweitens**:

Es gibt so viele Neuronen im *Fully Connected Layer*, wie es zu erkennende Objekte gibt. Hier im Projekt wird eine binäre Entscheidung getroffen — Auto da oder nicht da; heißt, die letzte Schicht besteht aus einem Neuron. Sollen aber mehrere Objekte erkannt werden, gibt es genau so viele *Fully Connected* Neuronen wie zu erkennende Objekte.

Die letzte Schicht *Softmax Layer* hat die Aufgabe, die Wahrscheinlichkeit, dass ein oder mehrere Objekte im Bild erkannt werden / worden sind, in ein Wahrscheinlichkeitswert wiederzugeben zwischen 0 und 1 — 0 für 0% und 1 für 100%.

Vorteil ein Framework zu verwenden — sei es Matlab wie hier im Projekt oder Python Implementationen wie PyTorch, Keras und mehr — ist, man braucht sich nicht expliziet um die konkrete Umsetzung der einzelnen Schichten kümmern. Diese Arbeit nimmt das verwendete Netzwerk ab. Man kann darauf vertrauen, dass die internen Strukturen und Methoden so effizient wie möglich implementiert wurden. Dennoch ist Wissen, wie mit dem Netzwerk umgegangen werden muss, Grundlagen, auf denen die Implementationen aufbauen, auch für den Anwender notwendig. Wie könnte man sonst abschätzen, ob das berechnete Ergebniss plausibel ist oder nicht? Also ist man auch als Anwender eines Frameworks, wie es die Matlab Integrated Developement Enviroment (IDE) bereitstellt, nicht von Grundwissen befreit.

## 1.4 Bisheriger Stand im Projekt

Das Ergebniss von Vorgängerarbeiten sind drei antrainierte Neuronale Netze. Die Netze arbeiten binär — Erkennung von Auto erkannt / nicht erkannt. Sie besitzet zwei unterschiedliche *Tiefen*:

- 25 Hidden Layers
- 50 Hidden Layers

In dieser Arbeit wird die Variante mit 25 Layers verwendet [BS20]. Es hat sich gezeigt, dass das flachere Netzwerk um ca. 900% bis 1000% schneller Objekte erkennt als das größere Netzwerk. Auch die Kompilierungszeit ist um eine ganze Potenz schneller. Wie es in der Zuverlässigkeit der Objekterkennung aussieht, bleibt offen. Da entschieden wurde, selbst wenn das größere Netzwerk eine höhere Zuverlässigkeit besitzt, ist die Geschwindikeit im Betrieb nicht tragbar — teils geringer als ein Frames Per Second (FPS). Genauer wird darauf im weiteren eigegangen.

# 2 Jetson Nano

In diesem Kapitel wird die Jetson Nano Hardware vorgestellt, weiterhin die Einrichtung der Hardware und die Ordenerstruktur — wo das Netzwerk und die Anwendungsprogramme liegen.

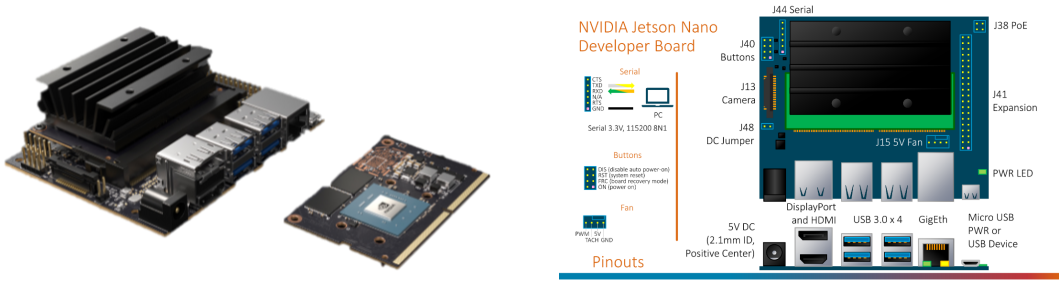
Es folgt:

- Übersicht Jetson Nano
- Installation der nötigen Abhängigkeiten
- Ordnerstruktur der Projektdateien

Die Hauptinformationsquelle seitens der Hardware ist Hersteller selbst [Nvi01a] und [Nvi01b].

## 2.1 Overview Jetson Nano

Einen Überblick der Hardware ist in Abb. 2.1 zu sehen. Der Jetson besteht aus einen Developement-Board — Eine Hardwareumgebung für die Entwicklung von Projekten — und einem Steckmodul. Das Steckmodul wird in die Steckleiste des Dev-Boards befestigt. Der Grafik- wie auch der Mikroporzessor sitzen auf dem Steckmodul. Somit kann nach Entwicklung und Testen das Steckmodul unmittelbar in das Produktivsystem integriert werden; sprich, es kann auf die Entwicklerplatine im Endprodukt verzichtet werden. Eine gute Einführung ist auf dieser Internequelle zu sehen.




(a) (b)

Abbildung 2.1: Developement-Board mit Steckmodul (a) und Schema (b)

Abb. 2.2 zeigt, wie der Jetson Nano im Gesamtprojekt eingesetzt wird. Im System gibt es vier wesentliche Komponenten:

- Raspberry Pie — Zentrale Steuereinheit des Systems

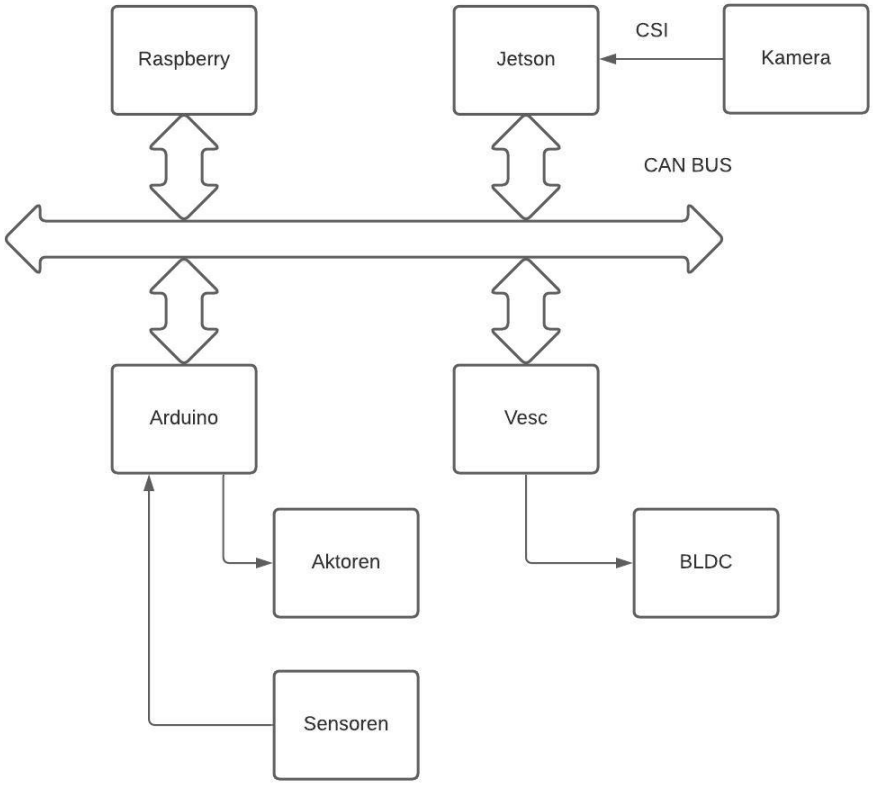



Abbildung 2.2: Alle Hardwarekomponenten des Systems

- Jetson Nano — Das “Auge” des Systems
- Arduino — Auswertung von Sensoren und Stellglied von Aktoren
- Vesc Motorcontroller — Stellglied für den BLDC Motor

Alle Hauptkomponenten sind mit dem *CAN*-Bussystem verbunden — siehe Abb. 2.2. Hierüber wird die Kommunikation abgewickelt. Der Jetson nimmt über ein Kameramodul Bilder der Umgebung auf. Intern werden die aufgenommen Bilder einem Neuronalen Netzwerk übergeben, das das Bild auf interessierende Objekte untersucht. Hier sind die interessierenden Objekte Fahrzeuge. Und zwar nur Fahrzeuge als solches, heißt, es wird nicht weiter zwischen verschiedene Fahrzeugarten unterschieden. Die Objekterkennung ist somit binärer Natur. Wurde ein Fahrzeug erkannt, wird eine Nachricht über das Bussystem verschickt. Es ist Aufgabe des Can-Protokolls, die Nachricht dem richtigen Adressaten zu übergeben. Hier ist der richtige Adressat das Raspberry-Modul (Raspberry Pi). Dort wird entschieden, was mit den eingehenden Informationen geschehen soll.

## 2.2 Software-Installation

Die Softwareinstallation wird detailiert auf der Nvidia Homepage beschrieben [Nvi01b]. Deswegen hier nur ein paar Punkte, die aufgefallen sind und zu erwähnen sich lohnen:

- Jetpack
  - Aktuellste Installation auf SD Card Image
  - SD Card Image von *Getting Started with Jetson Nano* herunterladen
  - Gibt auch andere Quellen, aber dort nicht garantiert, dass *JetPack* und Abhängigkeiten installiert werden
- SD Karte manchmal nicht lesbar gewesen
  - Staubeinschluss
  - Wie bei Nintendo 64 oder Gameboy
    - SD Karte entfehrnen und "pusten" hilft
- Starten im "Headless" Mode
  - benötigt IP Adresse
  - mit Befehl `$ ipconfig` im Jetson zu bekommen

## 2.3 Ordner Struktur

Es existieren zwei Hauptordnerstrukturen:

- yoloCNNForDetection
- voodo_code

Beide liegen im Home-Verzeichnis des Users `</home/<user>>`, wobei `<user>` in diesem Fall `jetson` ist.

*yoloCNNForDetection*: Im der ersten Struktur liegt der von Matlab generierte Code und das Kompilat (Statische Library). Jedes mal, wenn der Anwender am Host PC in Matlab einen neuen Algorithmus definiert und übersetzen lässt, diesen anschliessend auf das Zielsystem überträgt, finden sich die generierten Files und die kompilerte Datei in dieser Ordnerstruktur.

Matlab erstellt eine tiefe Struktur, heißt: sehr viele Unterorder und Verzweigungen. Aber sämtlicher Code und Kompilate finden sich dort.

**Wichtig!!!**: Auch wenn später statische Library in anderes Programm — anderer Ausführungsprozess — eingebunden wird, liegt das YoLo Netzwerk in dieser Struktur und ist somit eine **Abhängikeit**. Die Ordnerstruktur darf **nicht** gelöscht oder verändert werden. Die statische Library verwenden absolute Pfade.

*voodo_code*: Dort sind wiederum drei Haupt-Unter-Verzeichnisse angedacht:

- detector
- filter_1
- can

Jeder dieser Unterverzeichnisse definiert ein seperates Programm, das jeweils einen eigenen Prozess startet.

*detector*: Das ist ein in Cuda geschriebenes Programm, das die von Matlab generierte statische Library einbindet und autark auf der Zielhardware nach dem Kompilierungsvorgang (NVidia Cuda Compiler — NVCC) ausführbar ist. Vorteil dieser Vorgehensweise:

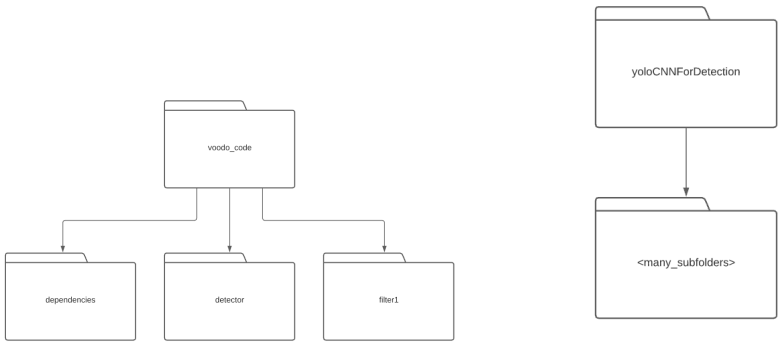
- Host PC zum Starten des Algorithmus nicht mehr nötig
  - Statische Library in unabhängiges Programm (späterer Prozess) eingebunden
- Anwendungsprogrammierer kann beliebige weitere Funktionalitäten dem Programm hinzufügen
  - Kommandozeilen Parser
  - Individuelle Ausgabe
    - Visuellen Modus
    - Ausgabe File Descriptor
    - Bit Stream
    - usw ...

Nachteilig zu erwähnen ist: Programmierer sollte sich mit CUDA Syntax auskennen. Aber oft nicht so großes Problem, da Schnittmenge mit C++ respektive C Syntax vorhanden. Dennoch ist erstrebenswert: so wenig wie möglich in CUDA Code zu arbeiten. Deswegen Aufteilung in weitere Programme nach Motto: viele kleine Programme, die nur *eine* Aufgabe haben, aber diese gut und effizient umsetzen.

Makefile für den Kompaliervorgang im Projekt hinterlegt und kann mit `$ make` Befehl gestartet werden.

*filter_1*: Die übertragung der Information (Objekt erkannt oder nicht) ist als Bit-Stream über die Standartausgabe definiert. Da oben erwähnt, ein Ziel ist, so wenig wie möglich in CUDA zu Programmieren — sehr lange Compile-Time, sehr viel Abhängikeiten, kompliziertes Make-File —, werden weitere Manipulationen in einem getrennten Programm umgesetzt *Filter*. Der Filter ist in C++ geschrieben und bereitet die Versendung der Informationen über CAN vor. Eine weitere Bearbeitung der Daten könnte wie folgt ausehen:

Es werden nur weitere Nachrichten an Busteilnehmer gesendet, wenn signifikante änderungen in der Objekterkennung vorliegen. Bspw. Objekt (Fahrzeug) wurde erkannt und Information über Bus gesendet. Es werden in der Sekunde mehrere Bilder aufgenommen und auf erkannte oder nicht erkannte Objekte überprüft. Es gibt jetz zwei Möglichkeiten: Es kann immer eine Nachricht versendet werden, wenn Objekt erkannt — unter Umständen viele Nachrichten pro Sekunden über Bus — oder es wird eine Initialnachricht versendet: Signalisierung Objekt erstamlig erkannt und weitere nur, wenn sich bspw. Entfehrnung oder Näherungsgeschwindigkeit signifikant ändern. Nachteil der ersten Möglichkeit ist, dass Bus




(a) (b)

Abbildung 2.3: (a) voodo_code: Dort sind sämtlchen Prozesse definiert und werden von dort ausgeführt (Bash-Skript), (b) Yolo Netzwerk: Viele Unterverzeichnisse von Matlab generiert; Ordnerstruktur ist im Home-Verzeichnis hinterlegt

mit Nachriten "überschwemmt" wird, aber aus den vielen Nachrichten sich keine neue Erkenntnis generieren lassen. Dieser Nachteil wird in der zweiten Möglichkeit versucht zu umgehen. Dort werden nur weitere Nachrichten versendet, wenn es wesentliche änderungen gibt — hohe Näherungsgeschwindigkeit von erkanntem Fahrzeug, ....

*can*: Im Projekt wurde das *Hauptprogramm* und der *Filter* implementiert. Die entgültige Kommunikation über CAN Bus blieb aus. Folgearbeiten können mit einer beliebigen anderen Sprache — bspw. Python, C, Rust, ... — durchgeführt werden. Nvidia stellt bspw. ein Python Interface für die GPIO bereit. Es ist dementsprechend zu prüfen, ob ein Interface für SPI Controller existiert. Der Jetson hat kein eigenen CAN Controller, deswegen CAN Kommunikation nur über SPI möglich.

**Zusammengefasst**

Die komplette Bearbeitung läuft in drei getrennten Prozessen ab. Eine unidirektionale Kommunikation ist notwendig. Die Informationen von "tiefer" liegenden Prozessen müssen an den darüberliegenden Prozess weitergegeben werden. Erwähnt wurde, dass als Ausgang ein Bit-Strom über Standart Ausgabe definiert wurde. Da auf dem Jetson ein Linux Betriebssystem ausgeführt wird, kann die Umsetzung der *Inter-Prozess-Kommunikation* dem Betriebssystem übergeben werden. Als Mechanismus werden dafür Pipes verwendet.

- `$ <Hauptprogramm> | <Filter_1> | <CAN_Dispatcher>`

Der senkrechte Balken signalisiert dem Betriebssystem, dass ein Kommunikationskanal von Prozess 1 zu Prozess 2 gebildet werden soll. Die gesamte Ordnerstrukture ist wie folgt in Abb. 2.3 gezeigt umgesetzt.

# 3 Deployment / Installation

In der Objekterkennung hat sich der YOLO Algorithmus bewährt. Die YOLO Variante unterscheidet hauptsächlich in den Letzten Schichten von anderen CNN Netzwerk Architekturen.

Es gibt mehrere Technologien, die den Aufbau eines Neuronalen Netzwerks unterstützen. Einige Bibliotheken in Python sind beispielsweise: PyTorch, Keras, Tensor Flow. Auch Matworks stellt ein eigenes Framework bereit. Vorteil von Matlab ist: Es gibt sehr viele speziell auf Matlab angepasste Tools für den kompletten Erstellungs-, Evaluierung- und Anwendungsprozesses. Hingegen die frei zugänglichen Alternativen wie in Python bieten einen kleineren Anwendungsbereich. Beispielweise, wenn für die Evaluierung im Bild die Objekte gekennzeichnet werden sollen, müssen andere Tool als z. B. nur Tensor Flow verwendet werden wie OpenCV zur Bildverarbeitung. OpenCV reichert das Bild mit Rahmen um erkannte Objekte an und setzt Labels. In Matlab ist alles dabei und weitere externen Tools sind unnötig.

Im Projekt wird Matlab als Entwicklungsumgebung verwendet. Vorgängerarbeiten bauten darauf auf. Würde man sich gegen Matlab als Framework entscheiden, müsste der Erstellungsprozess von vorne begonnen werden.

Losgelöst von den verwendeten Werkzeugen, kann die Installation *engl. Deployment* abstrahiert werden, will heißen, der übergeordnete Vorgang ist losgelöst von konkreten Technologien — zu sehen in Abb. 3.1.

In diesem Kapitel werden folgende Fragen beantwortet:

- Die Installation mittels Matlab. Es wird der konkrete Code — so wie er im Projekt angewendet wird — beschrieben.
- Zusammenfassung / Evaluierung

## 3.1 Installation

Der Vorgang, einen Matlab Algorithmus auf die Zielhardware zu übertragen und auszuführen, wird in drei Schritten abgearbeitet.

**Schritt 1**: Ein Algorithmus muss erstellt werden. Der Algorithmus hat die Form einer Routine (Funktion). Die Routine wird in der Matlab IDE erstellt. Der erstellte Code wird im Laufe des Vorgangs in Cuda Code umgewandelt und in der Zielhardware übersetzt (kompiliert). Der Vorgang wird Cross-Kompilierung genannt.

Jetzt foglt die Beschreibung des Algorithmus. Er ist in der Matlab-Syntax verfasst. Die Beschreibung setzt sich wie folgt zusammen: erst Code, anschließend Erläuterung. Eine Flussdiagram ueber den Algorithmus ist in Abb. 3.2 zu sehen.

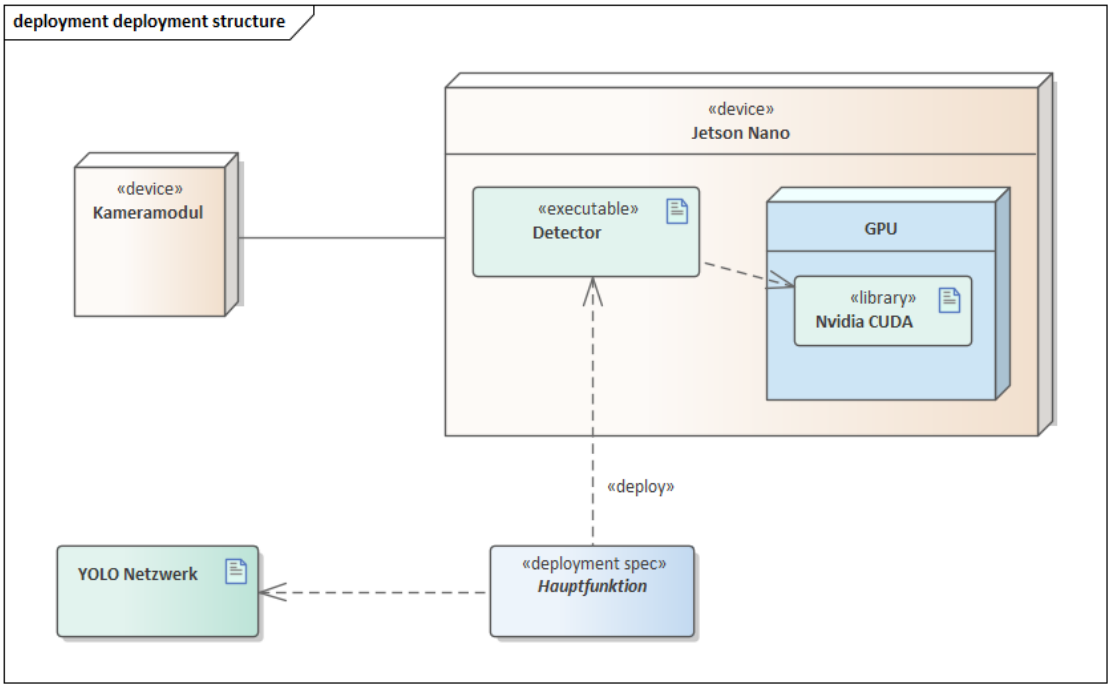



Abbildung 3.1: Die Hauptfunktion wird als Routine geschrieben. Sie verwendet als externe Abhängigkeit das von der Vorgängergruppe erstellte Neuronales Netzwerk. Der Host (f. i. Windows PC) überträgt Quelldateien auf die Zielhardware. Dort werden die Quelldateien übersetzt und eine ausführbare Datei *Binary* erstellt (hier: eine statische Library). Die Binary verwendet Nvidia CUDA Bibliotheken. Dort werden die aufwendigen Berechnungen der Objekerkennung ausgeführt. Die Objekterkennung verwendet ein Kameramodul.

```
function [x y width height score] = detectFunction()
%codegen
```

Es wird eine Fnktion deklariert, die fünf Ausgabewerte liefert. Auch werden nicht alle Matlab-Interne Funktionen für die Konvertierung unterstützt. In der Funktion ist die `#codegen` Direktive zu empfehlen. Es wird dann eine Prüfung während der Erstellung des Algorithmus durchgeführt, ob Code gültig ist oder nicht.

```
persistant mynet
persistant hwobj
persistant cam

if isempty(mynet) || isempty(hwobj) || isempty(cam)
    hwobj = jetson
    cam   = camera(hwobj, 'vi_output, imx219 6-0010', [1280 720]);
    mynet = coder.loadDeepLearningNetwork('yoloNetwork.mat');
end
```

Die Variablen *mynet, hwobj, cam* sind mit dem Qualifier *persistant* versehen. In anlehnung

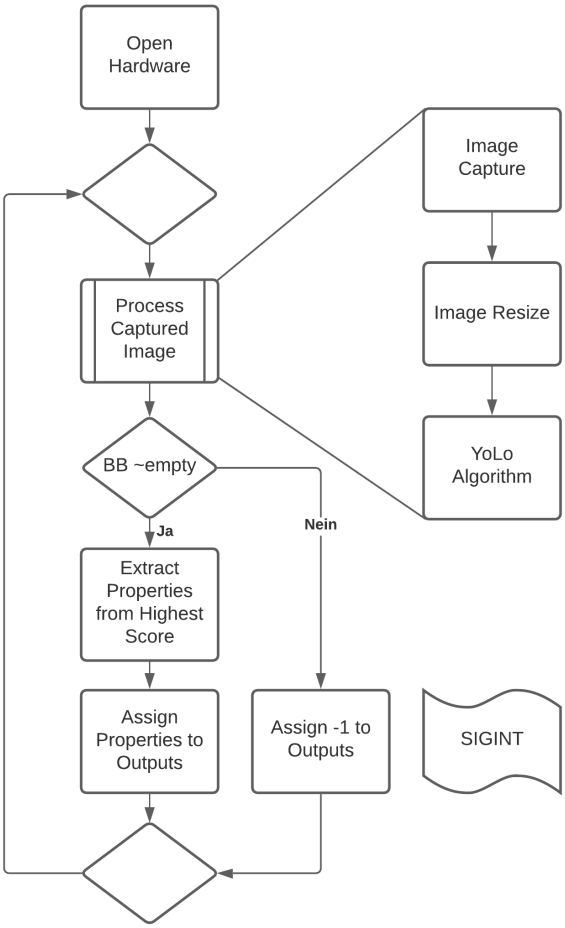



Abbildung 3.2: Flussdiagram Detektor Algorithmus

an C hat das den selben Effekt wie eine *static* Deklaration. Der Code im if-Block wird nur im ersten Funktionsaufruf ausgeführt. Dort werden die Abhängigkeiten initialisiert.

```
img = snapshot(cam);
img = imresize(imt, [224 224]);

[bboxes scores] = detect(mynet, img, 'Threshold', 0.5);
\% only the one with the highest score
[score_ index] = max(scores);
```

Es wird ein Bild eingefangen — PiCam — und dessen Greoße angepasst. Das verwendete Netzwerk hat 224 mal 224 mal 3 Eingangsneuronen. Der Faktor 3 steht für den RGB Farbaum. Jeder Farbraum wird intern vom Netwerk getrennt berechnet und am Ende der Pipline wieder zusammengeführt. Somit benötigt ein Farbbild mehr Rechenleistung. Zukünftig könnte geprüft werden, ob ein Netzwerk, das eine Graustufenabbildung verarbeitet erstens genau so zuverslässig und zweitens schneller oder genauso schnell rechnet. *detect* ist eine Matlab-Interne Funktion. Ihr übergibt man das KNN, das interessierende Bild und defi-

niert einen Schwellwert, ab welchen Zuverlässigkeitswert ein Objekt auch als das tatsächliche Objekt angesehen werden kann oder nicht.

```
if ~isempty(bboxes)
   tmp_bbox = bboxes(idx,:);
   x = tmp_bbox(1);
   y = tmp_bbox(2);
   width  = tmp_bbox(3);
   height = tmp_bbox(4);
else
   \% nothing detected
   x = -1
   ...
   score = single(-1);
```

Wenn Objekt erkannt, wird darum ein Rahmen vom Netzwerk gezogen. Dementsprechen ist das bbox Objekt nicht leer. Darin sind die Koordinaten der Boudingbox gespeichert. Im ersten if-Teil werden die Koordinate ausgelesen und den Ausgangsparametern übergeben. Wurde kein Objekt erkannt, ist die Boudingbox leer und es wird nach den Konventionen die Werte ausgefüllt (kein gültiger Code — wie *errno* in C).

**Schritt 2**: Jetzt wird der in Schritt 1 erstellte Algorithmus auf die Zielhardware übertragen. Dazu muss Verbindung vom Host zur Zielhardware aufgebaut werden. Matlab verwendet eine SSH Verbindung über TCP/IP.. Die Verbindung wird über ein Hardware Objekt hergestellt.

- Target: <IP-Address>
- Login Name: *jestson*
- Passwort: *1111*

Für den Zugriff auf die Hardware des Jetsons muss eine IP-Adresse eingerichtet werden. Es kann entweder eine statische oder dynamische Adresse vergeben werden seitens der Zielhardware. Mit dem Befehl `$ ifconfig` kann die eigene IP-Adresse des Jetsons eingesehen und in der Matlab IDE eingegeben werden.

```
hwobj = jetson(<ip-address>, ‘jetson’, ‘1111’);
```

Das Hardwareobjekt wird mit den Login Informationen der Zielhardware ausgefüllt. Damit kann eine SSH Verbindung vom Host erstellt werden.

```
gpuEnvObj = coder.gpuEnvConfig('jetson');
gpuEnvConfig.BasicCodegen = 1;
...
results = coder.checkGpuInstall(gpuEnvObj);
```

*coder* ist eine statische Funktion bereitgestellt von Matlab. Sie erstellt ein Konfigurationsobjekt, das seinem Randbedingungen unterzogen wird. Jedes Projekt hat andere Randbedingungen. Demzufolge werden auch in unterschiedlichen Projekten unterschiedliche Optionen gesetzt. Zuletzt wird das Konfigurationsobjekt geladen und über die zuvor erstellte Verbindung Prüfungen auf der Zielhardware durchgeführt.

Ist die Prüfung erfolgreich durchgeführt — alle Abhängigkeiten auf Zielhardware geladen und vorhanden —, kann der in Schritt eins erstellte Algorithmus in Code Zielcode umgewandelt werden. Dann erfolgt die Compilierung auf der Zielhardware.

**Schritt 3**: Der Matlab Coder generiert nun Code — nur von Funktionen, nicht von Skripten! Es sollte daher von Anfang an eine Funktion geschrieben werden, anstatt ein Skript zu erstellen und es hinterher in eine Funktion zu konvertieren.

Matlab exportiert den Quellcode auf den Jeton. Ist die Übertragung abgeschlossen, wird auf dem Jetson über den NVIDIA Cuda Compiler *nvcc* der von Matlab generierte Quellcode in eine ausführbare Datei kompiliert.

Zusammengefasst, nötige Schritte für eine Installation sind:

- YOLO Netzwerk bereitstellen
- Zielhardware konfigurieren
- Host konfigurieren
- Main-Funktion erstellen
- Code generieren
- Einbindung der Binary auf Zielhardware

**YOLO Netzwerk**: Es muss ein vorhandenes Netzwerk zur Erkennung von Objekten vorhanden sein. Getestet wird jeweils das von den Vorgängern erstellte und neue erstellte Netzwerk.

**Zielhardware**: Auf der Zielhardware müssen zusätzliche Bibliotheken installiert und Globale Systemvariablen exportiert werden. Benötigte Bibliothek ist die *Simple Direct Medial Layer v1.2* in der standart und developement Version. Der Cuda Pfad, in dem die Cuda Binaries und die Cuda Bibliotheken liegen, kann beispielsweise in der *.bashrc* hinterlegt werden.

**Host**: Die Codegenerierung — C++ und Cuda — sind von externen Tools abhängig. Matlab benutzt die *Gnu Compiler Collection* und mehrere *Cuda Libraries* von Nvidia für die Erstellung der Binaries. In Linux ist der gcc standardmäßig in den Repos hinterlegt. Auf Windows Maschinen heißt das Compiler Paket MinGW. Die Abhängigkeiten seitens Nvidia müssen von deren Homepage heruntergeladen und installiert werden. Es werden Windos- und Linuxsysteme unterstützt.

**Main-Funktion**: Funktion, die in einer Endlosschleife Bilder von der Webcam aufnimmt und dem Detektor übergibt. Der Detektor scannt das aufgenommene Bild nach Objekten.

Wenn Objekte gefunden wurde, dann wird die entsprechende Bounding Box und Label auf das Bild gebunden und anschließend auf dem Bildschirm angezeigt. Hierbei muss entweder ein externer Monitor an den Jetson angeschlossen werden oder eine Remote- hergestellt werden.

**Code Generation**: Der Code wird in der Matlab Umgebung mit ein paar Codezeilen generiert. Das Kompilat kann auf mehreren Wegen auf den Jetson überspielt werden. Entweder über einem USB, via Ethernet Verbindung, etc. .... Man kann allgemein die Binaries in einem beliebigen Pfad installieren. Es muss nur darauf geachtete werden, dass die ausführbaren Dateien hinterher in den Pfad angefügt werden.

**Zielhardware**: Es ist nicht zwingend, eine ausführbare Datei zu erstellen. Eine weitere Möglichkeit ist, statische oder dynamische Bibliotheken zu generieren. Die Bibliotheken können dann in einem anderen Programm eingebunden werden. Der Vorteil, den Code für den Jetson direkt in Matlab als Executable zu generieren, ist, es müssen keine zusätzlichen Einstellungen getroffen werden.

## 3.2 Zusammenfassung

Es wurde ein Algorithmus für die Objekterkennung vorgestellt. Der Algorithmus verwendet das Kameramodul, um in Echtzeit die Umgebung aufzunehmen und auszuwerten. Der Algorithmus ist so geschrieben, dass er in der übergeordneten Umgebung zyklisch aufgerufen und abgefragt werden kann *Polling*.

Es wurde weiterhin die Installation des Algorithmus ausgehend von Matlab auf das Zieldevice gezeigt. Die Installationsroutine kann als Matlab Skript umgesetzt werden.

Der komplette Installationprozess ist graphisch in Abb. 3.3 nochmals aufgezeigt.

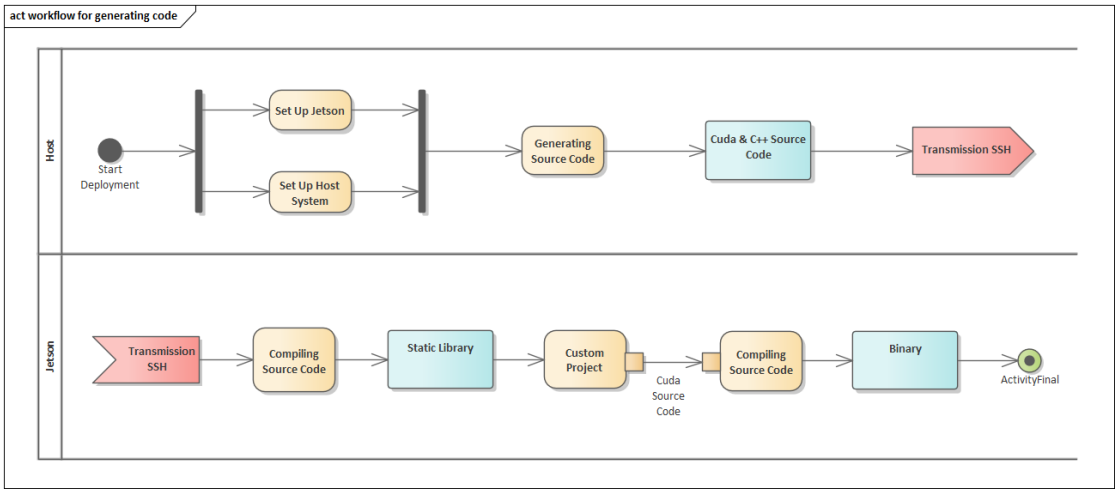



Abbildung 3.3: Die Zielhardware (Jetson) und der Host (f.i. WIN oder UNIX) muss eingerichtet werden. Die Reihenfolge der Einrichtung spielt keine Rolle. Der Quellcode wird im Host generiert und wird auf die Zielhardware übertragen. Im Jetson wird eine static Library generiert. Die Library wird in ein eigenes Cuda Projekt eingebunden und hieraus ein ausführbares Programm generiert.

# 4 Detektor & Filter

Bilder werden über ein Kameramodul aufgenommen, einem internen neuronalen Netzwerk übergeben und auf erkannte Objekte (Fahrzeuge) ausgewertet. Sind Objekte — oder auch keine Objekte — erkannt worden, findet eine Informationsweitergabe statt. Die Ergebnisse des Neuronalen Netzwerks und seinem unterliegendem Algorithmus (YoLo) werden der Verarbeitungspipeline übergeben.

Hintergedanken für diese Herangehensweisse:

- Weiterführende Datenaufbereitungen sind somit von "Detektor" Prozess entkoppelt
  - Neue Algorithmen können sehr einfach durch andere Algorithmen der "Daten-Pipeline" ersetzt oder hinzugefügt werden.
  - Abarbeitungsreihenfolgen der Algorithmen können beliebig neu angeordnet werden
  - Neue Algorithmen können in gewünschter Sprache implementiert und der Pipeline hinzugefügt werden
    - Somit keine Neukompilierung von eiem einzigen *monolitischen* Prozess mit NVidia Cuda Compiler (NVCC) notwendig

Ein digitaler Filter ist ein diskreter Algorithmus, eine Rechenvorschrift. Wenn hier von einem Filter gesprochen wird, ist ein Algorithmus in einem eigenen Ausführungsprozess gemeint. Werden mehrere Filter hintereinander ausgefuhrt, findet eine Pipeline-Verarbeitung statt[1]. Es existiert ein unidirektionaler Kommunikationskanal zwischen den Prozessen, d. h. die Kommunikationsrichtung fließt nur in eine Richtung.

Fragen, die hier behandelt werden, sind:

- Wie sind die Prozesse intern aufgebaut
- Wie sind deren Schnittstellen zur *Außenwelt* definiert
- Welche Daten werden übergeben

Möchte man eine Verbindung über Serielle Schnittstelle aufbauen, sind folgende Daten nötig:

- Passwort: 1111
- Login Name: jetson
- Baudrate: 115200

[1] https://www.elektronik-kompendium.de/sites/com/1705221.htm

## 4.1 Detector

Die Architektur ist Stufenweise aufgebaut. In Abb. 4.1 ist zu sehen, Softwareteile höherer Ordnung sind weiter oben angeordnet und deren Abhängigkeiten liegen tiefer und Verbindungslinien kennzeichnen die Abhängigkeiten.

Der oberste Block kennzeichnet den Hauptprozess. Die nächsten beiden Softwareblocke sind der *Command Line Parse* und die *Detect Funktion*. Der Command Line Parser wird beim ersten Prozessstart ausgeführt und ließt die in der Kommandozeile beim Starten des Prozess definierten Zusatzparameter und dekodiert diese, was die weitere Prozessabarbeitung beeinflußt. Bspw. wird der Prozess wie folgt:

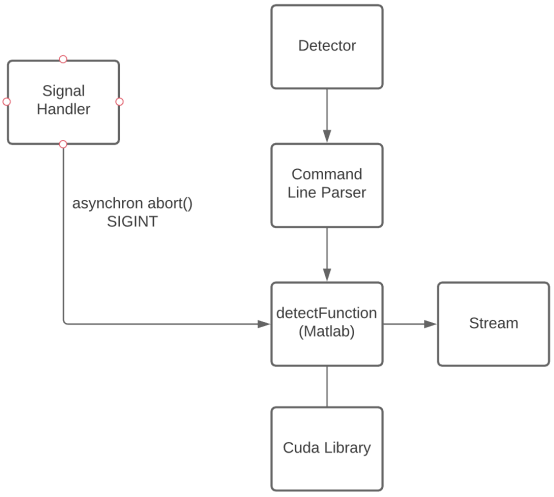



Abbildung 4.1: Softwareaufbau von Detector Hauptprozess

```
$ detector --visual-mode
```

gestartet, ließt der Parser das übergebene Flag und die Ausgabe des Prozesses ist für den Anwender in lesbarer Form. Im *visual mode* ist aber eine Prozessausführung innerhalb der Pipeline so nicht möglich. Der ausgehende Prozess muss Schnittstellenkonform mit dem nachfolgenden Prozess der Pipeline sein, was *visual mode* nicht erfüllt. Somit kann der Prozess in *visual mode* nur getrennt gestartet und betrachtet werden (z.B. Zeitmessung, wie schnell *FPS* die Objekterkennung arbeitet).

Die *Detect Funktion* ist der in Matlab generierte CUDA Algorithmus für die Objekterkennung. Der Algorithmus wird in einer Endlosschleife zyklisch ausgeführt. Die direkten Abhängikeiten des Algorithmus sind die CUDA Libraries. Vorteil von Matlab generierten Algorithmus ist: Softwareentwickler definiert den Algorithmus in Matlab-Code und Matlab produziert den CUDA Code, kompiliert ihn und erzeugt eine statische Library, die in ein unabhängiges Projekt eingefügt werden kann. Wie hier: Eingliederung CUDA Library in Detector Hauptprozess. Somit muss der Entwickler nicht direkt mit den CUDA Libraries arbeiten. Deren Aufrufe sind in der statischen Library *wegabstrahiert*. Ergebnis ist: Es kann weitgehend auf CUDA Code Syntax verzichtet werden und mit C++ respektive C Code Style gearbeitet werden, weil eine gemeinsame Schnittmenge mit CUDA Syntax vorhanden ist.

Der Hauptfunktionalität ist ein *Signal Handler* übergeordnet. Der Signal Handler reagiert auf System Signale und für diese, wofür er definiert wurde, unterbricht er den Hauptprozess und stellt sicher, dass die abhängige Hardware wie Kamera Modul sicher gelschloßen wird. Bspw. mit dem asynchronen Befehl:

```
$ <CTR+C> [SIGINT]
```

wird der Prozess unterbrochen und die Hardware und diverse Filedeskriptoren geschlossen. Vor dem Signal Handler traten Probleme auf bei Testdurchläufen, in denen das Programm mehrmals unterbrochen wurde, dass die Hardware nicht mehr reagierte. Die Hauptfunktionalität übergibt gewonnenen Daten dem *Output-Stream*.

Informationen werden mittels dem Betriebssystem bereitgestellten Pipelinesystem übertragen. Die Schnittstellendefinition zum nächsten Prozess der Pipeline ist daher jeweil der Outputstream. Dem übergeordneten Verarbeitungsprozess (Filter) wird mit einem definierten Signal mitgeteilt, dass ab diesem Zeitpunkt gültige Daten am Stream anliegen. Ein Codeausschnitt ist:

```
// 1ter Prozess der Pipeline
std::out << some_random_data;
my::flag(std::cout, 0xBADEAFFE);
std::out << valid_data;

////////////////////////////////

// 2ter Prozess der Pipeline
do_some_random_stuff();
wait(my::flag(std::cin, 0xBADEAFFE));
do_important_stuff();
```

Bash Skript starte bspw. beide Prozesse in einer Pipelinestruktur:

```
$ Detector | Filter1
```

Gibt *Detector* Daten vor dem Flag *0xBADEAFFE* aus, werden diese von *Filter1* nicht weiter bearbeitet. Erst wenn das Flag gelesen wurde, werden die nachfolgenden Daten des Streams als gültige Daten betrachtet und gehen in die Bearbeitung mit ein.

Bleibt nur noch zu klären, welche Daten der *Detector* Prozess der Pipeline übergibt.

## Datenstruktur

Das unterliegende Neuronale Netzwerk hat eine Eingangsgröße von:

$$[224\ 224\ 3].$$

Die *3* ist die Tiefe — Rot Gelb Blau (RGB) Farbraum. Das Kameramodul nimmt Bilder in *720p* Auflösung auf. Die Bilder müssen anschließend heruntergerechnet werden und werden dann dem Neuronalen Netzwerk übergeben. Um erkannte Objekte (Fahrzuege) werden *Bounding-Boxes* (Begrenzungskästen) gezogen. Sie signalisieren, wo im Bild das erkannte Objekt liegt. Kleinere Bounding-Boxes bedeuten, dass das erkannt Objekt weiter entfehrnt ist, wobei größere weiter weg bedueten. Sie können weiterhin weiter links in der Aufnahme liegen oder weiter rechts, sowie nach unten oder oben versetzt. Somit besitzen Bounding-Boxes mehrere Attribute:

- Position
  - X-Kord
  - Y-Kord
  - Height
  - Widht
- Konfidenz-Score

Das Positions-Attribute bezieht sich auf die linke obere Ecke der Bounding-Box relativ zur linken oberen Ecke der Aufnahme. Die Y-Koordinate wird nach unten abgetragen (Abb. 4.2). Der Konfidenz-Score gibt an, wie "sicher" sich das Netzwerk ist, ob das erkannte Objekt tatsächlich ein Fahrzeug ist oder nicht. Der Konfidenz-Wert kann in spätere Algorithmen verwendet werden. Wird kein Objekt in einer Aufnahme erkannt, nimmt der Konfidenz-Score einen Wert von -1 an.

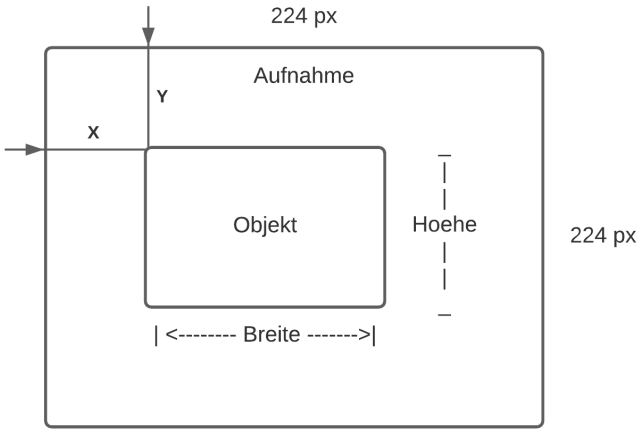



Abbildung 4.2: Aufnahme + Objekterkennung. Attribute: X, Y, Hohe, Breite

Daten, die über den Stream in die Pipelineverarbeitung miteingehen, sind:

- X-Kord
- Y-Kord
- Height
- Width
- Score

Wie oben beschrieben, ist eine Aufnahmen maximal $224 \cdot 224$ Bildpunkte tief. Somit gilt:

$$M := \{X, Y, Height, Width\}$$

$$\{x \in M \mid x \leq 224\}.$$

Das heißt, die benötigte Breite beträgt:

$$Bitbreite = \lceil ld(224) \rceil.$$

Der Score liegt zwischen: $0 \leq Score \leq 100$ und $Score < x \in M$. Zusammengefasst: Jedem Wert genügt eine Bitbreite von mindestens 8 Bit. Die gesamte Datenbreite ist: $5 \cdot 8Bit = 40Bit$. Die Reihenfolge der Daten wird wie folgt ausgegeben:

$$[X\ Y\ Height\ Width\ Score].$$

Wird kein Objekt (Fahrzeug) erkannt, wird eine Null-Folge versendet. Plausibilität: In einer Nullfolge ist jedes Bit 0. Somit auch die Höhe und Breite 0. Eine Bounding-Box mit Höhe und Breite 0 ist nicht vorhanden. Daraus folgt: Eine Nullfolge kennzeichnet "kein Objekt erkannt".

## Temperatur & Verarbeitungsgeschwindigkeit

Die Verarbeitungsgeschwindigkeit wird allgemein von der Hardware begrenzt. Im Speziellen hier von der Graphic Process Unit (GPU). Die GPU ist aus Halbleiterstrukturen gefertiegt. Es gilt: Je wärmer das Bauteil wird, desto größer die Leitfähigkeit der inneren Schalter, desto größer die Verlustleistungen und das resultiert wiederum in mehr produzierte Abwärme, was wiederum zum Anfang der Argumentationskette führt. Eine Konsequenz daraus ist, dass die Wärme abgeführt werden muss. Wird sie nicht in einem ausreichenden Maße abgeführt, drosselt sich die GPU. Im gedrosselten Zustand produziert sie weniger Abwärme. Die GPU drosselt sich soweit herunter, bis sich ein Gleichgewicht einstellt zwischen produzierter und abgeführter Wärme.

Der Jetson Nano hat von Werk aus ein Passiv-Kühlsystem — siehe Abb. 2.1. Betriebsarten des Jetson sind:

- X Server
- Terminal

Der X-Server startet beim Hochfahren des Betriebssystem die graphische Oberfläche — wie normaler Betrieb in bspw. Windows 10. Wurde in dieser Betriebsart der Objekterkennungsprozess angestoßen, kam nach wenigen Augenblicken eine Mitteilung, dass die Temperatur unzulässig hoch sei und deshalb die GPU gedrosselt werde. Diese Meldung wird nicht im Terminal Mode angezeigt, expliziet nur im X Server Betrieb.

Als Gegenmaßnahme ist eine Aktiv-Kühlung verbaut worden. Eine Besonderheit ist: Die Aktiv-Kühlung muss vom Anwender "angestoßen" werden. Befehle sind aus [bgu] entnommen. Die wichtigsten sind:

- `$ sudo /usr/bin/jetson_clocks`
  - Setzt CPU, GPU auf FULL POWAH
- `$ sudo sh -c 'echo 255 > /sys/devices/pwm-fan/target_pwm'`

| | Passiv Kühlung | Aktiev Kühlung |
|---|---|---|
| [FPS] | 8–9 | 8–9.5 |

Tabelle 4.1: Messung: Aktiv-/ vs. Passivkühlung; Zeitintervall: 60$s$

  - Aktiviert Fan

- `$ sudo sh -c 'echo 0 > /ysy/devices/pwm-fan/target_pwm'`
  - Deaktiviert Fan

Die Ausführungsgeschwindigkeit mit der Passiv-Kühlung betrug im Schnitt ca. 8–9 FPS. Mit Aktiv-Kühlung lagen die Werte auf ca. 8–9.5 FPS.

In Tabelle 4.1 sind beide Kühlarten gegenübergestellt. Die Messungen wurden jeweils in einem Zeitintervall von 60 Sekunden und dem selben Bild durchgeführt. Es ist somit zweifelhaft, ob die Aktivkühlung einen Mehrwert bietet. Wenn es gegen den erhöten Energiebedarf des Lüfters gegengerechnet wird, ist die Passivkühlung die bessere Wahl.

**Achtung!!!**
Es hat sich herausgestellt, sobal Lüfter aktiviert wurde, "fror" das System nach kurzer Zeit ein. Aktueller Stand: Auf aktive Kühlung verzichten. Aber Nachteil verbleibt: Meldung, wenn Objekterkennung aktiv, dass Command Prozess Unit (CPU) und GPU heruntertaktet.

## 4.2 Filter 1 — Transition

Dieser Prozess hat die Aufgabe, die empfangenen Inofrmationen des Detectors zu verarbeiten. Die Grundidee ist, nicht jede Empfangene Nachricht vom Detector unbehandlet durchzulassen. Erst wenn signifikante Änderungen auftreten, soll dem übergeordneten Prozess (bspw. dem Prozess, verantwortlich für die CAN-Übertragung) eine Mitteilung gemacht werden.

Das System soll mit Zuständen modelliert werden. Drei Zustände sind definiert: *Warten*, *Objekt* und *Velocity*. Der Wartezustand wird gleich nach dem Prozessstart eingenommen. Dort wird so lange verharrt, bis ein Objekt erkannt wurde. Ist ein Objekt erkannt worden, soll eine Message ausgegeben werden. Allgemein gilt: Alle Messages, die der gesamte Prozess erzeugt, werden der Verarbeitungspipeline übergeben.

Jetzt befindet man sich im *Objekt-State*. Jetzt soll es wiederum Bedingungen geben, in denen Aktionen definiert sind. Alle Aktionen beziehen sich impliziet auf das erkannte Objekt.

**Objekt-State**:

- Objektdistanz $> X_{Tresh}$ — No Message
  - Erkanntes Objekt liegt über dem Grenzwert $X_{Tresh}$, d. h. Objekt ist nich zu nah, sondern weit "genug" entfehrnt. In diesem Fall muss nur noch die Positionsänderung bezogen auf die Zeit untersucht werden. Diese wird im *Velocity-State* durchgeführt $\Rightarrow$ Übergang in nächsten State.

- Objektdistanz $\leq X_{Tresh}$ — Message
  - Erkanntes Objekt unterschreitet kritischen Wert $\Rightarrow$ Objekt zu nah.

Nur wenn das erkannte Objekt weit genung entfernt ist, wird in den nächsten State (*Velocity-State*) gewächselt, in dem die Positionsänderung bezogen auf die Zeit untersucht wird. Ändert sich die Position zu schnell, wird eine Message ausgegeben. Liegt die Positionsänderung unter einem kritischen Wert, wird auf die Message verzichtet. Aber in beiden Fällen wird wieder nach der Untersuchung in den *Objekt-State* zurückgegangen.

**Velocity-State**:

- Objekt-Position ändert sich nicht signifikant: $\frac{\partial x}{\partial t} + \frac{\partial y}{\partial t} \leq V_{Tresh}$ — Keine Message
- Objekt-Position änder sich signifikant: $\frac{\partial x}{\partial t} + \frac{\partial y}{\partial t} > V_{Tresh}$ — Message

Das beschriebene Vorgehen ist in Abb. 4.3 zu sehen.

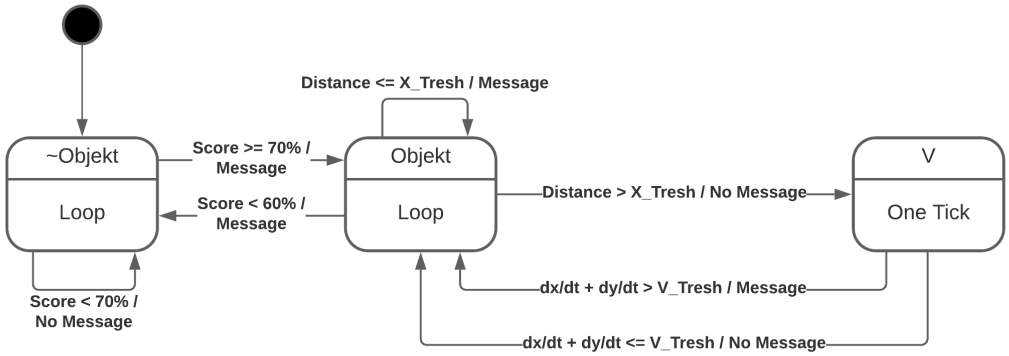



Abbildung 4.3: Filterung Objekt Erkennung: Filter1 — Transmission

Wenn vom *Wait-State* in den *Objekt-State* gewechselt wird, ist die Bedingung: Confidence Score $\geq$ 70% erfüllt. Das soll *Jittering* verhindern. Als *Jitter* wird eine Signalschwankung bezeichnet. Ein häufiges "Hin- und Herpendeln" des Signals am Grenzwert verursacht ein genauso häufiges "Hin- und Herpendeln" der Wirkung. In der Hardware wird hierfür ein *Schmitt-Trigger* verwendet, der diesen of unerwünschten Effekt verhindert. Hier hat es folgenden Hintergrund: Der Detector generiert ein Signal, wenn die Confidence $\geq$ 50% ist. Wird nun bei 50% direkt in die Objekt-Erkennung gewechselt, und im darauffolgenden Messvorgang des Detectors (zur Erinnerung: ca. 10 FPS) wird das Objekt nicht mehr erkannt[2], wird die nachfolgende Pipeline mit Nachrichten der Art: Objekt erkannt, Objekt verschwunden — überschwemmt.

[2]Entweder weil einfach gesehen das Objekt verschwunden ist (guter Fall) oder weil das Objekt an der Grenze ist, ob als Objekt erkannt oder nicht. Es kommt auch vor, dass sich das Netzwerk tauescht. Bspw. eine Orange ist kein Fahrzeug, kann aber unter besonderen Bedingungen als solches gehalten werden. In diesem Fall wäre aber ein niedriger Confidence Score wünschenswert. Gegenmaßnahmen wären bswp. Netzwerk mit erweiterten Datensätzen trainieren

## CAN Dispatcher

Der CAN Dispatcher ist das letzte Glied der Verarbeitungspipeline. Es ist logisch, da im Endeffekt eine Informationsweitergabe über den CAN Bus angedacht ist. Der Empfänger der CAN Nachricht ist im Projekt der Rasperry Pi. Dort findet die übergeordnete Informationsbearbeitung statt.

Die Aufgabe des CAN Dispatchers ist es nun, eine CAN Nachriht zu versenden. Dieser Prozess wurde noch **nicht** implementiert. Im nächsten Kapitel wird zusammengefasst, warum und welche Arbeiten hierfür noch nötig sind.

Punkte, die für die Aufteilung eines seperaten getrennten Prozesses für die Versendung sind:

- Versendung Informationenn über CAN komplett entkoppelt von anderen Prozessen/Aufgaben der Verarbeitungspipeline
- Der Algorithmus kann in einer beliebigen Sprache geschrieben werden (da komplett von anderen Prozessen entkoppelt)
- Von NVIDIA geschriebene Skripte bspw. in Python können somit problemlos verwendet werden, wenn sich für Python als Sprache entschieden wird

Das was zu beachten ist: Der erstellte Prozess muss in das Bash-Skript noch eingefügt werden.

## Bash Skript — Pipeline

Sind alle gewünschten Prozesse definiert und bereitgestellt worden, können sie in die Verarbeitungspipeline hinzugefügt werden. Dafür ist ein Bash-Skript angelegt/vorgesehen worden folgender Form:

```
#!bin/bash
## Definition Signal Handler
Detector | Transition | ... | CAN_Dispatcher
```

Es kann gesehen werden, dass weitere Prozesse in die Pipeline hinzugefügt werden weiter können. Bei aller Flexibilität, was diese Herangehensweiße bereitstellt, gibt es dennoch ein evtl. Nachteil. Ob dieser für die Problemstellung als groß angesehen werden kann, entscheidet der konkret vorliegende Fall.

Da alle Prozesse der Reihe nach ausgeführt werden, entsteht so eine Totzeit. Die Totzeit ist abhängig von der Rechengeschwindigkeit, mit der der Prozessor taktet. Ist die Totzeit im Verhältniss gegenüber der im System größten Zeitkonstante klein, dann diese vernachlässigt werden. Hier ist die größte Zeitkonstante die Verarbeitungsgeschwindigkeit der Objekterkennung (ca. 10 FPS). Der Prozessor Taktet mit mehreren *GHz* Takt[3]. Wenn sehr großzügig geschätzt wird, kann mit einem Mindesttakt von grob 1 GHz gerechnet werden. Der Kehrwert ist die Verstrichene Zeit pro Rechenoperation. Da verbauter Prozessor ARM Prozessor

[3]Variable Frequenz. Siehe Datenbaltt. Grundtakt ca. 1.8 GHz

ist und dieser wiederrum Reduced Instruction Set Computer (RISC) Architektur verwendet, kann mit einer schnelle Befehlsabarbeitung von ca. 1 Assembler Befehl pro Takt (Ziel von RISC Architektur) gerechnet werden. Jetzt kommt es an, wie viel Code den Prozessen zu Grunde liegt und schlußendlich, wie viele Prozesse in der Pipeline definiert sind.

Ein Nachteil einer Hochsprache wie C oder C++ ist, dass man nicht so einfach die Assembler Befehle zählen kann, die der Compiler produziert (starke Optimierung ⇒ für Menschen sehr schwer lesbarer Code). Aber wenn im "Hauptabarbeitungspfad" — d. h. dort, wo hauptsächlich die Rechenzeit aufgewendet wird — effizient programmiert wird, kann davon ausgegangen werden, dass der Prozessor im Vergleich zu den $\frac{1}{10}s$ (benötigte Zeit Objekterkennung) den Verarbeitungsprozess nicht allzu stark beeinflusst.

Fazit:

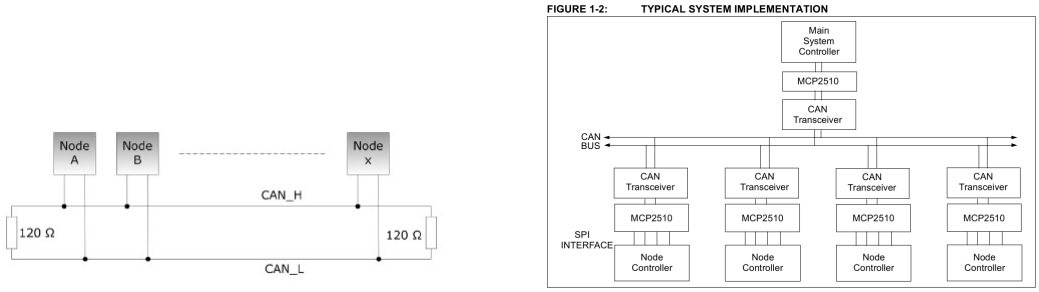
- Code Komplexität so gering wie möglich, aber so viel wie nötig halten. Vor allem in den Pfaden, in denen angenommen werden kann, dass dort ein Großteil der Rechenzeit aufgewendet wird.
- Anologe Überlegung auch auf die Verarbeitungspipeline: So wenig wie möglich, aber so viel wie nötig

# 5 Fazit

Es wird ein Überblick über die CAN Versendung gegeben. Danach wird ein kurzer Rückblick gegeben, was im Projekt umgesetzt wurde. Am Ende werden die Punkte angesprochen, die noch offen sind oder im Laufe der Arbeit aufgefallen sind.

## 5.1 CAN Transmission

In einem CAN Netzwerk teilen sich die einzelnen Teilnehmer den gesamten Bus — Abb. 5.1 (a). Ein Teilnehmer wird als CAN-Node bezeichnet.




(a) (b)

Abbildung 5.1: CAN Netzwerk; (a): Netzwerk Overview (b): Konkrete Umsetzung mit SPI/CAN Controller

Im autonomen Fahren Projekt kommunizieren die einzelnen Teile (Jetson Nano, Raspberry Pie, Arduino Due und VESC Motor-Controller) über CAN. Deshalb ist es letztendlich das Ziel, die Information über erkannte Objekte den anderen Teilnehmern — konkret: Raspberry Pie — zu übergeben.

Der Jetson Nano besitzt keinen eigenen CAN-Controller. Die Kommunikation muss deshalb über Umwegen realisiert werden. Als Baustein für die Realisierung SPI nach CAN gibt es extra Zusatzhardware, die zwischen Main Controller (SPI) und CAN-Transceiver zwischengeschalten wird. Beispielhafte Umsetzung mit einem konkreten Baustein *2510* von Microchip ist in Abb. 5.1 (b) zu sehen. Die CAN-Node beinhaltet jetzt den Main Controller (Jetson Nano), SPI-To-CAN Baustein und den CAN-Transceiver.

Die CAN Kommunikation wird über den SPI-Controller des Jetson Nano "angestossen". Die User-Software muss als Interface den SPI-Controller verwenden.

Im Linux-Betriebssystem (Jetson Nano: Ubuntu) kann nicht direkt auf die unterliegende Hardware zugegriffen werden. Der Zugriff erfolgt über Treibermodule. Diese müssen vor Verwendung geladen werden, wie in Abb. 5.2 dargestellt ist.

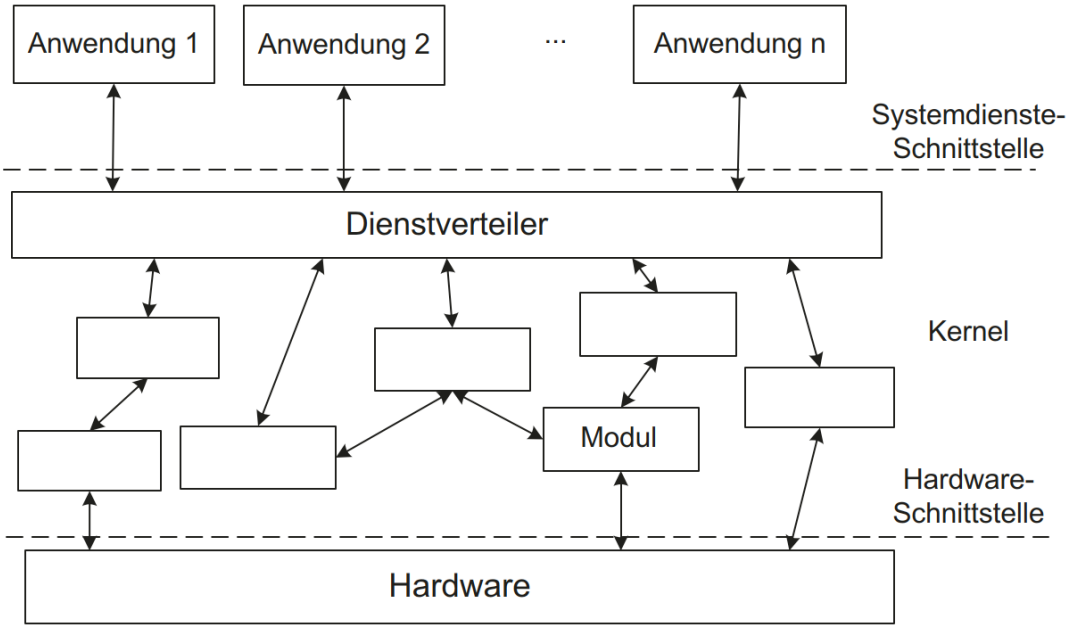



Abbildung 5.2: Hardware Dienstverteiler; Hardwarezugriff erfolgt nur über Kernelmodule. Die Kernelmodule müssen davor geladen werden.

Was anfangs zu prüfen ist, ob schon Module verfügbar sind, bspw. von NVIDIA, die den Zugriff auf den SPI-Controller bereitstellen.

## 5.2 Zusammenfassung

Es wurde einen Algorithmus für die Objekterkennung in Matlab geschrieben. Der Algorithmus verwendet ein CNN (YoLo-Variante) als Abhängigkeit. Das CNN übernimmt die Objekterkennung und wurde von Vorgängerarbeiten bereitgestellt.

Der Algorithmus wurde als statische CUDA-Library auf dem Zielsystem (Jetson Nano) kompiliert. Die Library wurde dann in einem getrennten Programm als Abhängigkeit eingebunden. Diese Vorangehensweise, Matlab Algorithmus als statische Library zu kompilieren und im Zielsystem als statische Abhängigkeit einzubinden, bietet den Vroteil:

- Änderungen im Objekterkennungsprozess sind unabhängig vom Host-PC
  - Auf Host-PC wurde der Algorithmus in der Matlab-IDE erstellt
  - Somit entfallen langwierige Übertragungs- und Kompilierungsprozesse, -/zeiten
    - Übertragung von Matlab-Code muss via LAN übertragen werden
    - Starten von “geschlossenen” Matlabalgorithmen werden von der Matlab-IDE im Host-PC “angestossen”
- Das Starten von Algorithmen kann automatisch über ein Skript “angestossen” werden

Es wurde eine Verarbeitungspipeline vorgestellt. Der Objekterkennungsprozess ermittelt Objekte in den Liveaufnahmen. Weitere "Filterungen" werden in anderen, getrennten Prozessen durchgeführt. Bspw. wurde eine Idee vorgestellt, nicht für jeden aufgenommenen Frame immer das selbe erkannte Objekt zu" übertragen, dass nur unter bestimmten Bedingungen neue Nachrichten für das selbe Objekt übertragen werden.

Es wurde weiter ausgeführt, dass andere Algorithmen die Pipeline einfach erweitern, oder andere entfehrnt werden können. Der letzte Prozess der Verarbeitungspipeline ist für die Versendung der entgühltigen Nachrichten vorgesehen. Allgemein können weitere Prozesse der Pipeline in beliebigen Sprachen geschrieben werden. So kann auch der letzte Prozess, der die Nachricht schlussendlich über CAN versendet, bspw. in Python geschrieben werden. Von NVIDIA sind zahlreiche Skripte und Libraries in Python bereitgestellt.

Zuletzt ist erläutert worden, wie die CAN Kommunikation konkret in Hardware umgesetzt werden kann.

## 5.3 Ausblick

Dinge, die noch angegangen werden müssen oder Überlegungen, die sich lohnen:

- Unterliegendes CNN von binäre Erkennung von Klassen auf Erkennung mehrerer Klassen, bspw. Auto, LKW, Fahrrad, Fußgänger, ...
- Prüfen unterliegendes CNN bessere Ergebnisse, wenn nur Graustufen zur Objekterkennung wirken
- Algorithmus bereitstellen, der mehr als nur ein Objekt im Frame erkennt und diese mehreren Objekte entsprechend in weiteren Algorithmen behandeln (Filter in Bearbeitungspipeline modifizieren, weitere erstellen, ...)
- Untersuchung Kühlsystem auf Systemstabilität
- Umsetzung CAN Kommunikation
- Bash-Skript für automaitsche Ausführung in Auto-Start-Folder hinterlegen. Somit entfällt das manuelle Starten, wenn System gestartet wird und Anwender braucht keine Remote-Verbindung herstellen und Objekterkennungsprozess händisch starten.

# Literaturverzeichnis


[bgu] BGULLA:
*Jetson Nano Cheat-Sheet.*
http://gist.github.com/bgulla/5d7afdb6575e8ef0260b7ab0507b014b, . –
Last Accessed: 2021-04-27

[Bos20] BOSL, A.:
*Einführung in MATLAB/Simulink: Berechnung, Programmierung, Simulation.*
Carl Hanser Verlag GmbH & Company KG, 2020 https://books.google.de/books?id=SVoAEAAAQBAJ. –
ISBN 9783446465466

[BS20] BRIEM, Stefan ; STROHMAIER, Lukas:
*Objekterkennung mit KNN*, Hochschule Aalen, Diplomarbeit, 2020

[Nvi01a] NVIDIA:
*JETSON NANO.*
http://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-nano/education-projects/, 2020-09-01. –
Last Accessed: 2021-04-14

[Nvi01b] NVIDIA:
*JETSON NANO - Getting Started.*
http://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit, 2020-09-01. –
Last Accessed: 2021-04-14