

Gesten- und Objekterkennung durch schwache FPGAs in autonomen Fahrzeugen mittels neuronaler Netze

Projektbericht

**Philipp Altnickel
Tuncer Catalkaya
Viktor Chechulin
Toni Dragojevic
Felix Müller
Tobias Nießen
Jonas Philipp
Ismail Sastim
Lukas Schimanski
Johannes Steffen
Fannese Manuela Tchanga Seunga
Mattia Uhlenbrock
Stanislav Voytas
Reena Wichmann
Prof. Dr. Jan Bredereke**



Bremen University of Applied Sciences
Fakultät 4: Elektrotechnik und Informatik
12. April 2021 – 01. September 2021

Zusammenfassung

Geschrieben von Felix Müller

Künstliche neuronale Netze versprechen Problemstellungen, die mit manuell programmierter Software nur schwer bewältigbar sind, effizienter und genauer zu lösen. Besonders in einem Teilgebiet der Bildverarbeitung, der Objekterkennung in Bildern, wurden in den letzten Jahren Ergebnisse erreicht, die klassische Bildsegmentierungsmethoden in den Schatten stellen. Zur Objekterkennung werden größtenteils gefaltete Netze verwendet, deren Verarbeitung sehr aufwendig, aber auch sehr parallelisierbar, ist. Diese Parallelität kann besonders mit speziell entwickelter Hardware ausgenutzt werden. Hier bieten sich FPGAs durch ihre Rekonfigurierbarkeit, Effizienz und Verfügbarkeit für spezielle Bereiche, wie dem Weltraum, an. Ein Framework, welches vom Potential neuronaler Netze auf FPGAs Gebrauch macht, ist das FINN. Diese Projektarbeit verwendete das genannte Framework, um auf einem Digilent PYNQ-Z1 mit Zynq SoC einen realen Anwendungsfall umzusetzen. Hier wurde ein mit einer Kamera ausgestattetes autonomes Modellfahrzeug gewählt, welches zwei Aufgaben bewältigen soll: Personenverfolgung durch Lokalisierung im Bild und Gestenerkennung zum Starten/Stoppen des Fahrzeuges. Zu diesem Zwecke wurde ein eigener Datensatz mit drei unterschiedlichen Gesten entwickelt und mit einem angepassten Netz der FINN-Forschungsgruppe trainiert und folgend in eine Hardwareschaltung für das FPGA umgewandelt. Auf der CPU des SoC wurde eine Anwendung entwickelt, die Bilddaten einer USB-Kamera vorbereitet und das Netz auf dem FPGA entsprechend ansteuert. Die Umsetzung vom Datensatz bis zur Ausführung auf der Hardware wurde erfolgreich bewältigt und hat in der Evaluation erste vielversprechende Ergebnisse gezeigt. Mit der erreichten Genauigkeit des Netzes lassen sich Gesten auch mit realen Bilddaten einer Kamera grundsätzlich unterscheiden, sodass eine Basis für weitere Verbesserungen in der Fortführung des Projektes geschaffen werden konnte.

Inhaltsverzeichnis

1	Einleitung	7
1.1	Forschungskontext	7
1.2	Projektziele	7
2	Grundlagen	9
2.1	Objekterkennung	9
2.2	Neuronale Netze	9
2.3	Convolutional Neural Networks (CNN)	10
2.3.1	Intersection over Union	15
2.3.2	Bounding Box	15
2.3.3	Average Precision	15
2.3.4	Mean Average Precision	16
2.3.5	Inferenz	16
2.3.6	Bewährte CNN-Modelle zum Klassifizieren von Bildern	16
2.3.7	Region-based Convolutional Neural Network	20
2.3.8	YOLO	27
2.3.9	Single Shot Detector	28
2.4	Binary Neural Networks (BNN)	30
2.4.1	Forward Propagation	31
2.4.2	Backpropagation in BNNs	33
2.4.3	Optimierung	34
2.4.4	Erkenntnisse aus BNN Entwicklung	36
2.4.5	Erkenntnisse aus BNN Implementierung auf FPGA	38
2.5	CNN mit Tensorflow	38
2.6	Brevitas	39
2.7	ONNX	39
2.8	FINN	40
2.9	TinyML	42
2.10	AXI Bus	42
2.10.1	Was ist AXI4?	42
2.10.2	Datenübertragung in AXI4	43
2.10.3	AXI4-Lite	44
2.10.4	AXI4-Stream	44
2.11	Vorhandene Hardware	45
2.11.1	Digilent PYNQ-Z1	45
2.11.2	Fahrzeug	46
2.12	Vorgängerprojekt	47
3	Ausgewählter Anwendungsfall im Projektgebiet	48
3.1	Konzept zur Umsetzung der Projektziele	48
3.2	Konzept für die Kollaboration der Projektmitglieder	48
3.3	Reevaluation der Boardauswahl	49
4	Datensatz für Gestenerkennung	51
4.1	Wahl des Datensatzes	51
4.2	Wahl des Inhalts	52
4.3	Erstellung der Bilder	52
4.4	Umwandlung in einen PyTorch Datensatz	53
4.5	Wahl eines Modells (Variante 1)	54
4.6	Wahl eines Modells (Variante 2)	55
5	Auswahl eines Frameworks für KNN auf FPGAs	55
5.1	Vorhandene Frameworks zur Synthetisierung für FPGAs	55

5.2	Brevitas als Modellframework	58
6	Evaluation einer Umsetzung von Region Proposals oder Single Shot Detektoren auf FPGAs mit Brevitas	60
7	Training in der Cloud	61
7.1	Auswahl einer Cloud Plattform	61
7.2	Microsoft Azure Cloud	61
7.3	Cloud-Konfiguration und Kostenübersicht	62
7.4	Vergleich der Trainingszeit	63
8	Übersetzung der Netztopologien für das FPGA	64
9	Anpassung der Fahrzeugsteuerung des Vorgängerprojektes	67
9.1	Anbindung des neuronalen Netzes auf dem FPGA	67
9.1.1	Probleme und Vorgehen	67
9.1.2	Reverse Engineering	67
9.1.3	Datentransfer zwischen Processing System und neuronalem Netz	68
9.1.4	Datenformat	68
9.2	Modularisierung des Quellcodes	69
9.2.1	Controller	72
9.2.2	NetworkConnectionController	72
9.2.3	NetworkRxController	72
9.2.4	CameraController	73
9.2.5	TilingController	73
9.2.6	InferenceController	73
9.2.7	InferenceResultController	74
9.2.8	StateController	75
9.2.9	MotorController	78
9.2.10	PIDController	78
9.2.11	NetworkTxController	78
9.2.12	ConfigController	78
9.2.13	Logger	78
9.2.14	DMA	79
10	Anpassung der Base Station des Vorgängerprojektes	80
10.1	Anpassung für die Gestenerkennung	80
10.2	Anpassung der Base Station an den NetworkTxController	80
11	Evaluation	81
11.1	Genauigkeit der Netztopologie (Variante 1)	81
11.1.1	Evaluationstests mit Version 4 des Datensatzes	81
11.1.2	Evaluationstests mit Version 5 des Datensatzes	82
11.2	Evaluationstest mit Version 6 des Datensatzes	83
11.2.1	Auswertung Evaluation	83
11.3	Genauigkeit der Netztopologie (Variante 2)	83
11.4	Ressourcenverbrauch der erzeugten Schaltung	84
11.5	Systemevaluation	84
11.5.1	Testvideos	84
11.5.2	Testergebnisse	85
11.5.3	Versuche zur Optimierung	86
12	Ergebnisse und Ausblick	88
12.1	Ergebnisse	88
12.2	Ausblick	89

12.2.1	Zukünftige Testszenarien	90
12.2.2	Weitere Ideen für Zustandsdiagramme für die Fahrzeugsteuerung	91
Literatur		94
A	Abbildungsverzeichnis	101
B	Tabellenverzeichnis	103
C	Network Dataframes (Netzwerkdatenrahmen)	104
D	Anleitung: Installation und Nutzung des Gestendatensatzes	105
D.1	Installation	105
D.2	Nutzung	105
D.2.1	Struktur	105
D.2.2	Variationen des Datensatzes erzeugen	105
D.2.3	Ausschneiden von Bildern	105
E	Anleitung: Trainieren des neuronalen Netzes	107
E.1	Clonen der benötigten Git Repositories	107
E.2	Train.ipynb Prepare Environment	107
E.3	Train.ipynb Training	108
E.4	Train.ipynb Measure Accuracy	108
E.5	Train.ipynb Finn Export	108
F	Leitfaden zur Erstellung von Datensatz-Bildern	109
F.1	Vorgaben	109
F.1.1	Posen	109
F.1.2	Konstante Parameter	110
F.1.3	Variable Parameter	110
F.2	Ergebnisse hochladen	111
G	Anleitung: Installation von FINN, Vitis (Vivado) und Xilinx-Runtime (Xrt) in Ubuntu	112
G.1	Ubuntu 18.04.5 LTS	113
G.1.1	Vitis (und Vivado) installieren	113
G.1.2	Optional: Nvidia Toolkit mit angepasstem Docker	114
G.1.3	Docker installieren	116
G.1.4	Xrt installieren	116
G.1.5	FINN-Framework installieren	116
G.1.6	FINN starten	118
G.2	Ubuntu 20.04.2.0 LTS	120
G.2.1	Vitis (und Vivado) installieren	120
G.2.2	Docker installieren	122
G.2.3	Xrt installieren	122
G.2.4	FINN-Framework installieren	123
G.2.5	FINN starten	125
G.3	Sonstiges / Mögliche Fehler	127
H	Anleitung: Synthese bei Änderung der FPGA-Hardware	128
I	Anleitung: Inbetriebnahme Pynq-Z1 und Installation vom Cross-Compiler	130
I.1	Cross-Compiler	131
I.2	Kompilieren direkt auf dem Board	132
J	Anleitung: Übersetzung der Netztopologie für das FPGA	133

K Training in Microsoft Azure	141
L Anleitung: Installation der Basisstation	152

1 Einleitung

Geschrieben von Jan Brederke

Dieses Projekt erarbeitet Lösungen, wie man neuronale Netze möglichst effizient auf vergleichsweise leistungsbegrenzten FPGAs ausführen kann, um solche Netze auch auf autonomen Vehikeln nutzen zu können. Diese Aufgabe umfasst unter anderem die Bereiche feldprogrammierbare Digitalerschaltung (FPGA), System-on-Chip (SoC) sowie optische Objekterkennung mit neuronalen Netzen, und dazu auch etwas Regelungstechnik für autonomes Fahren und ein wenig Elektrik und Mechanik des Vehikels.

1.1 Forschungskontext

Geschrieben von Jan Brederke

Es ist eine aktuelle Forschungsfrage, wie man den vielseitigen Nutzen neuronaler Netze nicht nur in Rechenzentren mit leistungstarker und stromhungriger Spezialhardware, sondern auch am Rande der Cloud („Edge“), d.h. nahe bei den Sensoren und Aktoren, oder gar autonom von Daten- und Stromversorgungsverbindungen, ausschöpfen kann. Die CPU eines Mikrocontrollers ist dafür zu rechenschwach. Mehr Leistung, sowohl absolut als auch pro Stromverbrauch, verspricht der Einsatz eines feldprogrammierbaren Gate-Arrays (FPGA). Grundsätzlich ist ein FPGA für die hochparallele Struktur eines neuronalen Netzes sehr gut geeignet. In der Praxis ergeben sich aber viele Optimierungsaufgaben, die erst gelöst werden müssen, bevor das FPGA diesen Vorteil voll ausspielen kann.

Die konkrete Motivation für das Projekt entstammt der Raumfahrt. Auf Bordrechnern ist besonders wenig Rechenleistung verfügbar, und eine Verbindung zu einer Bodenstation ist meist nur zeitweise gegeben. Aufgrund der hohen Belastung durch Weltraumstrahlung würden übliche heutige Prozessoren sehr schnell ausfallen. Daher verwendet man Spezialrechner, deren Chips Strukturbreiten von mindestens 65 nm aufweisen. Diese Spezialrechner sind robust genug, aber entsprechend weniger leistungsfähig als solche, die mit aktuellen 16 nm oder 10 nm hergestellt sind. Aufgrund der äußerst geringen Stückzahlen dieser Art von Rechnern werden sie oft nicht mit speziell entwickelten Chips (ASICs), sondern mit programmierbarer Standard-Hardware (FPGAs) realisiert. Strahlungsfeste Versionen bestimmter FPGAs mit entsprechend großer Strukturbreite sind hierfür verfügbar. Zunehmend besteht Bedarf an mehr Onboard-Rechenleistung, zum Beispiel für Bildverarbeitung an Bord, etwa für autonome Rover auf anderen Himmelskörpern oder für Schwärme von Kleinsatelliten mit jeweils nur wenig Bandbreite zu einer Bodenstation.

1.2 Projektziele

Geschrieben von Jan Brederke

Als konkretes Vehikel in unserem Projekt bauen wir auf einem autonomen Modellauto auf (siehe Abb. 1 auf der nächsten Seite), das als Ergebnis von zwei Vorgängerprojekten [Mül+21a; Hut+20] entwickelt wurde und das als Stellvertreter für ein autonomes Raumfahrzeug dient. Es ist mit einer Kamera und einem SoC Zync-7020 ausgerüstet. Das SoC enthält außer einer Arm-CPU auch ein FPGA Artix-7, dessen Leistungsfähigkeit recht gut einem weltraumtauglichen FPGA entspricht.

Das Optimieren der resultierenden Rechenleistung ist in vielerlei Hinsicht möglich. Das Verwenden eines FPGAs wurde bereits genannt. Weiterhin benötigt das Ausführen eines bereits trainierten neuronalen Netzes („Inferenz“) Größenordnungen weniger an Rechenleistung, als es zu trainieren. Indem das Trainieren vorab (d.h. bei Raumfahrtaufgaben am Boden) erledigt wird, rücken jedenfalls einige Anwendungen der Bildverarbeitung in greifbarere Nähe. Quantisierte neuronale Netze sind ein Ansatz, bei dem die Daten mit weniger Bits als üblich (bis hinunter zu nur einem einzigen Bit) dargestellt werden. Dies kostet zwar Qualität des Ergebnisses, aber der Gewinn bei den Ressourcen ist so groß, dass das Reinvestieren eines Teils des Gewinns in mehr Knoten für das Netz reicht, um die Qualität wiederherzustellen.

Diese ersten Schritte sind bereits in den Vorgängerprojekten gegangen worden. Sie haben aber eine riesige

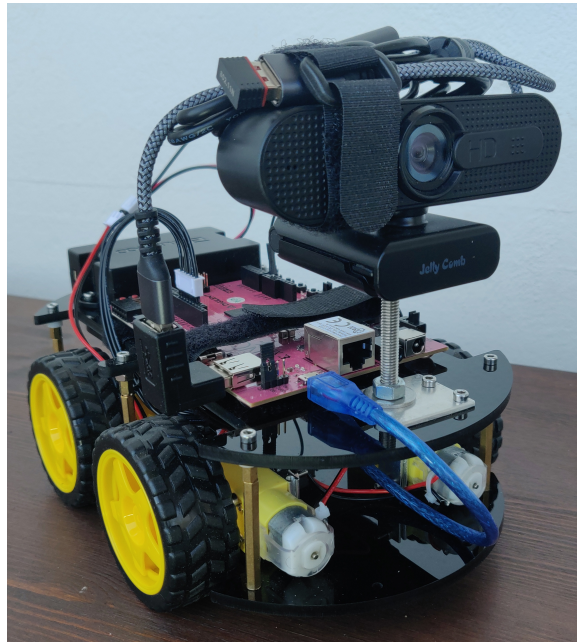


Abbildung 1: Das im Vorgängerprojekt realisierte Fahrzeug mit optischer Objekterkennung auf einem FPGA. Foto: Niklas Krekel

Palette an weiteren Möglichkeiten eröffnet. Der breitbandige Eingabedatenstrom muss effizient zum neuronalen Netz geführt werden, aus den Bildern des Videostroms müssen für das neuronale Netz effizient geeignete Bildausschnitte angefertigt werden (Tiling), und es sind viele übliche Optimierungstechniken für Digitalschaltungen wie z.B. Pipelining möglich. Eine Verbesserung um mehrere Größenordnungen ist insgesamt sicher noch möglich.

Zwar wurde die Machbarkeit bereits mit einem fahrenden Vehikel demonstriert, aber selbst grundlegende Aufgaben konnten bisher noch nicht angegangen werden. Anstatt vorab ein neuronales Netz zu trainieren, wurde nur ein fertig trainiertes Netz für eine Standardaufgabe (Verkehrsschilderkennung, nur für „Stop“-Schild) verwendet und die Navigationsaufgabe darauf angepasst. Und selbst hierbei ist die Größe des neuronalen Netzes weder besonders gut an die Aufgabe noch an die verfügbaren Ressourcen angepasst. Die Motorsteuerung hat erhebliche Abweichungen und benötigt wohl andere Motoren, eine andere Ansteuerung oder eine weitere Navigationshilfe wie etwa einen Kompass. Für das leichtere Aktualisieren der Schaltung auf dem Vehikel während Experimenten existieren viele noch nicht umgesetzte Ideen, ebenso für andere Debug-Hilfen.

Projektziel ist also, die Leistung des neuronalen Netzes unter den vorhandenen Hardware-Randbedingungen massiv zu steigern und damit entsprechend auch die Leistung des Vehikels beim autonomen Fahren zu verbessern. Projektergebnis soll ein besseres Verständnis für die vielen Optimierungsmöglichkeiten sein, sowohl auf Seiten der Digitaltechnik als auch auf Seiten der neuronalen Netze.

2 Grundlagen

Geschrieben von Felix Müller

Dieses Kapitel enthält die nötigen Grundlagen zum Verständnis der Projektarbeit. Als Erstes folgt eine Einführung in die Objekterkennung und wie diese mithilfe neuronaler Netze umgesetzt werden kann. In Zuge dessen werden passende Netztopologien und Optimierungen (für Hardware) vorgestellt. Nach einer Beschreibung weiterer Technologien, wie dem verwendeten Hardwarebus, und Frameworks wird die verwendete Hardware und zuletzt das Vorgängerprojekt vorgestellt.

2.1 Objekterkennung

Geschrieben von Tobias Nießen

Zur Steuerung des Fahrzeugs wird eine Objekterkennung benötigt, welche die Art und Position bekannter Objekte bestimmen kann. In der Fertigungstechnik wird die Objekterkennung häufig mittels Sensorik oder RFID-Tags realisiert, da die zu unterscheidenden Objekte hier klar definiert sind [ESC20]. Diese Sensorik erkennt beispielsweise die Farbe eines Werkstücks oder dessen Etikettes. Eine komplexere Anwendung stellt die Erkennung von Formen dar. Diese Formen werden in einem Programm mathematisch beschrieben und so unterschieden. In dem Anwendungsfall dieses Projektes sollen Menschen und zudem ebenfalls Gesten dieser Menschen erkannt werden, sodass mit diesen Gesten ein Fahrzeug gesteuert werden kann. Die Erkennung eines Menschen innerhalb unterschiedlichster Hintergründe ist für die mathematische Kantenerkennung mittels Filter bereits schwierig, da Menschen unterschiedlich sind. Wenn hier außerdem noch eine Gestenerkennung hinzukommt, wird die Wahrscheinlichkeit einer richtigen Detektion wesentlich geringer. Außerdem benötigen solche Filter eine hohe Rechenleistung, da für den gesamten Bildbereich Berechnungen durchgeführt werden müssen, was die für dieses Projekt geforderte Anwendung in Echtzeit erschwert. [Kim+12] [Lem+19]

Eine wesentlich weniger rechenintensive Möglichkeit ist die Verwendung neuronaler Netze, welche mittels verschiedener Optimierungsverfahren signifikant schneller sind, als traditionelle Filter. Hier können zum Beispiel nur bestimmte Gebiete eines Bildes genauer betrachtet werden, oder die geforderte Genauigkeit reduziert werden, um einen höheren Durchsatz zu erreichen. Außerdem sind neuronale Netze einfacher Anpassbar, wodurch neue Anwendungsszenarien besser umgesetzt werden können.

2.2 Neuronale Netze

Geschrieben von Fannese Tchang

Neuronale Netze beziehen sich auf das Neuronennetz des menschlichen Gehirns. Dieses dient als Analogie und Inspiration für in Computern simulierte Künstliche neuronale Netze. Diese Analogie steht bei heutigen Arbeiten zu neuronalen Netzen jedoch häufig nicht mehr im Vordergrund. neuronalen Netzen erhalten, verarbeiten und geben Informationen aus, wobei sich das Netz während der Verarbeitung umstrukturieren kann[Gün08].

Im Allgemeinen besteht Neuronale Netze aus mehreren Neuronen. Diese Neuronen werden auch als Units, Einheiten oder Knoten bezeichnet. Eine Input-Ebene, eine Versteckte Ebene mit einer beliebigen Anzahl von Neuronen zur Informationsverarbeitung und eine Output Ebene, zwischen denen mehrere Verbindungen bestehen. Das einzelne Neuron beinhaltet zumindest jeweils eine Aufsummierung des Inputs, einen Bias und eine Aktivierungsfunktion. Zunächst stellen die Verbindungen zwischen den Neuronen Gewichtungen dar. Im Lernprozess wird zunächst der Input verarbeitet, indem Informationen vom Input, über die Neuronen in den Output fließt(Forward Propagation). Danach werden die Werte der Gewichtungen und das Bias so angepasst, dass der Fehler zwischen erwarteter und tatsächlicher Ausgabe des Neuronalen Netzes minimiert wird. Dieses Vorgang nennt man Backpropagation[Gün08].

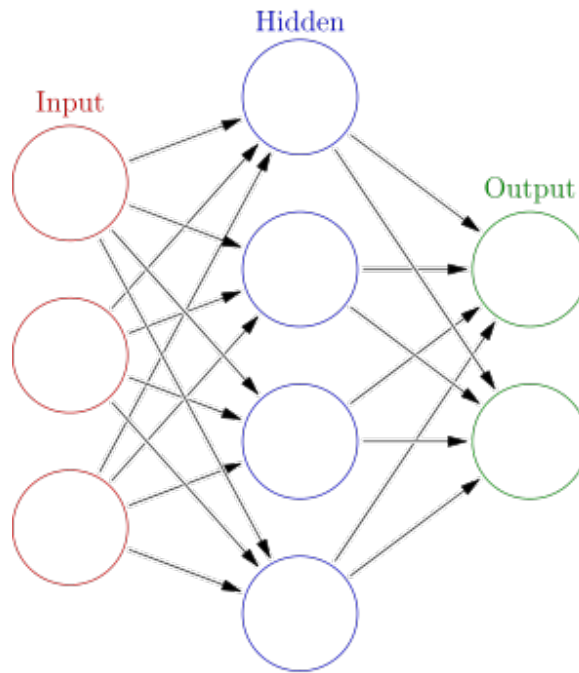


Abbildung 2: Der Aufbau eines Neural Network [IT-19]

Abb. 2 zeigt ein Modell eines künstlichen Neurons. Es gibt ebenfalls Eingaben, und je nach den Eingangssignalen überträgt das Neuron über eine Aktivierungsfunktion (der Hidden in Abb. 2) ein Ausgangssignal. Die Aktivierungsfunktion kann eine einfache Schwellenwertfunktion sein, wobei das Neuron ausgeschaltet ist und erst dann eine Ausgabe ausgibt, wenn die Summe der Eingaben einen Schwellenwert erreicht oder überschreitet. Ein Neuronales Netz lernt, indem die Gewichte der einzelnen Eingangssignale in das jeweilige Neuron angepasst werden [Gün08]. In Abb. 2 ist die Zeitablauf von Links nach rechts durch den roten Pfeil gekennzeichnet. Die mittlere Schicht wird als verdeckt bezeichnet, da sie im Gegensatz zu den Eingabe- und Ausgabeschichten keine direkte Verbindung zur Außenwelt aufweist [Gün08].

Unseres Thema befasst sich mit vielen Arten von Neuronalen Netzen, die in den unten stehenden Kapitel Erläutert werden.

2.3 Convolutional Neural Networks (CNN)

Geschrieben von Philipp Altnickel

CNNs sind eine spezielle Form der Neuronalen Netze. Sie sind besonders gut für die in diesem Projekt benötigte Objekterkennung geeignet. Durch ihre spezielle Architektur sind sie, im Vergleich zum klassischen Aufbau neuronaler Netze, besser dazu geeignet z.B. Bilddaten zu analysieren. Die Erkennung von Objekten oder Mustern, sowie deren Position wird sehr vereinfacht. Im Folgenden Abschnitt geht es nur um einen Überblick über die wesentlichen Elemente eines Convolutional Neural Networks. Um ein tiefergehendes Verständnis, auch aus mathematischer Sicht zu erlangen, ist es zu empfehlen einige Literatur heranzuziehen. Gut geeignet ist folgende Quelle, die auch als Grundlage für diesen Überblick diente: [Agg18], ab Seite 315.

Ein CNN zeichnet sich in seinem Basisaufbau im Wesentlichen durch drei verschiedene Arten von Ebenen aus. Die Eingangsdaten werden zunächst im Convolutional Layer mit Hilfe von Filtern auf bestimmte Formen (Patterns) untersucht. Mit diesen Filtern werden sogenannte Featuremaps erzeugt, in denen sichtbar wird, an welcher Stelle im Bild sich die jeweiligen Features befinden. Durch einen anschließenden

den Pooling-Layer kann die Größe der Maps und somit die Menge der zu verarbeitenden Daten deutlich reduziert werden. Hier finden Subsampling Operationen statt: Es wird immer ein gewisser Bereich von Pixeln zu einem Pixel mit dem Maximalwert des Bereiches (Max-Pooling) oder dessen Durchschnittswert (Average-Pooling) zusammengefasst. Durch die Verkleinerung gehen keine wichtigen Daten verloren, da die ungefähre Position der Features erhalten bleibt. Im Gegenteil: Es werden sogar Probleme wie Overfitting (Zu genaue Anpassung bestimmte Bilder, die dann wieder stark vom allgemeinen Vorbild abweichen) oder Rauschen minimiert. Die Kombination von Convolutional- und Poolinglayer kann beliebig oft wiederholt werden, so können zunächst einfache Formen, wie Linien oder Kreise erkannt werden. Danach wird auf die Featuremaps wieder ein Convolutional Layer angewendet und größere/komplexere Formen erkannt. Anschließend wird das Ergebnis auf eine eindimensionale Menge von Neuronen ausgerollt. Es folgt der Fully-Connected-Layer, hier wird die Klassifizierung und schlussendliche Erkennung der Objekte aus den vorher gefundenen Features in Form eines klassischen Neuronalen Netzes vorgenommen. Der große Vorteil des CNN Ansatzes ist, dass er die Verarbeitung großer Datenmengen, z.B. von Bildern erlaubt. Es muss die grundsätzliche Ebenenstruktur vorgegeben werden, danach kann das Netz komplett anhand von Eingabedaten trainiert werden. Die Backpropagation, also das anpassen der Parameter innerhalb des Netzes, funktioniert nicht nur im Fully Connected Layer, auch die Patterns in den Convolutional Layers können selbstständig erlernt werden. [Agg18]

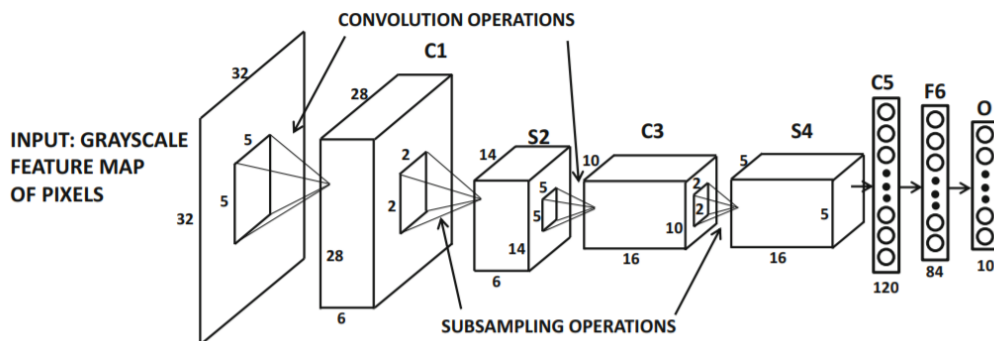


Abbildung 3: Der Basisaufbau eines CNN [Agg18]

Convolutional Layer Im Convolutional Layer geschieht der wesentliche „Trick“, warum diese Netzstruktur so viel effizienter arbeitet. Er besteht nämlich nicht aus klassischen Neuronen, die bei denen alle Gewichte und Kanten einzeln berechnet werden müssen. Auf der Eingangsseite steht die Matrix der Pixel des Eingangsbildes, bzw. steht diese Ebene an späterer Stelle erneut, befindet sich hier die vorher ermittelte Ergebnismatrix. In jedem Convolutional Layer gibt es eine Menge von Filter, die über die Eingangsmatrix gelegt werden. Jeder Filter hat die gleiche Größe und wird schrittweise, je um einen Pixel zur Seite, bzw. nach unten verschoben. Im Folgenden Beispiel wird eine 3x3 Filtermatrix verwendet, d.h. die 3x3 Matrix, die der Filter im Eingangsbild abdeckt, wird mit der Filtermatrix multipliziert, um einen Wert der Ausgangsmatrix zu erhalten:

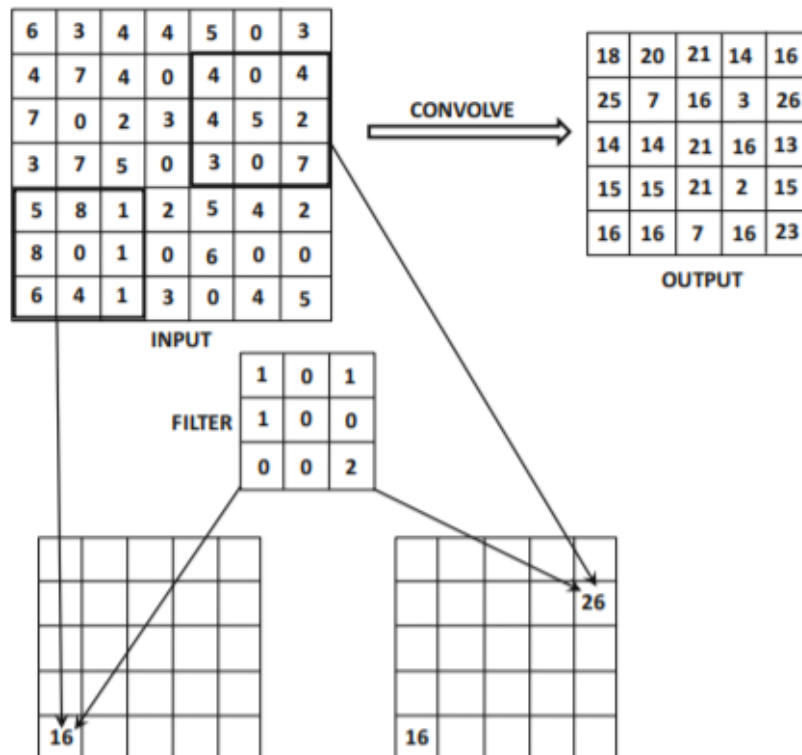


Abbildung 4: Anwendung von Filtern im Convolutional Layer [Agg18]

$$5 * 1 + 8 * 1 + 1 * 1 + 1 * 2 = 16 \quad (1)$$

$$4 * 1 + 4 * 1 + 4 * 1 + 7 * 2 = 26 \quad (2)$$

Geschrieben von Fannese Tchang

Wie oben bereits beschrieben werden wir Anhang anderen Beispiel zeigen wie eine Faltung verwendet wird. Dafür betrachten wir einen Filter der Größe 3*3 und ein Bild der Größe 5*5, führen eine Elementweise Multiplikation zwischen den Bildpixelwerten durch, die der Größe des Kerns und dem Kernel selbst entsprechen, und fassen sie Zusammen. dies liefert uns einen einzelnen Wert für die Feature-Zelle[Fur21].

$$2*1+4*2+9*3+2*(-4)+1*7+4*4+1*2+1*(-5)+2*1=51$$

$$Image = \begin{bmatrix} 2 & 4 & 9 & 1 & 4 \\ 2 & 1 & 4 & 4 & 6 \\ 1 & 1 & 2 & 9 & 2 \\ 7 & 3 & 5 & 1 & 3 \\ 2 & 3 & 4 & 8 & 5 \end{bmatrix} \quad Filter = \begin{bmatrix} 1 & 2 & 3 \\ -4 & 7 & 4 \\ 2 & -5 & 1 \end{bmatrix} \quad Feature = \begin{bmatrix} 51 \end{bmatrix}$$

Der Filter läuft weiter auf dem Bild und erzeugt neue Werte, unten gezeigt.

$$4*1+9*2+1*3+1*(-4)+4*7+4*4+1*2+2*(-5)+9*1=66$$

$$Image = \begin{bmatrix} 2 & 4 & 9 & 1 & 4 \\ 2 & 1 & 4 & 4 & 6 \\ 1 & 1 & 2 & 9 & 2 \\ 7 & 3 & 5 & 1 & 3 \\ 2 & 3 & 4 & 8 & 5 \end{bmatrix} \quad Filter = \begin{bmatrix} 1 & 2 & 3 \\ -4 & 7 & 4 \\ 2 & -5 & 1 \end{bmatrix} \quad Feature = \begin{bmatrix} 51 & 66 \end{bmatrix}$$

und am ende bekommen wir eine endgültige Faltungsoperation:

$$Image = \begin{bmatrix} 2 & 4 & 9 & 1 & 4 \\ 2 & 1 & 4 & 4 & 6 \\ 1 & 1 & 2 & 9 & 2 \\ 7 & 3 & 5 & 1 & 3 \\ 2 & 3 & 4 & 8 & 5 \end{bmatrix} \quad Filter = \begin{bmatrix} 1 & 2 & 3 \\ -4 & 7 & 4 \\ 2 & -5 & 1 \end{bmatrix} \quad Feature = \begin{bmatrix} 51 & 66 & 20 \\ 31 & 49 & 101 \\ 15 & 53 & -2 \end{bmatrix}$$

Geschrieben von Philipp Altnickel

Die Filter dienen dabei dazu, bestimmte Formen (Patterns) in der Eingangsmatrix zu erkennen. Im folgenden Beispiel soll eine horizontale Kante erkannt werden. Wird der Filter über Bereiche gelegt, die besonders gut auf diesen passen, wird sich ein hoher Wert in der Ausgangsmatrix, auch Featuremap genannt ergeben. Je schlechter die Übereinstimmung ist, desto geringer fällt dieser, auch Aktivierung genannte, Wert aus.

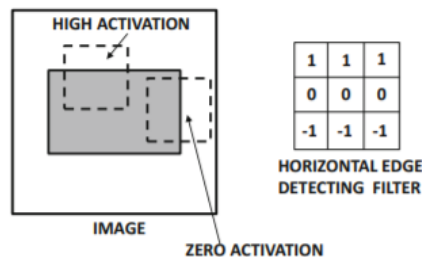


Abbildung 5: Funktionsweise eines Filters [Agg18]

Durch die Aneinanderreihung mehrerer Convolutional Layers, können auch Formen innerhalb der bisher gefundenen Formen erkannt werden. Das bedeutet, zunächst werden beispielsweise nur einfache Kanten (horizontal, vertikal, diagonal, ...) erkannt. Folgende Layer erkennen dann immer komplexere, daraus zusammengesetzte Formen.

Pooling Layer Beim Pooling wird ein Rechteck über die Eingangs-Featuremap gelegt. Alle Pixel, die sich in diesem Rechteck befinden werden zu einem einzigen Pixel in der Ausgabematrix zusammengefasst. Die Seitenlängen des Rechtecks können variiert werden, um zu bestimmen, in welchem Umkreis zusammengefasst werden soll. In der Regel sollten die Seitenlängen gleich sein, damit ein Quadrat entsteht und alle Richtungen gleichermaßen zusammengefasst werden. Um z.B. Schatten oder Rauschen zu minimieren, kann es sinnvoll sein, diesen Bereich zu vergrößern. Des Weiteren kann die Schrittweite (Stride) variiert werden. Je weiter das Rechteck bei jedem Schritt zur Seite bzw. nach unten weitergeschoben wird, desto kleiner wird die Ausgabematrix. Die Featuremap wird dann stärker komprimiert. Zum Pooling wird in der Regel eines der beiden Verfahren Max-Pooling oder Average-Pooling verwendet. Bei ersterem

wird der höchste Wert im jeweiligen Rechteck übernommen, beim zweiten wird der Durchschnitt all dieser Werte gebildet und übernommen. [Agg18]

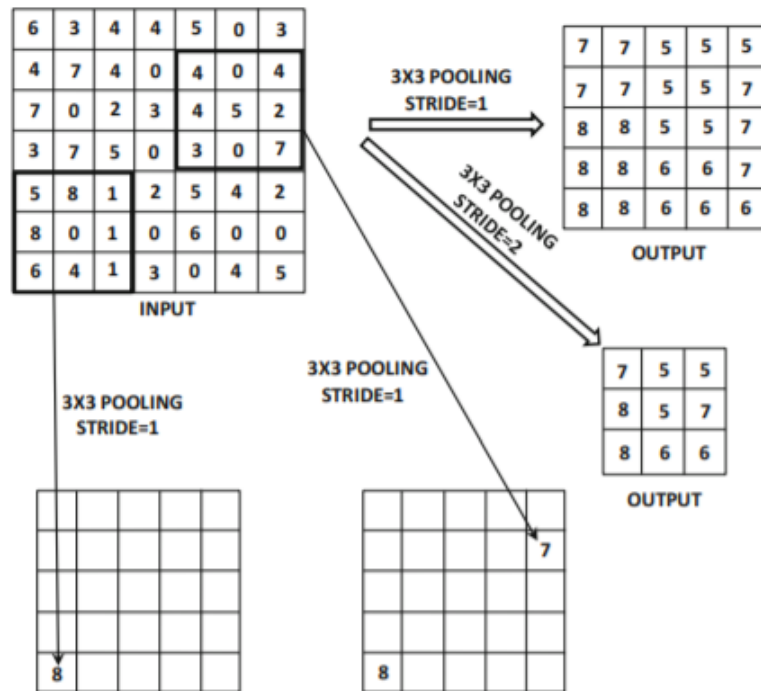


Abbildung 6: Beispiel für das Max-Pooling einer Featuremap [Agg18]

Fully Connected Layer In diesem Layer wird die schlussendliche Klassifizierung der Ergebnisse vorgenommen. Ziel war es von Anfang an, bestimmte Objekte in der Eingabematrix zu erkennen. Durch die vorherigen Ebenen wurden markante Muster heraus gefiltert und Ebene für Ebene zu komplexeren Formen zusammengefügt. Nach dem letzten Layer sind also bestimmte große Features, wie z.B. Räder, Lichter, etc. erkannt worden, wenn man sich ein Szenario zur Erkennung von Fahrzeugen vorstellt. Um daraus schlussendlich ein Auto, ein Fahrrad o.ä. zu erkennen, werden diese Features zu einer eindimensionalen Menge ausgerollt und. Jedes Feature ist dabei ein Eingangsneuron, jedes zu erkennende Objekt ein Ausgangsneuron. Dazwischen befindet sich eine klassische Neuronale Netzstruktur, bei der es Hidden-Layers, also nach außen nicht sichtbare Ebenen gibt. Alle Neuronen sind mit allen anderen der vorherigen und nachstehenden Ebenen verbunden. Auch das Training erfolgt hier auf die klassische Weise. [Agg18]

Geschrieben von Lukas Schimanski

In den folgenden Unterkapiteln werden unterschiedliche CNN Modelle vorgestellt, die einen Überblick über den Stand der Technik, deren Anwendungsfälle sowie Vor- und Nachteile geben sollen. Um diese Modelle miteinander vergleichbar zu machen, werden verschiedene Metriken und Datensätze verwendet, die hier zunächst definiert werden sollen.

2.3.1 Intersection over Union

Geschrieben von Lukas Schimanski

Um die Leistung bzw. Genauigkeit eines neuronalen Netzes zur Objekterkennung und Objektlokalisierung zu quantifizieren wird als Metrik die Intersection over Union (IoU) verwendet. Der IoU Wert beschreibt während des Trainings die Überlagerung der vorhergesagten Bounding Box mit der in den Trainingsdaten hinterlegten.

Sie ist definiert als $IoU = \frac{|A \cap B|}{|A \cup B|} = \frac{|I|}{|U|}$ [Rez+19]

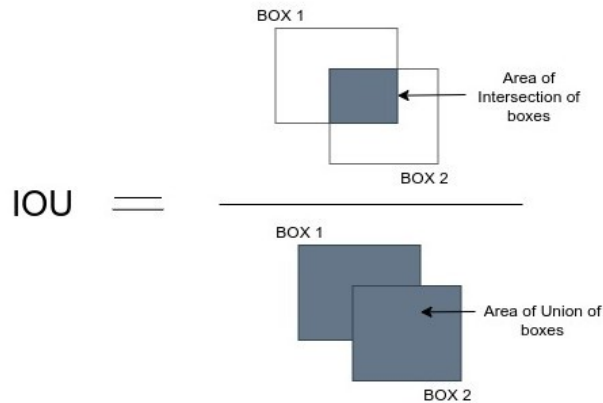


Abbildung 7: Darstellung der Berechnung der IoU [Sub]

Die IoU kann somit zwischen 0 und 1 liegen, wobei ein größerer Wert eine höhere Übereinstimmung zwischen der in den Trainings/Testdaten hinterlegten Bounding Box und der vom Netz vorhergesagten Bounding Box beschreiben.

2.3.2 Bounding Box

Geschrieben von Lukas Schimanski

Eine Bounding Box wird in der Objekterkennung zur Lokalisierung und Klassifizierung von Objekten in Bildern verwendet. Diese Boxen, wie sie bspw. in 14 zu sehen sind, werden um einzelne Objekte innerhalb eines Bildes gelegt, um damit ein neuronales Netz für die Objekterkennung zu trainieren. [AH20] Mit Hilfe der IoU kann somit die Genauigkeit des Netzes für die Lokalisierung bestimmt werden.

2.3.3 Average Precision

Geschrieben von Lukas Schimanski

Um die Average Precision (AP) zu definieren, werden vorerst zwei weitere Metriken benötigt. Zum einen die Sensitivität (engl. recall) und die Genauigkeit (engl. precision). Die Sensitivität gibt hierbei an, wieviele tatsächlich positiver Fälle als solche klassifiziert werden und ist definiert als $\frac{TP}{TP+FN}$. Die Genauigkeit hingegen gibt an, wieviele der als positiv klassifizierten Fälle tatsächlich positiv sind und ist definiert als $\frac{TP}{TP+FP}$. Hier bei steht TP für True Positive, FP für False Positive und FN für False Negative. Recall und Precision lassen sich in Abhängigkeit voneinander in einer Precision-Recall-Kurve festhalten.

Die Average Precision ist der Durchschnittswert der Precision p in Abhängigkeit des Recall r .

$$AvgP = \int_0^1 p(r) dr [Zhu04]$$

2.3.4 Mean Average Precision

Geschrieben von Lukas Schimanski

Die Mean Average Precision (mAP) ist bei der Objekterkennung der Mittelwert aller Average Precision Werte der einzelnen Klassen und ist somit definiert als:

$$MAP = \frac{\sum_{q=1}^Q AvgP(q)}{Q}$$

mit Q gleich der Anzahl an Klassen.[Zhu04]

2.3.5 Inferenz

Geschrieben von Lukas Schimanski

Als Inferenz wird die "Schlussfolgerung" eines neuronalen Netzes mit einer bestehenden Wissensbasis (engl. Knowledge base) bezeichnet. Im Gegensatz zum Training, wo die Knowledge base, im Falle eines neuronalen Netzes die Gewichte zwischen den Neuronen, durch Backpropagation angepasst wird, wird bei der Inferenz lediglich eine Ausgabe auf gegebene Eingangsdaten erzeugt.

2.3.6 Bewährte CNN-Modelle zum Klassifizieren von Bildern

Geschrieben von Tuncer Catalkaya

Im Verlaufe der Jahre haben sich Forscher damit beschäftigt CNN-Modelle, für die Klassifikation von Bildern, zu entwickeln. Hierbei haben sich einige Modelle durchgesetzt, diese werden als sogenannte „bewährte CNN-Modelle“ bezeichnet. Im Verlaufe dieses Kapitels wird auf **LeNet** und **ResNet** detailliert eingegangen. Es gibt noch viele weitere Modelle, allerdings sind die beiden aufgeführten Modelle große Meilensteine für die Welt der künstlichen Intelligenz. Im Anschluss werden einige bewährte CNN-Modelle, für die Klassifikation von Bildern, aufgelistet, aber nicht in diesem Bericht weiter erläutert. Die wichtigste Erkenntnis, die mitgenommen werden sollte ist, dass sich bereits Forscher mit der Anzahl/Typen von Neuronen und Schichten etc. beschäftigt haben. Somit muss das Rad nicht dauernd neu erfunden werden, stattdessen sollte man Nutzen von diesen bewährten CNN-Modelle beziehen. Zumal man mit sehr hoher Wahrscheinlichkeit selber kein besseres initiales Modell entwickeln würde.

LeNet LeNet gehört zu den bewährten CNN-Modellen zum Klassifizieren von Bildern und wurde im Jahr 1998 von Yann Lecun für handgeschriebene Ziffernerkennung entwickelt. [LeC+98]

Die handgeschriebene Ziffernerkennung ist häufig das einleitende Beispiel für Objekterkennung mit neuronalen Netzen. Üblicherweise wird zum Training der Daten der MNIST Datensatz herangezogen, siehe Abbildung 8. Die Bilder in dem Datensatz sind 28x28 große Graustufenbilder. [LeC+98]



Abbildung 8: Beispiele aus dem MNIST Datensatz [LeC+98]

Zusätzlich zu dem MNIST-Datensatz wird in Kombination, zu dem einleitenden Beispiel, die LeNet-5 Architektur herangezogen, da diese Architektur optimiert wurde für die handgeschriebene Ziffernerkennung. Die Architektur von LeNet-5 wird in Abbildung 9 dargestellt. LeNet-5 beschränkt sich auf maximal 32x32 große Graustufenbilder (ganz links im Bild zu sehen). Grundsätzlich besteht LeNet aus sieben Layern (ohne Input zu zählen): 3x Convolutional Layer, 2x Average Pooling Layer und 2x Fully Connected Layer. In Abbildung 9 wurden die Bezeichnungen für die Layer abgekürzt. Diese Kürzel bestehen aus einem Buchstaben (C = Convolutional Layer, S = Average Pooling Layer, F = Fully Connected Layer) gefolgt von einer Zahl (1 = erster Layer, 2 = zweiter Layer, ...). [LeC+98]

Es folgt nun eine genauere Beschreibung der Layer von LeNet-5 (vgl. Abb. 9):

- **INPUT:** Die Eingabe des neuronalen Netzes beschränkt sich auf maximal 32x32 große Graustufenbilder. [LeC+98]
- **C1:** Ein Convolutional Layer mit 6 feature maps, die jeweils eine Größe von 28x28 aufweisen. Übersetzt bedeutet dies, dass die Größe des Bilds verringert wurde (von 32x32 auf 28x28) während die Tiefe sich erhöht hat (von 1 auf 6). [LeC+98]
- **S2:** Ein Average Pooling Layer mit 6 feature maps, die jeweils eine Größe von 14x14 aufweisen. Es wurde also die Größe um die Hälfte verringert (von 28x28 zu 14x14) und die Tiefe beibehalten. [LeC+98]
- **C3:** Ein Convolutional Layer mit 16 feature maps, die jeweils eine Größe von 10x10 aufweisen. Erneut wurde die Größe des Bilds verringert, aber dafür die Tiefe erhöht. [LeC+98]
- **S4:** Ein Average Pooling Layer mit 16 feature maps, die jeweils eine Größe von 5x5 aufweisen. Erneut wurde die Größe halbiert während die Tiefe beibehalten wurde. [LeC+98]
- **C5:** Ein Convolutional Layer mit 120 feature maps, die jeweils eine Größe von 1x1 aufweisen. Erneut wurde die Größe des Bilds verringert während die Tiefe erhöht wurde. [LeC+98]
- **F6:** Ein Fully Connected Layer mit 84 Neuronen. Die Anzahl von 84 Neuronen hat einen Grund, welcher aber hier nicht genauer beschrieben wird (siehe [LeC+98] (Kapitel II Abschnitt B) falls Interesse für den Grund besteht). [LeC+98]
- **OUTPUT:** Ein Fully Connected Layer mit 10 Neuronen. Die Anzahl 10 kommt durch die verschiedenen möglichen Ausgaben bei einer Ziffernerkennung. Bei einer Ziffernerkennung sind die möglichen Ausgaben: 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9. [LeC+98]

Zusammengefasst ist der Ablauf von LeNet-5, dass das Originalbild (32x32) immer kleiner wird, während die Tiefe sich immer weiter erhöht. Dies wird solange durchgeführt bis das Bild, mit einer gewissen Tiefe, 1x1 groß ist. Daraus werden dann mithilfe von Fully Connected Layern die Neuronen immer weniger bis die gewünschte Anzahl an Neuronen für die Ausgabe erreicht wurde. [LeC+98]

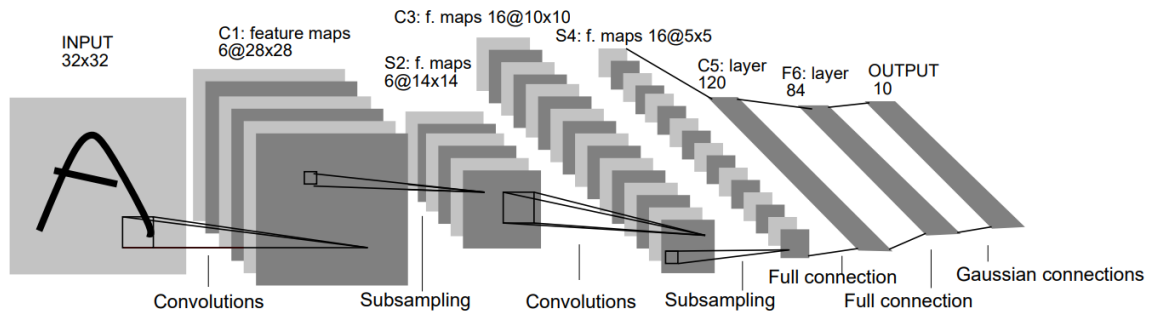


Abbildung 9: LeNet-5 Architektur [LeC+98]

ResNet Residual Neural Network (ResNet) gehört zu den bewährten CNN-Modellen zum Klassifizieren von Bildern und wurde im Jahr 2016 von Microsoft Research entwickelt.

ResNet ermöglicht es tiefere Netzwerke zu entwickeln und hat damit ein großes Phänomen beseitigt, welches in den CNN Netzwerken vor ResNet beobachtet werden konnte. Dieses Phänomen bewirkte, dass ein Netzwerk mit mehr Schichten eine höhere Trainings-Fehlerrate und damit auch eine höhere Test-Fehlerrate aufweist als ein Netzwerk mit weniger Schichten. Es war zudem bereits bekannt, dass dies nicht an Overfitting (Überanpassung, das Modell kennt die Trainingsdaten zu gut) liegt. Dieses Phänomen wird in Abbildung 10 dargestellt, wobei sich dies auf den CIFAR-10 Datensatz bezieht (siehe [KH+09] für genauere Informationen zum Datensatz). [He+16]

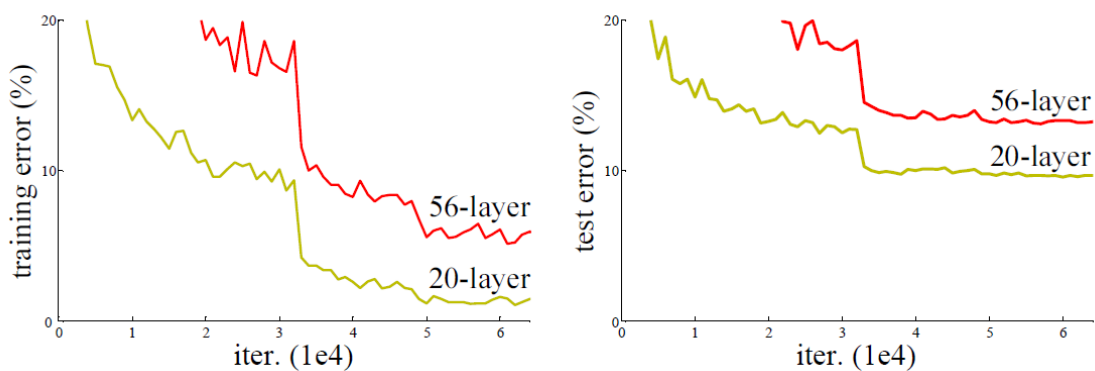


Abbildung 10: Training und Test Fehlerrate bei CNN Netzwerken vor ResNet (CIFAR-10) [He+16]

ResNet beseitigt dieses Phänomen indem es mit sogenannten Sprüngen arbeitet. Diese Sprünge gab es in den CNN Netzwerken vor ResNet nicht, wodurch bei einem tieferen Netzwerk die tieferen Schichten irgendwann nur noch die identity Funktion (Ausgang entspricht dem Eingang, $f(x) = x$) gelernt haben. Bei ResNet wird dies genau umgedreht, es wird standardmäßig die identity Funktion angenommen und um das verändert was gelernt werden soll, dieses Prinzip wird in Abbildung 11 dargestellt und wird „Residual Layer“ genannt.

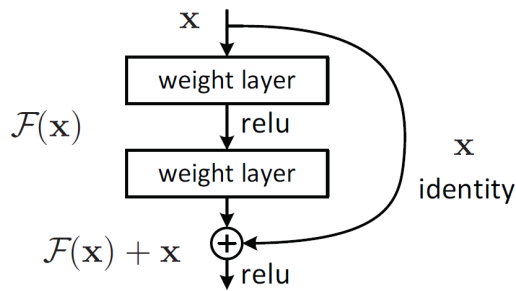


Abbildung 11: Prinzip von ResNet (Residual Layer) [He+16]

In Abbildung 12 wird die Auswirkung von den Sprüngen, die in ResNet verwendet werden, verdeutlicht, wobei dies sich auf den ImageNet Datensatz bezieht (siehe [KSH12] (Kapitel 2) für genauere Informationen zum Datensatz). Links in der Abbildung, bei den CNN Netzwerken vor ResNet, kann das übliche Phänomen beobachtet werden. Das bedeutet also, dass bei der Verwendung von mehr Schichten die Fehlerrate höher ist als bei einem Netzwerk mit weniger Schichten. Rechts in der Abbildung, bei ResNet, ist dies genau umgekehrt. Die Fehlerrate ist also geringer als bei einem Netzwerk mit weniger Schichten. [He+16]

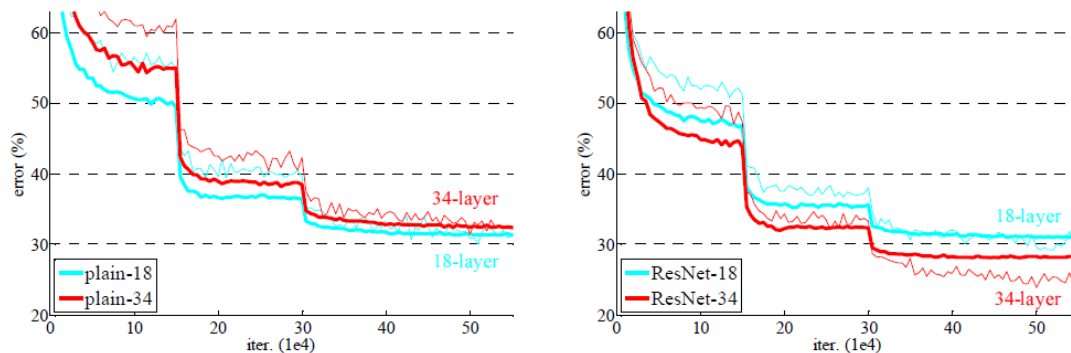


Abbildung 12: Fehlerrate bei CNN Netzwerken vor ResNet und ResNet (ImageNet) [He+16]

Weitere bewährte CNN-Modelle zum Klassifizieren von Bildern Es folgt eine Auflistung mit dem Namen und einem Verweis zur Quelle von weiteren bewährten CNN-Modellen zum Klassifizieren von Bildern. Der Verweis zur Quelle enthält die genauen Informationen zu diesen CNN-Modellen und kann, falls Interesse besteht, dort genauer nachgelesen werden:

- **AlexNet:** Siehe [KSH12] für genauere Informationen.
- **VGGNet:** Siehe [SZ14] für genauere Informationen.
- **GoogLeNet:** Siehe [Sze+15] für genauere Informationen.

Die obige aufgeführte Liste ist keine vollständige Liste. Es gibt noch viele weitere Modelle!

2.3.7 Region-based Convolutional Neural Network

Geschrieben von Tuncer Catalkaya

Region-based Convolutional Neural Network (R-CNN) ist eine Methode zur Objekterkennung und gehört zu den zweistufigen Detektoren. Die zwei Stufen sind folgende:

1. Es wird zuerst das Bild nach sogenannten **Regions of Interest (RoI)** untersucht, das sind die Regionen die ein Objekt enthalten könnten
2. danach werden die **RoI klassifiziert**

Der Fokus von R-CNN liegt auf Präzision und ist damit trotz all den Geschwindigkeitsoptimierungen (wird in den nachfolgenden Kapiteln genauer drauf eingegangen) ungeeignet für eine Echtzeiterkennung. Es wurden über der Zeit verschiedene Lösungsansätze und Verbesserungen entwickelt, diese werden in diesem Kapitel näher erläutert. [Gir+14]

R-CNN

R-CNN nimmt als Eingang ein Bild beliebiger Größe und extrahiert mittels Selective Search (mit Greedy Algorithm) die interessanten Regionen, auf Selective Search (mit Greedy Algorithm) wird in dem nachfolgenden Paragraphen genauer eingegangen. Die interessanten Regionen sind ungefähr auf maximal 2000 Regionsvorschläge begrenzt. Diese Regionen werden jeweils in eine feste Größe transformiert und mit einem CNN analysiert. Die CNN extrahierten Features werden zum Klassifizieren in eine Support Vector Machine (SVM, siehe [Gun+98] (Kapitel 2) für genauere Informationen) eingespeist. Zusätzlich zu der Klassifizierung wird mit einem BoundingBox-Regressor (siehe [Gir+14] für genauere Informationen) Position und Größe des Objekts im Bild bestimmt. Dieses Vorgehen wird anhand der Architektur von R-CNN in Abbildung 13 dargestellt. [Gir+14]

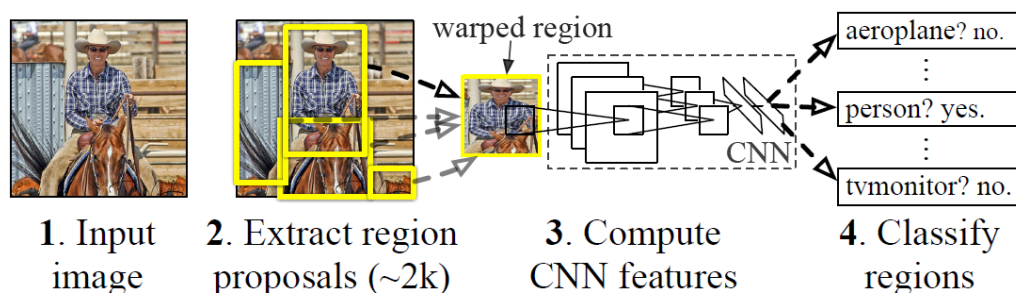


Abbildung 13: R-CNN Architektur [Gir+14]

Selective Search mit Greedy Algorithm Selective Search ist ein fixer Algorithmus um interessante Regionen in einem Bild zu bestimmen. Der Algorithmus besteht aus drei festen Schritten, diese sind:

1. Generiere initiale Segmentation, also „Unterteilung“ des Bilds
2. Kombiniere rekursiv kleine ähnliche Regionen mit Greedy Algorithm, der Greedy Algorithm besteht auch aus drei festen Schritten und ist ebenso ein fixer Algorithmus:
 - (a) Wähle von den Regionen zwei die am meisten Ähneln
 - (b) Kombiniere diese Beiden zu einer größeren Region
 - (c) Wiederhole die oben genannten Schritte für mehrere Iterationen
3. Benutze die vorgeschlagenen segmentierten Regionen um Position des Kandidatenobjekts zu generieren

In Abbildung 14 werden die oben definierten Schritte visuell dargestellt (von links nach rechts). Ganz links ist die initiale Segmentation (Schritt 1), diese ist laut Definition auf ≈ 2000 begrenzt. In der Mitte wird dargestellt wie es nach einigen Iterationen aussehen könnte (wiederholend Schritt 2 und 3), es wird als eine interessante Region (könnte ein Objekt enthalten) der Fernseher vorgeschlagen. Ganz Rechts wird dargestellt wie es nach vielen Iterationen aussehen könnte (wiederholend Schritt 2 und 3), jetzt wird zusätzlich eine Frau auf der rechten Seite als interessante Region angesehen, weil die Regionen durch Schritt 2 groß genug sind. [Uij+13]



Abbildung 14: Selective Search mit Greedy Algorithm [Uij+13]

R-CNN weist jedoch einige Probleme auf: Ein Problem davon ist die sehr lange Trainingszeit durch die große Anzahl der Regionen die klassifiziert werden müssen. Ein weiteres Problem ist die Auswahl von Selective Search als Region Proposal Algorithmus (Region Proposal = Regionsvorschläge), denn dieser ist fest definiert und kann sich aufgrund dessen über die Zeit nicht verbessern. [Gir15]

Fast R-CNN

Die Problematik der sehr langen Trainingszeit wird in Fast R-CNN aufgegriffen. Hierzu wird das ganze Bild mittels einem CNN analysiert, statt wie zuvor die einzelnen Regionen, was einen enormen Zeitvorteil bringt. Die interessanten Regionen werden mit Selective Search aus den CNN extrahierten Features ermittelt und in eine feste Größe transformiert. Das transformierte Bild wird wiederum in ein Fully Connected Layer eingespeist. Zur Klassifizierung wird ein Softmax-Layer statt einer Support Vector Machine verwendet (siehe [Gir15] (Do SVMs outperform softmax?) für genauere Informationen). Um Position und Größe des Objekts im Bild zu bestimmen wird nach wie vor ein BoundingBox-Regressor eingesetzt. Die Architektur von Fast R-CNN wird in Abbildung 15 dargestellt. [Gir15]

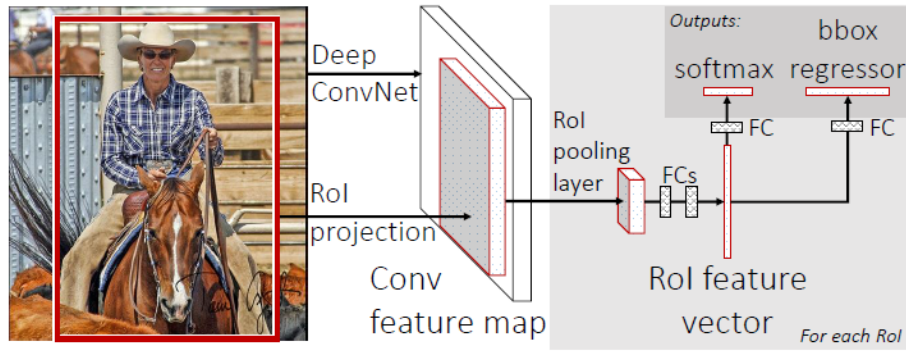


Abbildung 15: Fast R-CNN Architektur [Gir15]

Ein Zeitvergleich zwischen R-CNN und Fast R-CNN wird in Abbildung 16 dargestellt. Fast R-CNN ist mit seinen Verbesserungen deutlich schneller als R-CNN.

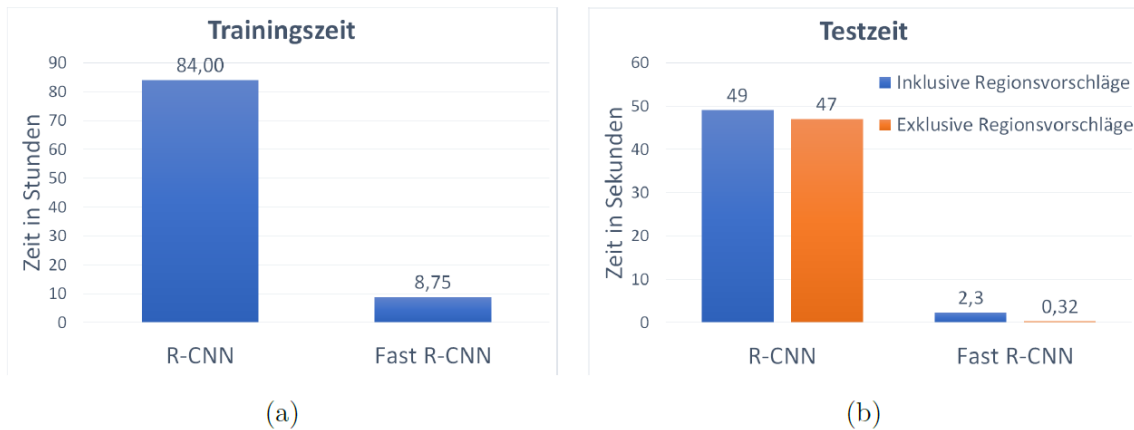


Abbildung 16: Zeitvergleich zwischen R-CNN und Fast R-CNN [Voi19]

Faster R-CNN

Die Problematik der Auswahl von Selective Search als Region Proposal Algorithmus wird in Faster R-CNN aufgegriffen. Es wird nach wie vor das gesamte Bild mittels einem CNN analysiert. Statt mit Selective Search die RoI aus der feature map zu ermitteln wird ein Mini-Netzwerk, bestehend aus Convolutional Layern, eingesetzt. Dieses Mini-Netzwerk wird als Region Proposal Network (RPN) bezeichnet. Es wird nach wie vor zur Klassifizierung ein Softmax-Layer verwendet, und um Position und Größe des Objekts im Bild zu bestimmen ein BoundingBox-Regressor eingesetzt. Die Architektur von Faster R-CNN wird in Abbildung 17 dargestellt. [Ren+16]

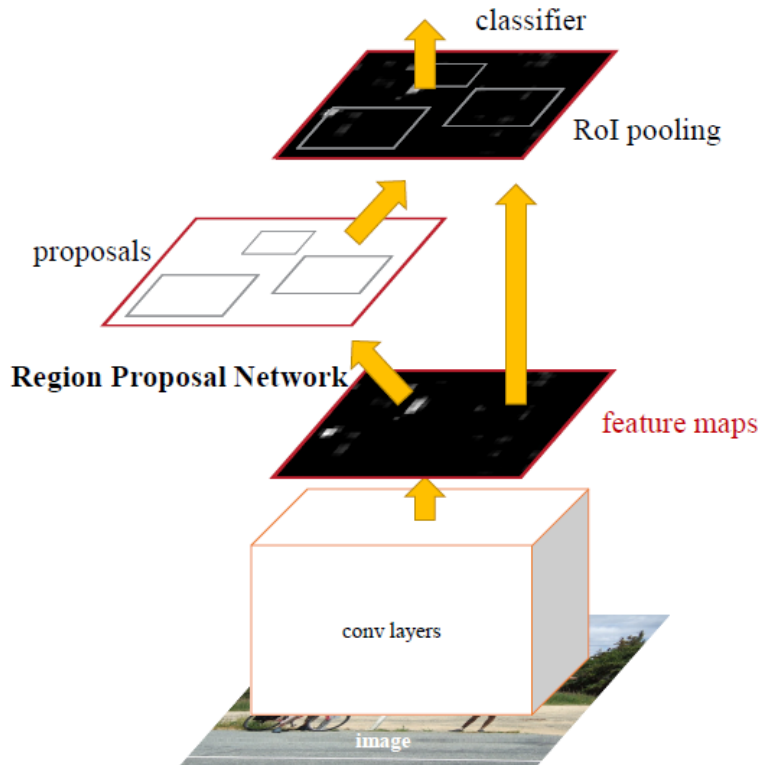


Abbildung 17: Faster R-CNN Architektur [Ren+16]

Region Proposal Network Das RPN nimmt als Eingabe ein Bild beliebiger Größe und gibt eine Menge von rechteckigen Objektvorschlägen aus, sogenannte Region Proposals. Zur Eingabe wird ein Sliding-Window verwendet. Ein Sliding-Window ist ein kleines quadratisches Fenster (z.B. 3x3 groß) und geht sukzessiv durch das Bild durch, diese Eingabe des Sliding-Windows wird jeweils zu einem lower-dimensional feature layer gemappt. Diese Features gehen wiederum in ein box-classification layer (cls) und box-regression layer (reg). [Ren+16]

Innerhalb des Sliding-Windows wird mit sogenannten Anchor gearbeitet. Anchor sind Kandidaten zur Klassifizierung um Regionsvorschläge zu bestimmen, und werden innerhalb des Sliding-Windows zentriert angelegt. Es werden somit mehrere Regionsvorschläge innerhalb eines Sliding-Windows getätigt [Wan+19]. Die maximal möglichen Regionsvorschläge (Anchor) innerhalb eines Sliding-Windows wird mit k notiert und ergibt sich aus der Anzahl der Skalierungen und Seitenverhältnissen, siehe Formel 3. Die Gesamtanzahl der Anchor innerhalb eines Bildes setzt sich aus der Größe des Bildes und der maximal möglichen Regionsvorschläge innerhalb eines Sliding-Windows zusammen, siehe Formel 4. [Ren+16]

$$k = \text{number_of_scale} \cdot \text{number_of_aspectRatio} \quad (3)$$

$$\text{max_anchor_in_picture} = \text{width} \cdot \text{height} \cdot k \quad (4)$$

Es folgt nun eine Beispielrechnung für die Bestimmung der Gesamtanzahl der Anchor. Dazu wird angenommen, dass das zu analysierende Bild (Eingabe des CNN) eine Größe von 1000 x 600 aufweist. Die berechnete feature map aus der CNN wäre damit 40 x 60 groß. Die feature map ist die Eingabe des RPN. Für die Anchor (k) werden 3 Skalierungen (z.B.: 128, 256, 512 pixel) und 3 Seitenverhältnisse (z.B.: 1:1, 1:2, 2:1) gewählt, dies entspricht laut Formel 3 Folgendem: $k = 3 \cdot 3 = 9$. Die Breite (width) und Höhe

(height) kann von der Größe der feature map entnommen werden. Zusammen mit dem zuvor berechneten k kann die Gesamtanzahl der Anchor berechnet werden, dies entspricht laut Formel 4 Folgendem: $max_anchor_in_picture = 40 \cdot 60 \cdot 9 = 21600$. Die Gesamtanzahl der Anchor beträgt für dieses Beispiel also 21600, diese Anzahl wäre zu hoch bzw. wäre dadurch die Trainingszeit erheblich länger. [Ren+16]

Um die Gesamtanzahl der Anchor zu reduzieren werden cross-boundary Anchor ignoriert, das sind Anchor, die über dem Bildrand hinausragen. Für das oben genannte Beispiel würde dies die Anzahl der Anchor von 21600 auf ≈ 6000 reduzieren. Zusätzlich wird Non-Maximum Suppression angewendet (NMS). NMS ist ein wichtiger Schritt in der Computer Vision Nachbearbeitung und löst die Problematik, dass mehrere Regionskandidaten versuchen dasselbe Objekt zu erfassen [RGV14]. Dazu wird von den Regionskandidaten, die versuchen dasselbe Objekt zu erfassen, der Kandidat gewählt, der den höchsten IoU Wert aufweist. Dies hat keinen großen Einfluss auf die Genauigkeit, verringert jedoch die Anchor Anzahl erheblich. Für das oben genannte Beispiel würde dies die Anzahl der Anchor von ≈ 6000 auf ≈ 2000 reduzieren. Diese Anzahl ist gering genug und würde damit die Trainingszeit erheblich reduzieren ohne eine große Auswirkung auf die Genauigkeit zu haben. [Ren+16]

Die Klassifizierung (cls, box-classification layer) beschränkt sich lediglich auf „ist ein Objekt oder ist kein Objekt“ (0-1), dazu wird mittels dem IoU Wert eine Aussage getätigt, siehe Formel 5. Sollte es keinen IoU Wert größer als 0.7 geben, dann wird der größte IoU Wert als „ist ein Objekt“ angenommen. [Ren+16]

$$\begin{aligned} \text{Objekt} : IoU > 0.7 \\ \text{Kein Objekt} : IoU < 0.3 \end{aligned} \tag{5}$$

Die Lokalisierung (reg, box-regression layer) enthält im Wesentlichen die typischen Variablen um das Objekt im Bild eine Position (x-Koordinate, y-Koordinate) und Größe (Breite, Höhe) zu vergeben. [Ren+16]

Die Architektur des RPN wird in Abbildung 18 dargestellt. Es ist zu erkennen, dass das Sliding-Window zur Eingabe auf die feature map angewendet wird und innerhalb des Fensters Anchor, mit verschiedenen Skalierungen und Seitenverhältnissen, zentriert angelegt werden. Dies wird jeweils dann zu einem lower-dimensional feature layer gemappt und geht in den box-classification und box-regression layer. Der box-classification layer (cls) enthält $2 \cdot k$ scores, die Zahl 2 kommt durch die Entscheidung, ob es ein Objekt ist oder nicht und ist abhängig von k , also der Anzahl der Anchor, weil dies für jedes Anchor durchgeführt werden muss. Der box-regression layer (reg) enthält $4 \cdot k$ coordinates, die Zahl 4 kommt durch die Anzahl der Koordinaten für eine Lokalisierung (x-Koordinate, y-Koordinate, Breite, Höhe) und ist ebenso abhängig von k . [Ren+16]

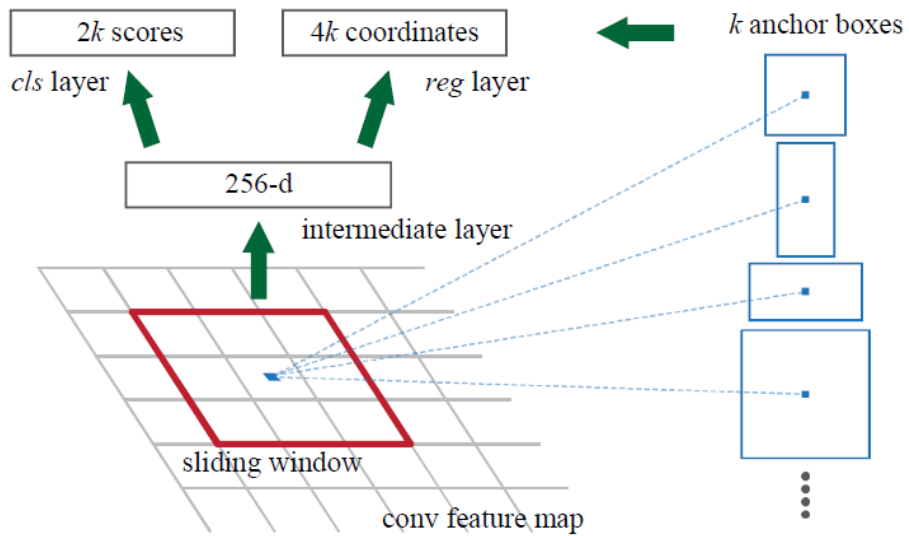


Abbildung 18: RPN Architektur [Ren+16]

Ein Zeitvergleich zwischen R-CNN, Fast R-CNN und Faster R-CNN wird in Abbildung 19 dargestellt. Faster R-CNN konnte die benötigte Zeit für Regionsvorschläge im Vergleich zu Fast R-CNN deutlich reduzieren. Wie schon vorher erwähnt reicht diese erhebliche Zeitreduzierung nicht aus für eine Echtzeit-erkennung bzw. sind die sogenannten einstufigen Detektoren, die noch im Verlaufe der Kapitel erklärt werden, schneller.

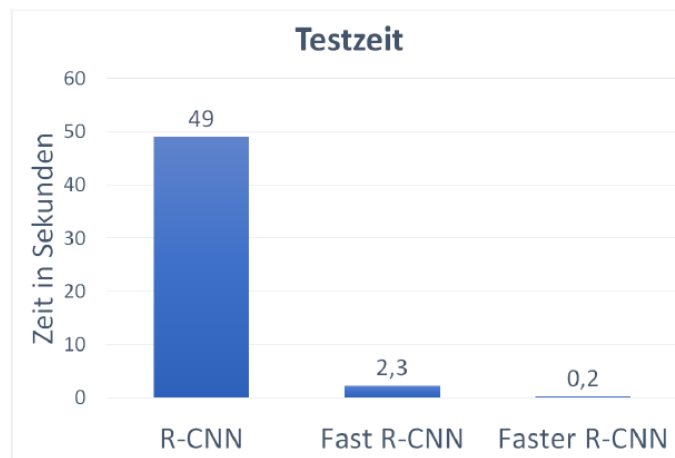


Abbildung 19: Zeitvergleich zwischen R-CNN, Fast R-CNN und Faster R-CNN [Voi19]

Mask R-CNN

Mask R-CNN erweitert Faster R-CNN um einen weiteren Zweig, welcher zuständig ist für das Vorhersagen von Segmentierungsmasken der einzelnen Interessenregionen. Dazu wird ein Fully Connected Netzwerk in Einsatz genommen, welches eine Segmentierungsmaske auf Pixelebene vorhersagen soll, dies führt nur zu einem minimal größeren Rechenaufwand. Außerdem wird ein quantisierungsfreier Layer eingeführt um die genauen Positionsdaten persistieren zu können. In Abbildung 20 wird die Architektur von Mask R-CNN dargestellt. [He+17]

Eine Maskierung ist für dieses Projekt zwar nicht relevant, kann aber in Zukunft nützlich sein, falls es einen Anwendungsfall gibt, bei dem ein genauere Umriss eines Objekts (auf Pixelebene) benötigt wird.

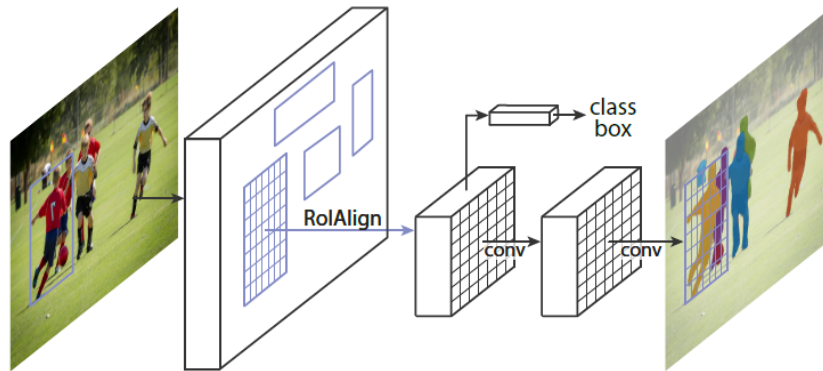


Abbildung 20: Mask R-CNN Architektur [He+17]

In Abbildung 21 werden einige Resultate von Mask R-CNN aus dem COCO Datensatz (siehe [Dat] für genauere Informationen zum Datensatz) gezeigt. Es kann gesehen werden, dass die Bounding-Boxen jetzt Masken sind, die auf Pixelebene voraussagen, wo sich die Objekte in dem Bild befinden.



Abbildung 21: Beispiele von Mask R-CNN Resultaten (COCO) [He+17]

2.3.8 YOLO

Geschrieben von Lukas Schimanski

You Only Look Once (YOLO) ist ein CNN Modell, welches auf die Echtzeiterkennung von Objekten auf Basis des PASCAL VOC 2012 [EGW] Datensatzes mit insgesamt 20 Klassen fokussiert ist. [Red16]

YOLO ist in seiner ersten Form (V1) im Jahr 2016 entwickelt worden und wurde seitdem von verschiedenen Entwicklern weiter optimiert.

YOLO V1 verarbeitet 224*224 große Eingangsbilder indem diese in ein gleichmäßiges Gitter aufgeteilt und in jedem Gitterfeld gleichzeitig Bounding Boxes und deren Klassenwahrscheinlichkeiten vorhergesagt werden. Siehe hierzu auch Abbildung 22.

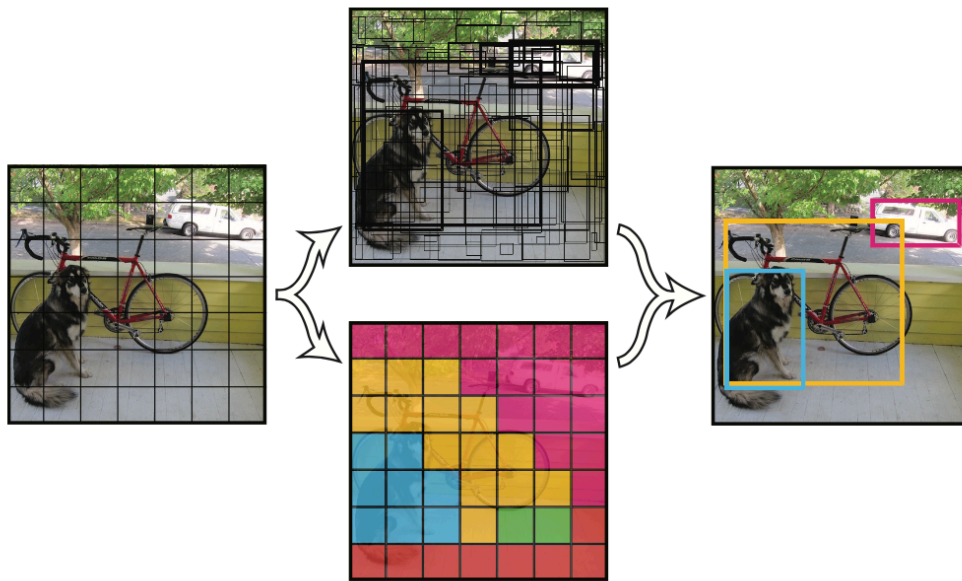


Abbildung 22: Bildverarbeitung in YOLO [Red+16]

Durch die Aufteilung in ein Gitter hat YOLO allerdings Schwierigkeiten kleine Objekte innerhalb eines Bildes zu erkennen, als auch Objekte zu generalisieren.[Red+16]

Dies passiert innerhalb eines CNN, welches, wie Abbildung 23 zeigt, aufgebaut ist.

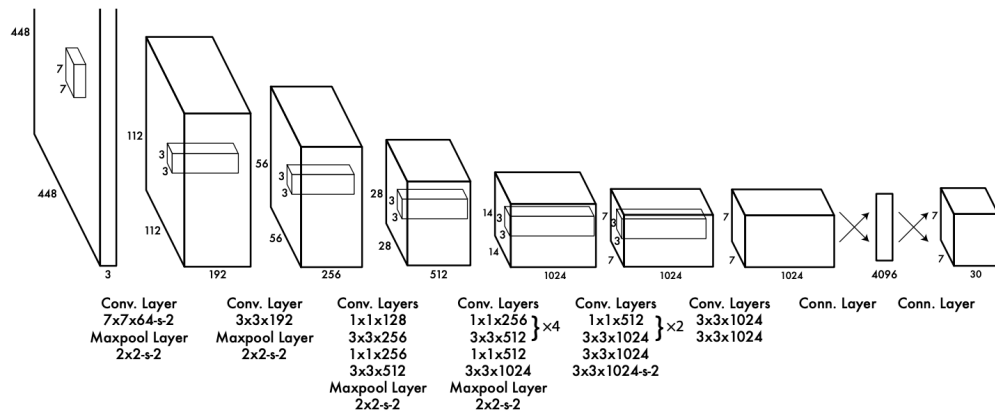


Abbildung 23: Aufbau YOLO V1 [Red+16]

Im Verlauf der Entwicklung wurde die CNN-Architektur zunächst auf Darknet-19, sowie Batch Normalization nach jedem Layer, und mit YOLO V3 dann auf Darknet-53 umgestellt, welches wiederum auf der ResNet Architektur basiert.[RF17; RF18]

Hierdurch konnte die Genauigkeit von YOLO kontinuierlich verbessert werden. Tabelle 1 zeigt einen Vergleich der Average Precision zwischen YOLO V2 und YOLO V3 mit dem COCO Dataset.

	backbone	AP	AP_{50}	AP_{75}
YOLOv2	Darknet-19	21.6	44.0	19.2
YOLOv3	Darknet-53	33.0	57.9	34.4

Tabelle 1: Vergleich YOLOv2 zu YOLOv3 (Daten basieren auf [RF18])

Es zeigt sich, dass die mittlere Genauigkeit von YOLO V2 zu YOLO V3, im Test mit dem COCO Dataset, um 50% gesteigert werden konnte.

2.3.9 Single Shot Detector

Geschrieben von Lukas Schimanski

Der Single Shot Multibox Detector (SSD) ist eine one-stage Objekterkennungsarchitektur. Sie zeigt in der Konzeption und Umsetzung parallelen zu YOLO auf, unterscheidet sich allerdings architektonisch in den abschließenden Layern des neuronalen Netzes dazu.[Liu+16]

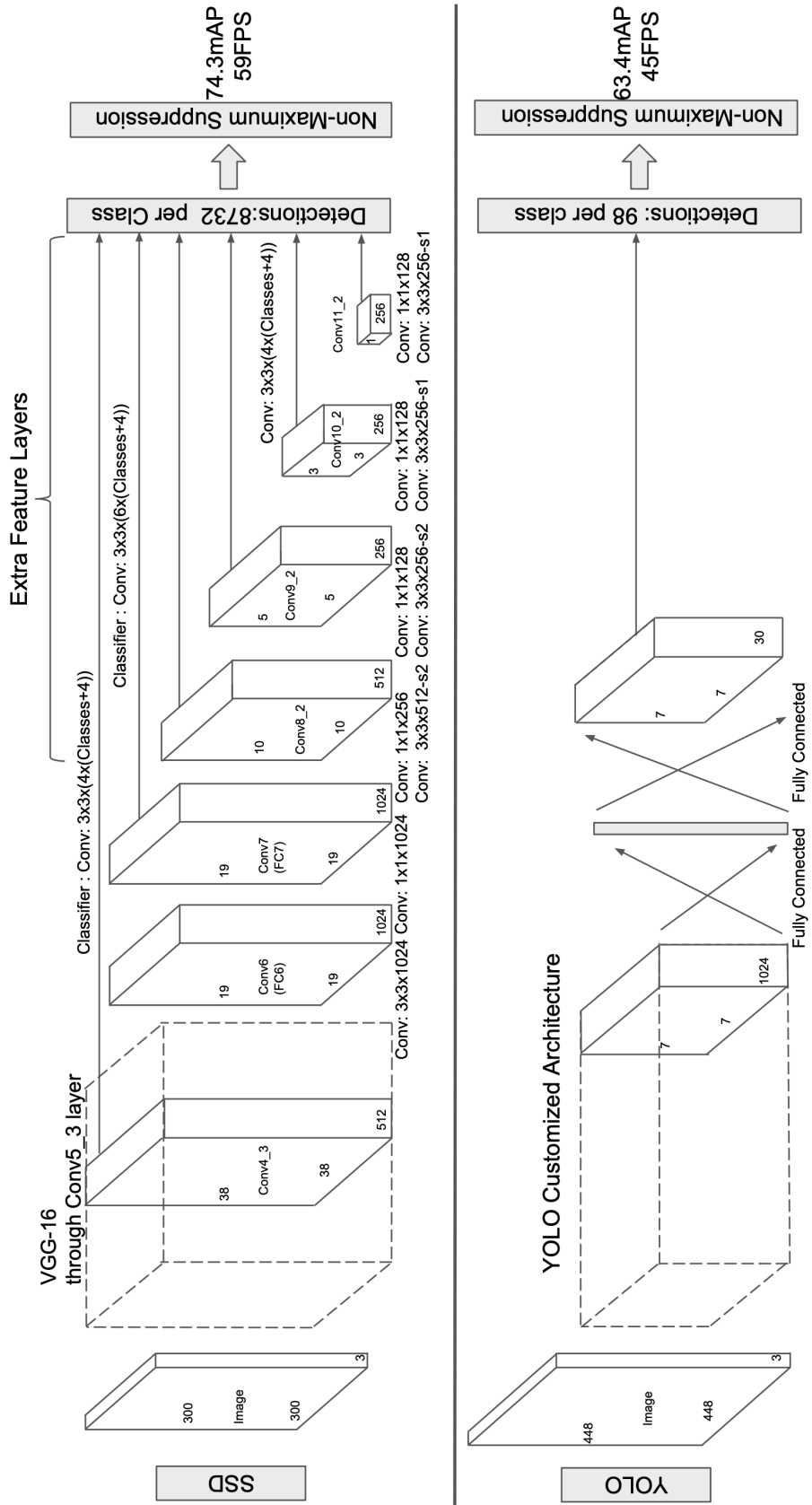


Abbildung 24: Vergleich des Aufbaus von SSD und YOLO V1 [Liu+16]

Wie aus Abbildung 23 und 24 hervorgeht, verwendet YOLO Fully Connected Layer um eine abschließende Klassifizierung des Bildes vorzunehmen.

SSD ersetzt diese Fully Connected Layer durch weitere Convolutional Layer, welche Offsets zu Standardboxen verschiedener Maßstäbe und Seitenverhältnisse und ihre zugehörigen Konfidenzen vorhersagen. [Liu+16]

Im Testvergleich mit dem VOC2007 Datensatz [EGW] zeigt SSD zum einen eine höhere Genauigkeit (Mean Average Percentage) als auch eine bessere Inferenzzeit, 17ms pro Bild im Vergleich zu YOLO's 22ms.

2.4 Binary Neural Networks (BNN)

Geschrieben von Jonas Philipp, Ismail Sastim, Toni Dragojevic, Viktor Chechulin

In diesem Kapitel werden die Grundlagen von BNN beschrieben. Im folgenden wurden die Paper (Wissenschaftliche Abhandlung) [Cou+16] und [SL19] verwendet, dabei eignet sich besonders [Cou+16] als Nachschlagewerk. Besonders wichtig für das Projekt sind die Unterkapitel „2.4.1 Forward Propagation“ und „2.4.2 Backpropagation in BNNs“.

Was sind BNN? Binary Neural Networks (BNN) sind eine spezielle Form von neuronalen Netzen (NN) und zählen zu den Quantisierten neuronalen Netzen. BNN wurden mit der Intention entwickelt Speicherverbrauch und Rechenaufwand von NN zu reduzieren. Dadurch wird der Einsatz von NN auf ressourcenschwachen Geräten ermöglicht, darunter zählt auch das FPGA Artix 7, benutzt in diesem Projekt. Darüber hinaus eignet sich der für das Projekt relevante Bereich Objekterkennung sehr gut für BNN, denn bei diesem Bereich verbrauchen NN mit reellen Gewichten und reellen Aktivierungen viel Speicherplatz. Das kommt daher das man für die Darstellung einer reellen Zahl, 32-Bit (Single Precision oder Einfache Genauigkeit), 64-Bit (Double Precision oder Doppelte Genauigkeit) oder 128-Bit (Quad Precision oder Vierfache Genauigkeit) benötigt. Zudem ist auch der Rechenaufwand, verursacht durch rechenaufwändige Multiplikationen reeller Zahlen sehr groß, denn eine Berechnung mit reellen Zahlen benötigt mehrere Speicherzyklen und nimmt zudem viel Hardware in Anspruch, sodass man dadurch nur wenige solcher Berechnungen parallel ausführen kann.

Die vorher erwähnten Probleme werden durch zwei Charakteristische Merkmale die BNN auszeichnen behoben. Das erste Merkmal ist, das in BNN, Gewichte und Aktivierungen auf zwei Werte (-1 oder 1) begrenzt werden und somit ein Bit ausreicht für die Darstellung, daher der Name „Binary“ und der Vorteil das BNN weniger (mindestens 32x) Speicher verbrauchen als z.B das im Kapitel 2.3 beschriebene CNN, welches reelle Gewichte und reelle Aktivierungen verwendet. Das zweite Charakteristische Merkmal ist, das durch binäre Gewichte und binäre Aktivierungen in BNN, rechenaufwändige Multiplikationen reeller Zahlen, welche z.B in CNN im Convolutional Layer für jedes Pixel durchgeführt werden, durch schnellere und weniger Hardware in Anspruch nehmende binäre Operationen (XNOR) ersetzt werden und somit eine erhebliche Performance Verbesserung erreicht wird (23x) [SL19].

Neben den Vorteilen gibt es auch Nachteile, denn in BNN leidet die Genauigkeit welche in Prozent angibt wie oft Bilder vom NN richtig erkannt wurden, erheblich durch die Quantisierung. Allerdings ist dieses Problem nicht unbekannt und die Forschung in diese Richtung ist schon sehr fortgeschritten, sodass BNN derzeit nahezu die Genauigkeit von reellen Modellen wie CNN erreichen.

2.4.1 Forward Propagation

Die Forward Propagation im Allgemeinen wird in einem Neuronalen Netz für die Inferenz - also für die Prognose der Eingaben - verwendet. Die Eingabe wird, wie in 2.3 aufgeführt, dabei von Layer zu Layer bis zur Ausgabe durchgereicht. Inferiert wird sowohl während des Trainings als auch im Produktiveinsatz.

Im Originalpaper von Binären Neuronalen Netzen wird der Algorithmus durch folgenden Pseudocode beschrieben [Cou+16]:

```
for k = 1 to L do
   $W_k^b \leftarrow \text{Binarize}(W_k)$ 
   $s_k \leftarrow a_{k-1}^b W_k^b$ 
   $a_k \leftarrow \text{BatchNorm}(s_k, \theta_k)$ 
  if k < L then
     $a_k^b \leftarrow \text{Binarize}(a_k)$ 
  end if
end for
```

Listing 1: Pseudocode der Forward Propagation

Wobei x^b eine binäre Matrix symbolisiert und L für die Anzahl der Gesamtlayers steht.

Der Algorithmus würde also alle Layer durchlaufen...

In jedem Layer die Gewichtungsmatrizen binarisieren...

Mit den binarisierten Matrizen des vorherigen Layers multiplizieren...

Das Ergebnis wird batchnormalisiert...

Falls es sich um das letzte Layer handelt, wird die batchnormalisierte Matrix nicht mehr binarisiert.

Quantisierung Von Interesse ist insbesondere die *Binarize()* Funktion. Es handelt sich hierbei um eine Quantisierungsfunktion, welchen die Binären Neuronale Netze ihren entscheidenden Speicher- und Performancevorteil zu verdanken haben. Es werden im Originalpaper allerdings zwei Funktionen vorgestellt, wovon jedoch nur einer hauptsächlich genutzt wurde. [Cou+16]

Deterministische Quantisierung Die deterministische Quantisierung ist die bevorzugte Variante, da sie sich auf Hardware hervorragend umsetzen lässt. Sie ist folgendermaßen definiert:

$$x^b = \text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases} \quad (6)$$

Graphisch ist dies in Abbildung 25 zu betrachten.

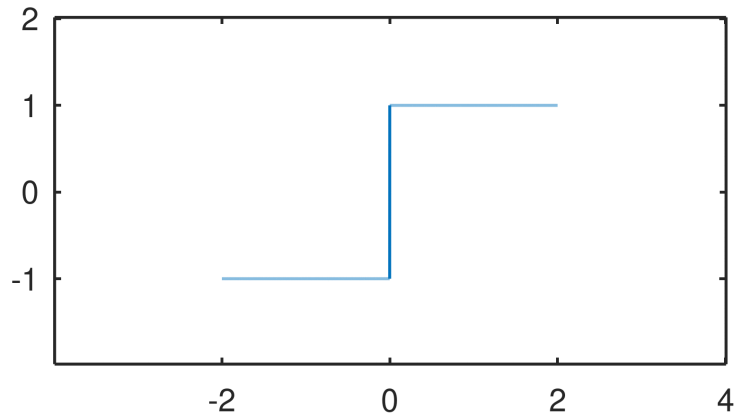


Abbildung 25: Signum Funktionsverlauf

Es wird deutlich, dass die $\text{sign}(x)$ Funktion nur einen Wertebereich von $\{-1, +1\}$ besitzt. Dies bedeutet zwar Informationsverlust und zudem lässt sich die Funktion nicht differenzieren, jedoch lässt sich an der Stelle bereits sagen, dass im Vergleich zu 32-Bit Fließkommazahlen einiges an Speicher sparen lässt. Beim Convolutional Process bzw. bei der Multiplikation wird später noch deutlich, dass sich noch ein weiterer entscheidender Vorteil verbirgt.

Stochastische Quantisierung Ein weiteres genanntes Verfahren ist die stochastische Quantisierung definiert als:

$$\sigma(x) = \max(0, \min(1, \frac{x+1}{2})) \quad (7)$$

$$x^b = \begin{cases} +1 & \text{with probability } p = \sigma(x), \\ -1 & \text{with probability } 1 - p \end{cases} \quad (8)$$

Die $\sigma(x)$ Funktion liefert also eine Wahrscheinlichkeit zurück. Je nachdem, wohin diese tendiert, wird wieder auf $\{-1, +1\}$ quantisiert. Der Verlauf der $\sigma(x)$ Funktion ist aus Abbildung 26 zu entnehmen. Da sich diese Variante nicht gut auf Hardware umsetzen lässt, wird zur deterministischen Quantisierung (2.4.1) zurückgegriffen. [Cou+16]

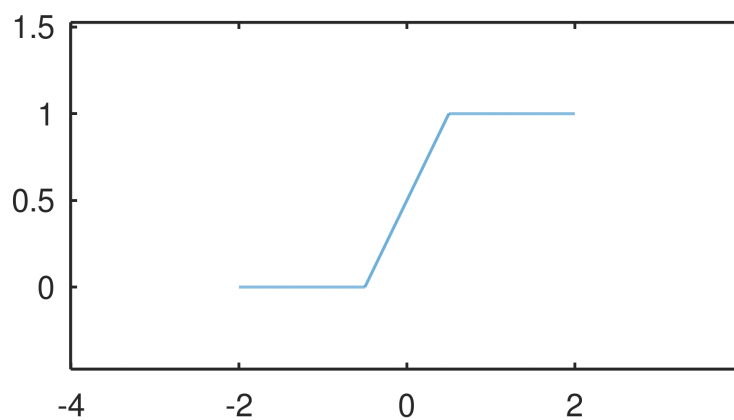


Abbildung 26: Sigmoid Funktionsverlauf

Convolutional Process Der Convolutional Process ist ein Verfahren, welches aus Convolutional Neural Networks bekannt ist und stellt die Multiplikation von Eingangs- und Gewichtsmatrix (Kernel) dar. Die Gewichtsmatrix wird wie eine Schablone auf die Eingangsmatrix gelegt. Die überlagerten Einträge werden komponentenweise multipliziert und die Summe der Produkte in die entsprechende Position abhängig von der Gewichtsmatrixlokation der Ausgabematrix geschrieben.

Dieser Prozess wird in Binären Neuronalen Netzen weitgehend vereinfacht, da sowohl Eingabematrix und Gewichtsmatrix binarisiert sind. Auf Hardwareebene kann für die Multiplikation ein einfaches XNOR-Netz genügen, wenn logisch 0 mit -1 substituiert wird. Dies verleiht den Binären Neuronalen Netzen einen wichtigen Leistungsschub im Vergleich zu Convolutional Neural Networks. [Qin+20]

XNOR			XNOR Modifiziert		
A	B	Q	A	B	Q
0	0	1	-1	-1	1
0	1	0	-1	1	-1
1	0	0	1	-1	-1
1	1	1	1	1	1

Tabelle 2: XNOR

2.4.2 Backpropagation in BNNs

Die Backpropagation in einem neuronalen Netz wird benutzt, um die Genauigkeit der Klassifizierungen durch Anpassung der Gewichtungen aller Aktivierungen entgegen der Fehlerrichtung zu verbessern. Backpropagation und Training eines NN durch stochastischen Gradientenabstieg kann nicht direkt auf binäre Netzwerke angewandt werden, da ihre streng ganzzahligen Gewichtungen nicht inkrementell angepasst werden können. Eine mögliche Lösung ist die Methode der Expectation Backpropagation EBP (oder vorausblickende Backpropagation) zum Trainieren von DNN. [Cou+16].

BinaryConnect Diese Methode, entwickelt von Courbariaux et al. verwendet Reelle Werte für die Gewichtungen, welche erst kurz vor Benutzung des NN binarisiert werden. [Cou+16].

Bitwise operations Diese bitweisen Operationen sind viel einfacher zu berechnen als Mehrbit-Gleitkomma- oder Festkomma-Multiplikation und -Akkumulation. Dies kann zu schnelleren Ausführungszeiten oder weniger erforderlichen Hardwareressourcen führen. Das Theoretisieren von Effizienzsteigerungen ist jedoch nicht immer einfach. Wenn wir beispielsweise die Ausführungszeit einer CPU betrachten, verwenden einige der hier besprochenen Artikel die Anzahl der Anweisungen als Maß für die Ausführungszeit. Der 64-Bit-x86-Befehlssatz ermöglicht es einer CPU, eine bitweise XNOR-Operation zwischen zwei 64-Bit-Registern auszuführen. Diese Operation erfordert eine einzelne Anweisung.[Cou+16].

Dies sind alles vorteilhafte Variationen von BNN Trainingsverfahren, welche wir in unser abschließliches Netzwerk einbinden können.

Straight Through Estimator Bei der Backpropagation von BNNs entsteht das Problem, dass man nicht mehr genau auf den vorherigen Gradienten kommt. Dieser wurde zuvor quantifiziert und ist somit schwer zuzuordnen. Dieses Problem ist mithilfe eines Straight Through Estimators, kurz STE, lösbar. Dieser schätzt die Gradienten einer Funktion, was dafür sorgt, dass die Quantisierungsfunktion zur neuen Aktivierungsfunktion wird. Dadurch wird die Quantisierungsfunktion bei der Backwardpropagation umgangen und man kommt mit dem binären Werten wieder auf die Reellen Werte. Diese Version benötigt eine Deterministische Quantisierung, statt einer Stochastischen Quantisierung. [Cou+16]

Explodierende oder verschwindende Gradienten Bei KNNs gibt es das Problem, dass Gradienten explodieren können, beziehungsweise auch verschwinden können. Dieses Problem gibt es auch bei Binä-

ren Neuronalen Netzen. Bei der Backpropagation, werden die Gewichtungen der einzelnen Knotenpunkte stetig verändert. Hierbei kann es bei sehr tiefen Netzen passieren, dass die Gewichtung auf einen zu hohen oder niedrigen Wert verändert werden. Dies passiert dadurch, dass sich die Werte von Schicht zu Schicht exponentiell verändern. Dies sorgt dafür, dass in sehr selten Fällen einige Werte, verschwindend klein oder groß werden. Diese zu großen oder zu kleinen Werte, nennt man explodierende oder verschwindende Gradienten. Da diese Werte besondere Fälle sind und sich sehr groß von den durchschnittlichen Werten unterscheiden, sorgen Sie für große Probleme im Neuronalen Netz. Dies sorgt dafür, dass das Neuronale Netz aus den aktuellen Daten nichts lernen kann und somit nicht trainiert werden kann. Sie verhindern also den Lernprozess des Neuronalen Netzes und machen das ganze Netz instabil. [PMB13]
 Hiergegen kann man unter anderem Gradienten Clipping betreiben. Dies macht nichts anderes als, Gradienten die zu klein oder zu groß werden, wieder zurück auf einen vorher festgelegten Wert zu setzen.

2.4.3 Optimierung

Binären Neuronalen Netzen entgehen durch die strenge Quantisierung Informationen, was sich auf die Präzision auswirkt. Zudem werden sie auf leistungsschwacher Hardware eingesetzt, wodurch es von Nöten sein kann, die Performance ein Stück weit zu steigern. Seit der Erstveröffentlichung Binärer Neuronaler Netze von [Cou+16], wurden einige Erkenntnisse gewonnen, die teilweise im Projekt verwendet wurden.

Partial Binarization Es besteht die Möglichkeit, falls noch genug Performance und Ressourcen übrig sind, diese für eine höhere Genauigkeit einzutauschen, indem nur unwichtige Layer quantisiert und die wichtigen bei voller Präzision belassen werden. [SL19] Im verwendeten *CNN* Model ist außer den Ein- und Ausgaben alles quantisiert. [Umu+16]

Padding Padding ist ein bereits genutztes Verfahren in Convolutional Layern, wo der äußere Rand der Ausgabematrix mit Konstanten (i.d.R. mit 0) gefüllt werden, sodass die Dimensionen beibehalten werden und die Ränder durch den Kernel besser erfasst werden können. Dies erhöht entsprechend die Genauigkeit des Netzes. [Ngu+19] Bei Binären Neuronalen Netzen gibt es andere Verfahren für das Padding, da die Nutzung von 0 nicht vorgesehen ist. [SL19] [Guo+18]

- Mit +1 füllen (+1 Padding)
- Mit -1 füllen (-1 Padding)
- Abwechselnd (Odd-Padding oder Even-Padding)
- Abwechselnd zwischen Channels (Odd-Even-Padding) (Abbildung 27)

Es stellte sich heraus, dass das Odd-Even-Padding die geringste Fehlerrate aufweist (Tabelle 3). [Guo+18]

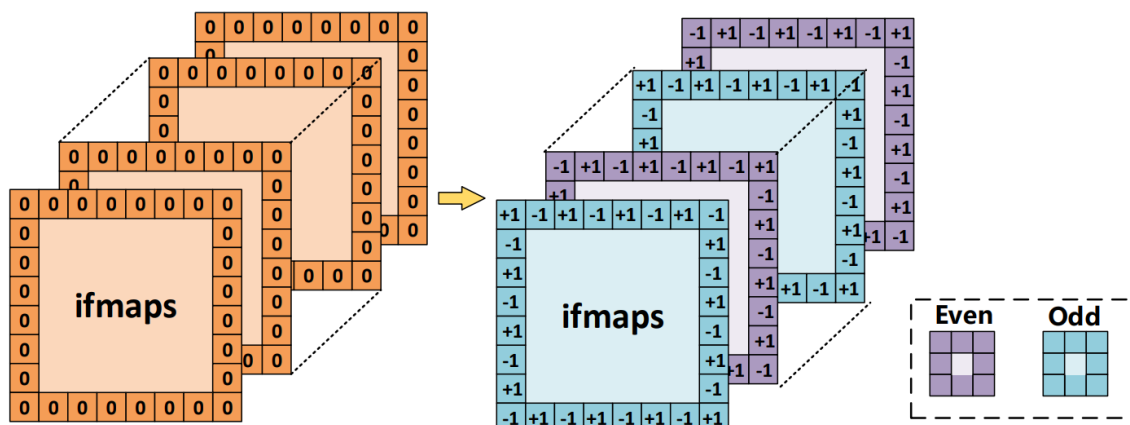


Abbildung 27: Odd-Even-Padding [Guo+18]

BNN Model	0-Padding	+1-Padding	Odd-Padding	Odd-Even-Padding
BNN-SVHN	3,14%	3,36%	3,28 %	3,15%
BNN-CIFAR10	11,39%	13,24%	12,42%	11,25%

Tabelle 3: Fehlerrate zwischen Padding-Methoden in BNNs [Guo+18]

Es wird in [Umu+16] nicht erwähnt, ob Padding im *CVV* Model [Papa] genutzt wird und aus dem Quellcode ist es ebenfalls nicht zu entnehmen. Es ist daher auszugehen, dass dieses Verfahren im Projekt nicht genutzt wurde.

Akkumulation Für das Summieren der Produkte aus der komponentenweisen Multiplikation im Convolutional Process, kann statt einer normalen Addition, auf die *Popcount* Funktion gesetzt werden. Diese berechnet das Hamming-Gewicht, ergo zählt sie die Einsen in einem Vektor von binären Werten. Die *Popcount* Funktion lässt sich besser auf die Hardware übertragen und steigert somit nochmals die Performance im Convolutional Layer. Bei 200MHz und 128 Bits hat man für die konventionelle Art 759Luts und 84FFs und für die *Popcount* Variante 376Luts und 29FFs benötigt [Umu+16]

```
foreach(i in width){
    Sum+= A[i]*B[i]
}
```

Listing 2: Herkömmliche Art

```
Product = Xnor(A, B) // (Xnor(10010, 01111)) = 00010
Sum = Popcount(Product) // Popcount(00010) = 1
Sum = 2*Sum-width // 2*1-5 = -3 = -1-1-1+1-1
```

Listing 3: Effizientere Art

Die XNOR-Popcount-Multiplikation wird durch die Optimierungen beim Übersetzen in IP-Blocks durch das *Synthesize.ipynb*-Notebook (angelehnt an [FINa]) im Projektrepository genutzt.

Batchnorm-Aktivierung als Schwellenwert *Signum* als Aktivierungsfunktion lässt nur Eingänge > 0 durch. Diese Eigenschaft kann man nutzen, um einen Schwellenwert für die *Batchnorm*-Funktion auszurechnen während des Trainings. Diesen Benutzt man dann bei der Inferenz anstelle der Berechnung der aufwändigen *Batchnorm*-Funktion. [Umu+16]

$$a_k^b = \text{Sign}(\text{BatchNorm}(a_k, \Theta_k)) \quad (9)$$

Aktivierungsfunktion

$$\text{BatchNorm}(a_k, \Theta_k) = \gamma_k \cdot (a_k - \mu_k) \cdot i_k + B_k \quad (10)$$

Batchnormalisierungsfunktion

$$\Theta_k = (\gamma_k, \mu_k, i_k, B_k) \quad (11)$$

Während Training angepasste Parameter der Batchnormalisierung

$$\text{BatchNorm}(\tau_k, \Theta_k) = 0 \Rightarrow \tau_k = \mu_k - \left(\frac{B_k}{\gamma_k \cdot i_k} \right) \quad (12)$$

Funktion 0 setzen und auflösen nach a_k bzw. τ_k , da hier der Schwellenwert der Signum-Funktion liegt

$$S = S_{+1} + S_{-1} \quad (13)$$

Summe aus Anzahl aller +1 und -1 Werte ist Gesamtanzahl aller Synapsen

$$\tau_k^+ = \frac{\tau_k + S}{2} \quad (14)$$

Berechnung des Mittelwerts aller Synapsen mit Schwellenwert τ

Auch die die Batchnorm-Aktivierung als Schwellenwert wird im FINN Framework genutzt [Umu+16] und kann aus dem *Synthesize.ipynb*-Notebook (angelehnt an [FINa]) aus dem Projektrepository entnommen werden, wo es auch *MultiThreshold* genannt wird.

2.4.4 Erkenntnisse aus BNN Entwicklung

Für das Projekt wurden die Paper [Cou+16], [Ras+16] und [THW17] gesichtet, mit dem Zweck gegebenenfalls BNN in diesem Projekt zu optimieren. Deswegen werden in diesem Abschnitt die Erkenntnisse und dadurch resultierende mögliche Verbesserungen/Optimierungen von den obigen Paper in eigenen Worten zusammengefasst. Doch verwendet wurden diese Optimierungen im Projekt nicht.

The Original BNN Das Original BNN [Cou+16] ist das erste BNN das nur 1-Bit für Gewichte und Aktivierungen während der Inferenz als auch dem Training verwendet, deswegen bauen nahezu alle BNN Forschergruppen auf diesem Paper auf und somit ist dies eine fundamentale Grundlage von BNN. Die Methoden aus diesem Paper wurden im Kapitel 2.4.1 und 2.4.2 beschrieben. Die Erkenntnisse vom Original BNN sind:

- Stochastische Binarisierung: Die Stochastische Binarisierung erhöht die Genauigkeit des BNN. Der Nachteil dabei ist dass durch die Stochastische Binarisierung zu jedem Wert eine Wahrscheinlichkeit berechnet werden muss und somit sowohl Inferenz als auch das Training belastet wird. Zudem ist die Stochastische Binarisierung schwieriger in Hardware umzusetzen als die Deterministische Binarisierung, sodass daher die Deterministische Binarisierung meistens verwendet wird.
- Shifted Base Methods: Die Shifted Base Methods sind Approximationen von den Original Funktionen, wie z.B der Batchnormalisierungsfunktion. Dabei ist der Vorteil das diese Approximationen nahezu keine Multiplikationen mehr brauchen, somit nicht so rechenaufwändig sind und im Vergleich zu den Original Funktionen es keine Unterschiede seitens Genauigkeit des BNN gibt.
- GPU Kernel Optimierung: Im Original BNN wird eine GPU Kernel Optimierung vorgenommen dabei wird ein GPU Kernel welcher auf Multiplikationen von reellen Zahlen ausgelegt ist, entsprechend auf binäre Multiplikationen (XNOR) optimiert, dadurch wird eine Performance Verbesserung von 7x gegenüber einer unoptimierten Version auf dem Datensatz MLP erreicht.

XNOR-Net Das XNOR-Net [Ras+16] ist eine Methode, basierend auf dem Original BNN [Cou+16]. XNOR-Net wurde konzipiert um auf dem ImageNet Datensatz gute Genauigkeiten zu erzielen. Dabei sind folgende Erkenntnisse hervorgekommen:

- Skalierungsfaktor: Der Skalierungsfaktor wird vom XNOR-Net erstmals vorgestellt und verbessert die Genauigkeit des BNN. Dabei hat der Skalierungsfaktor den Zweck die reellen Gewichte und reellen Aktivierungen zu approximieren um somit den Verlust durch das Binarisieren der Werte zu schmälern. Hierbei hat der Skalierungsfaktor im XNOR-Net aber einen sehr großen Nachteil in der Umsetzung, denn es muss jeweils ein Skalierungsfaktor für Aktivierungen und Gewichte berechnet werden und das für jedes Skalarprodukt im Convolutional Layer, sowohl während des Trainings als auch der Inferenz. Eben durch dieses ständige berechnen wird die Umsetzung von diesem Skalierungsfaktors in anderen Paper kritisiert, denn durch diese Umsetzung belastet man das Training als auch die Inferenz erheblich.
- Anordnung der Layer: Im XNOR-Net wurde mit der Anordnung der Schichten herumexperimentiert, genauer wo man das Pooling Layer in der Netzarchitektur platzieren sollte. Man fand heraus dass das Pooling Layer nach dem Convolutional Layer platziert werden sollte. Mit der Begründung das Daten in BNN bevor diese in das Convolutional Layer kommen, Binär (-1 oder 1) sind und man dadurch mittels Max-Pooling nicht viel erreicht, darüber hinaus stützt man auf besseren Resultaten, seitens Genauigkeit, bei der Anwendung des Pooling Layers auf reellen Zahlen.

Tang Tang et al. [THW17] haben keine neue Netzarchitektur entwickelt sondern haben die Netzarchitektur AlexNet binarisiert und sich mit den Problemen von BNN beim Datensatz ImageNet, seitens Genauigkeit beschäftigt. Zudem wurden neue Ideen vorgestellt die in anderen BNN Paper benutzt werden und es wurde das Training von BNN genauer unter die Lupe genommen.

- Skalierungsfaktor: Tang et al. [THW17] haben die Umsetzung des Skalierungsfaktors vom XNOR-Net kritisiert und es daraufhin besser umgesetzt. Anders als beim XNOR-Net wird der Skalierungsfaktor hier nicht mehr bei jedem Skalarprodukt im Convolutional Layer berechnet, sondern es wurde das PreLU-Layer als neue Schicht hinzugefügt, wo ein Lernfaktor gelernt und auch gleichzeitig angewandt wird. Als Resultat wird das Training und die Inferenz nicht mehr so erheblich belastet wie beim XNOR-Net und man verbessert dadurch die Genauigkeit des BNN.
- Kleinere Lernrate: Tang et al. [THW17] haben beobachtet das NN welche reelle Gewichte und reelle Aktivierungen nutzen eine Lernrate von 0.01-0.05 haben, doch damit kommen BNN nicht so ganz gut klar, was sich an einer niedrigeren Genauigkeit abzeichnet. Dieses Problem haben Tang et al. [THW17] erkannt und daraufhin mit der Lernrate experimentiert, dadurch kam heraus das eine Reduktion der Lernrate auf z.B 0,0001 die Genauigkeit um ~20% erhöht und dadurch sich eine kleinere Lernrate bei BNN lohnen tut.
- FC-Layer binarisieren: Um das Modell von BNN noch kompakter zu machen haben Tang et al. [THW17] das letzte Fully Connected-Layer (FC-Layer) binarisiert, denn im letzten FC-Layer befinden sich die meisten Parameter, dies liegt an der Zunahme von Filtern mit jeder Schicht, benötigt für das Convolutional Layer. Dabei muss für das Binarisieren des letzten FC-Layers ein Scale-Layer vor dem letzten FC-Layer hinzugefügt werden. Damit lernt das NN einen kleinen Faktor mit welchem die Reichweite der Werte geschmälert wird und somit die Sättigung der Klassifizierungsfunktion (Softmax) vermieden wird. Als Resultat nimmt das Modell des BNN deutlich weniger Speicherplatz ein und der Einsatz von BNN auf ressourcenschwachen Geräten wird erleichtert.
- Regularizer: Tang et al. [THW17] konnten beobachten das die Kostenfunktion von BNN nicht so stabil und glatt ist. Darunter litt die Genauigkeit des BNN, deswegen wurde ein Regularizer hinzugefügt, um der Kostenfunktion zu helfen und somit diese glatter und stabiler zu machen. Dabei wird gleichermaßen die Generalisierung des BNN verbessert. Um dies zu erreichen wird ein von Lambda skaliertes Term zu der Kostenfunktion addiert, welcher vom NN gelernt wird.

2.4.5 Erkenntnisse aus BNN Implementierung auf FPGA

Die FPGA Implementierung eines Binären Neuronalen Netzes, wurde bereits von [SL19] mit verschiedenen FPGA-Boards getestet. Unter anderem wurde das Zynq7 020 benutzt auf welches hier weiter eingegangen wird. Die Tests liefen mit verschiedenen Trainingsdatensätzen durch. So wurde mit dem MNIST-Datensatz, dem SVHN-Datensatz und dem Cifar10-Datensatz trainiert.

- MNIST-Datensatz: Mit dem Zynq7 020 wurde nach dem Training eine Genauigkeit von 97,69% erreicht. Zu diesem Datensatz wurde nur ein einziges Zynq-Board trainiert. Außerdem hat das Zynq-Board einen Stromverbrauch von 2,5W.
- SVHN-Datensatz: Bei diesem Datensatz wurden 2 verschiedene Netztopologien auf dem Zynq-Board trainiert. Diese erreichten unterschiedliche Genauigkeiten. Die erreichten Genauigkeiten lagen bei 96,9% und 97%. Der Stromverbrauch lag hier bei 3.2W
- CIFAR10-Datensatz: Bei diesem Datensatz wurden mehrere Netztopologien auf dem Zynq-Board trainiert. Die Genauigkeiten schwangten hier im Wertebereich von 80,1% bis 88,61%. Die beste Topology erreichte also eine Genauigkeit von 88,61% und benötigte für die Durchführung 3,3W.

Da die Datensätze unterschiedliche Größen und Komplexität haben, ist der vorhandene Genauigkeitsverlust beim Cifar10-Datensatz zu erwarten. Außerdem sieht man die erwähnten Positiven und Negativen Eigenschaften von Binären Neuronalen Netzen. So bekommen wir geringere Genauigkeiten im Austausch für schnellere Ergebnisse.

2.5 CNN mit Tensorflow

Geschrieben von Mattia Uhlenbrock, Stanislav Voytas, Reena Wichmann, Fannese Tchanga

Für den Einstieg in die Entwicklung eines CNN haben wir das Machine-Learning Framework Tensorflow verwendet. Dieses wurde von Google zum Entwerfen und Trainieren von Deep-Learning-Modellen entwickelt. Es lässt sich sowohl auf CPU, GPU, als auch TPU (Tensor Processing Unit) ausführen. Tensorflow unterstützt mehrere Client-Sprachen wie JavaScript, Python, C++, Go, Java und Swift.[Hei17]

Die High-Level API Keras ist direkt in Tensorflow integriert. Sie ermöglicht einen schnellen Einstieg in das Entwickeln von Deep-Learning Modellen, während Tensorflow als solches primär für erfahrene Entwickler geeignet ist. In Keras sind bereits einige gängige Datensätze enthalten, welche sich mit wenig Aufwand zum Trainieren eines NN laden lassen. Üblicherweise werden die einzelnen Schichten in einem „Sequential-Model“ oder einer Unterklasse von Model definiert. In diesem werden im Falle eines CNN mehrere Convolutional-Layer in Kombination mit Pooling-Layern „aufgeschichtet“. Zusätzlich lässt sich für jeden Layer eine Aktivierungsfunktion definieren, die entsprechend auf die einzelnen Neuronen angewendet werden. Anschließend werden die Dense-Layer im Modell ergänzt. Der Funktion compile() werden Verlustfunktion, Optimierer und optional die Metrik für das Modell übergeben. Das kompilierte Modell kann mit der fit() Funktion und mit übergebenen Trainings-Datensatz, sowie optionalen Test-Datensatz trainiert werden. Wenn ein Test-Datensatz übergeben wird, findet nach jeder Epoche des Trainings eine Validierung mit dem Test-Datensatz statt.[Mir21, S. 478 ff.]

Für die Implementierung eines Beispiel CNN auf Basis des Tensorflow Tutorials „Convolutional Neural Network (CNN)“ wurde der CIFAR10 Datensatz verwendet. Dieser ist bereits in Keras eingebettet. Der Datensatz enthält 60.000 RGB-Farbbilder, die in zehn Klassen mit 6.000 Bildern je Klassen eingeteilt sind. Des Weiteren ist der Datensatz in 50.000 Trainings- und 10.000 Testbilder unterteilt.[Tea21]

2.6 Brevitas

Geschrieben von Stanislav Voytas, Mattia Uhlenbrock, Reena Wichmann

Für Machine Learning auf leistungsschwacher Hardware, ist die Quantisierung ein sehr entscheidender Faktor. Hierbei werden die Wertebereiche der Gewichte zwischen den Neuronen auf eine bestimmte Menge von Bits beschränkt. Dadurch sinkt nicht nur die Größe des Modells sondern die Inferenz wird durch das Arbeiten mit kleineren Datentypen beschleunigt. Das Training hingegen dauert bei quantisierten Modellen länger.[WS20]

Brevitas ist eine PyTorch Bibliothek, welche ein quantisierungsbewusstes Training eines Modells ermöglicht. Die Bibliothek stellt eine Menge neuer Layer zur Verfügung, und ermöglicht für jeden dieser Layer eine Bitbreite für die Gewichte vorzugeben. Da Brevitas eine PyTorch Erweiterung ist verläuft das Training gleichartig, lediglich bei der Erstellung des Modells müssen die Brevitas Layer verwendet werden. In bestimmten Fällen ist es auch möglich, die Brevitas Layer mit herkömmlichen PyTorch Layern zu kombinieren. Beim Verwenden von Brevitas eröffnet sich außerdem die Möglichkeit weitere quantifizierte Layer zu implementieren, die eventuell in PyTorch, aber nicht als quantifizierte Variante in Brevitas, enthalten sind.[Papb]

Ferner ermöglicht Brevitas ein trainiertes Model im ONNX Format zu exportieren. Das ONNX Format wird vom FINN Framework unterstützt, welches wiederum die Umwandlung des Modells für das FPGA ermöglicht. Anzumerken ist, dass Brevitas noch in der Entwicklung und daher noch unvollständig dokumentiert ist.

2.7 ONNX

Geschrieben von Felix Müller

Open Neural Network Exchange (ONNX) ist ein standartisiertes Austauschformat für Modelle von künstlichen Neuronalen Netzen, welches als Alternative zu den vielen proprietären Dateiformaten der Frameworks entwickelt wurde. Die interne Realisierung erfolgt durch die Darstellung als gerichtete Graphen, welche Standart- und eigen erstellte Operation miteinander verketteten. Jede Operation kann neben Ein- und Ausgängen Parameter, wie etwa die trainierten Gewichten oder andere Konstanten enthalten. Ein sehr simples Beispiel kann in Abb. 28 betrachtet werden, welcher alle Werte im Bereich 0 bis 255 einer $3 \times 32 \times 32$ großen Matrix auf -1 bis 1 normalisiert. Im ersten Schritt wird die Eingangsmatrix durch die Operation *Div* durch den Wert 255 des konstanten Parameters *B* dividiert, folgend durch *Mul* mit dem Faktor 2 multipliziert und zuletzt um 1 durch *Sub* subtrahiert. Neben diesen sehr simplen Operationen werden auch komplexere Operationen unterstützt, wie etwa die Matrizenmultiplikation oder Faltungen mit Filtern.

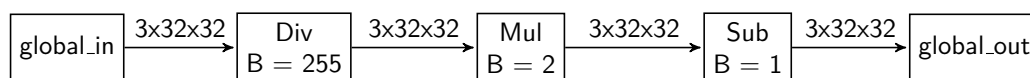


Abbildung 28: ONNX Beispielgraph

Der große Vorteil ist die Verwendbarkeit verschiedenster Frameworks für die Erstellung der Modelle, welche folgend mit unterschiedlichsten Interpretern ausgeführt werden können. Die Einordnung zwischen diesen zwei Schritten ist in Abb. 29 dargestellt.

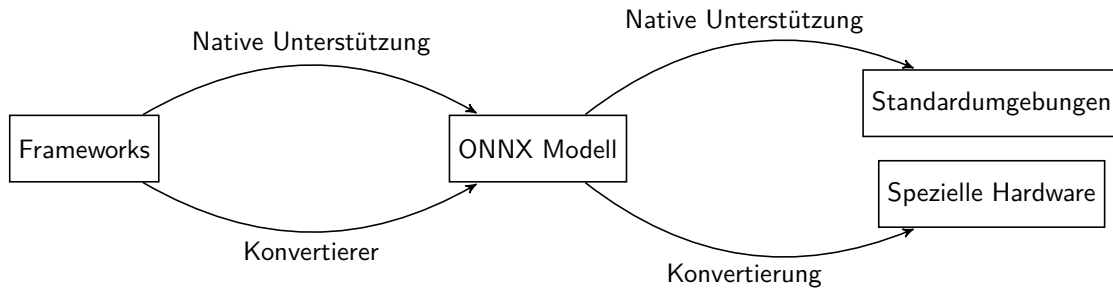


Abbildung 29: Einordnung von ONNX

Für die Erstellung und Verwendung von ONNX muss immer zwischen nativer Unterstützung und Konvertierung unterschieden werden. Die native Unterstützung für die Erstellung erfolgt durch Frameworks, welche die ONNX Modelle ohne externe Werkzeuge exportieren können. Ähnliches gilt für die Ausführung von ONNX Modellen, denn wenn ein Framework ONNX nativ unterstützt, kann es die Modelle ohne Vorverarbeitung ausführen. Die Besonderheit bei der Ausführung ist der mögliche Einsatz in unterschiedlichsten Umgebungen, was mit den proprietären Formaten nur teilweise oder bestimmte Umgebungen nur von bestimmten Frameworks unterstützt wurden. ONNX macht es möglich spezialisierte Werkzeuge für die Umwandlung der Modell zu nutzen, welche beispielsweise Hardware unterstützt, die gewissen Einschränkungen unterliegt, wie etwas Mobiltelefone oder generell schwacher Hardware [Fou21]. Ein solches Konvertierungswerkzeug ist FINN, welches es ermöglicht quantisierte Modelle in FPGA Schaltungen umzuwandeln.

2.8 FINN

Geschrieben von Felix Müller, Tobias Nießen

FINN ist ein Framework zur Übersetzung von trainierten, quantisierten neuronalen Netzen in Hardware-schaltungen auf FPGAs. Die Erstellung und das Training der Netzmodelle erfolgt mit dem Brevitas Framework (siehe Kap. 2.6). Zielplattform der Hardware-schaltungen sind größtenteils FPGAs der PYNQ-Familie und ALVEO-Familie, welche alle auf System-on-Chips (SoC) des Herstellers Xilinx basieren, die wiederum hauptsächlich aus einer CPU und einem FPGA zusammengesetzt sind.

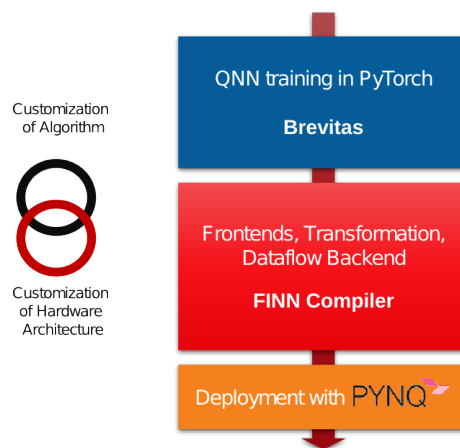


Abbildung 30: FINN Softwarestack [Xil]

Zur Generierung der Hardwarebeschaltung, werden von Hand erstellte Funktionen in C++ verwendet, welche mittels Vivado HLS für FPGAs synthetisiert werden können und in einer Bibliothek namens „finn-hlslib“ zusammengefasst sind. Diese Funktionen implementieren häufige Bestandteile von neuronalen Netzen, wie etwa die Matrizenmultiplikation. Ziel des Übersetzers ist es, das Eingangsmodell aus Brevitas in Aufrufe für die vorhandenen Funktionen umzuwandeln. Diese Umwandlung basiert stark auf dem ONNX Datenformat (siehe 2.7), welches die Netze als gerichtete Graphen repräsentiert. Um nun von dem Eingangsmodell zu den Funktionsaufrufen zu gelangen, werden Graphentransformationen vollzogen, welche die verwendeten Operationen im Modell zusammenfassen und gegen solche austauschen, die eine entsprechende Funktion in der HLS Bibliothek haben.

Das Hauptunterscheidungsmerkmal des FINN Frameworks ist die Umsetzung als Datenfluss. Klassische Hardwarebeschleuniger für neuronale Netze verwenden meist eine Matrix aus generischen Blöcken für häufig benötigte Funktionen, wie etwa Multiplizieren und Akkumulieren (MAC). Mithilfe dieser Blöcke werden die Netze Schicht für Schicht ausgewertet. FINN hingegen generiert eine spezialisierte Schaltung für genau eine Netzarchitektur, welche durch die Verwendung von FPGAs leicht ausgetauscht werden kann. Diese Schaltung besitzt für jede Schicht im Netz eigene Hardware, welche miteinander verkettet sind. Dadurch können die Daten durch das Netz fließen und mehrere Layer gleichzeitig in einer Pipeline verarbeitet werden, wodurch eine deutlich höhere Effizienz möglich ist. Falls genug Hardwareressourcen zur Verfügung stehen, kann eine gesamte Schicht innerhalb eines Taktzyklus verarbeitet werden, sodass kein Zwischenspeicher nötig ist. Ist dies nicht der Fall, dann können Schichten gefaltet werden und so der Verbrauch an Logikressourcen verringert werden. Jedoch wird dann ein Zwischenspeicher benötigt, da die Verarbeitung mehrere Takte benötigt und Zwischenergebnisse potentiell nicht direkt an die nächste Schicht weitergegeben können [Blo+18]. Dieser Datenfluss ist in Abb. 31 dargestellt.

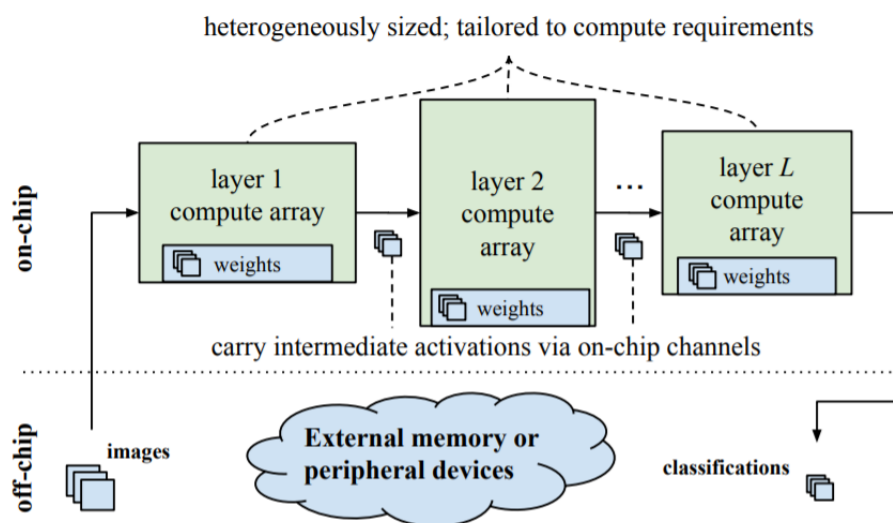


Abbildung 31: FINN Datenfluss [Xil]

2.9 TinyML

Geschrieben von Stanislav Voytas, Reena Wichmann, Tuncer Catalkaya

TinyML beschreibt einen Forschungsbereich des Machine Learnings, bei dem kleine neuronale Netze auf einfachen Mikrocontroller ausgeführt werden sollen. Diese Netze sollen wenig Speicher und möglichst wenig Batterie verbrauchen, damit das neuronale Netz unangetastet über einen längeren Zeitraum laufen kann. Zu dem Anwendungsbereich des TinyML gehören eingebettete Systeme, in denen Mikrocontroller mit bereits trainierten neuronalen Netzen eingebaut werden können. Hierbei soll es ermöglicht werden, maschinelles Lernen kostengünstig für Systeme zur Verfügung zu stellen.

Projekte, die zum Bereich von TinyML gehören, werden in TensorFlow Lite für Mikrocontroller implementiert, welches extra für die Anwendung auf speicherarmen Mikrocontroller entwickelt wurde. Dieses Framework ist eine Weiterentwicklung von TensorFlow Lite, welches für die Entwicklung auf leistungsstärkeren Geräten mit eingebauten Betriebssystem gedacht ist und wiederum eine Weiterentwicklung der Open Source Plattform TensorFlow ist. TensorFlow ist mittlerweile eine Open Source Plattform und wurde ursprünglich von Google entwickelt.

Um ein TinyML Modell zu trainieren, muss das Modell so entworfen werden, dass es auf den geringen Speicher des entsprechenden Mikrocontrollers angepasst ist. Es muss zudem beachtet werden dass nicht alle Funktionen von TensorFlow zu Verfügung stehen. Mit dem TensorFlow Lite Konverter geschieht die Umwandlung in ein TensorFlow Lite Modell, welches wiederum in ein C-Byte-Array konvertiert wird. Dieses wird auf dem Mikrocontroller gespeichert und kann dort ausgeführt werden. Das Modell wird auf einem anderen leistungsstarken Medium trainiert und auf dem eigentlichen Gerät lediglich ausgeführt. [WS20] Auf der GitHub Seite von Tensorflow Lite für Mikrocontroller stehen Beispielprojekte zur Verfügung.

Im Vorgängerprojekt wurde ein vortrainiertes Netz verwendet, welches mit dem FINN Compiler ausgeführt werden konnte. Aus diesem Grund wurde die Implementierung auf das FINN Framework ausgelegt. Aus der FINN Dokumentation lässt sich schließen, dass ein mit Brevitas trainiertes Modell ebenfalls mit dem FINN Compiler übersetzt werden kann. Da es keine vergleichbaren dokumentierten Erfahrungen für die Ausführung eines TinyML auf FPGAs gibt, wurde sich in diesem Projekt dazu entschieden, den Ansatz mit TinyML nicht weiter zu verfolgen und sich stattdessen auf die Umsetzung mit Brevitas zu konzentrieren. Es lässt sich aber herausstellen, dass TinyML eine Möglichkeit darstellt, die durch Implementierung mit dem Framework TensorFlow genutzt werden könnte. Ein mögliches Toolkit hierfür wäre LeFlow. [DW18]

2.10 AXI Bus

Geschrieben von Viktor Chechulin, Philipp Altnickel

In diesem Kapitel werden die Grundlagen vom AXI Bus beschrieben. Für das Projekt ist besonders das Kapitel „2.10.2 Datenübertragung in AXI4“ wichtig.

2.10.1 Was ist AXI4?

Das Advanced eXtensible Interface oder kurz AXI genannt, ist ein Teil des AMBA (Advanced Microcontroller Bus Architecture)-Standards [ARM10] und eignet sich besonders gut für die On-Chip Kommunikation. AXI4 ist dabei die zweite Version von AXI und umfasst drei Protokolle: AXI4, AXI4-Lite und AXI4-Stream [XIL11].

Das grundlegende Prinzip von AXI für die Kommunikation ist dabei sehr simpel, die Komponenten welche miteinander kommunizieren wollen werden im Design in Master und Slave unterteilt, wobei der Master Daten vom Slave anfordern (Read), als auch diesen Daten zuschicken (Write) kann. Es handelt sich dabei um direkte 1:1 Verbindungen, d.h. ein Bus ermöglicht die Kommunikation zwischen genau einem Master und einem Slave. Aus diesem Grund gibt es in mittels Vivado erstellten Hardwareschaltungen sogenannte „AXI Interconnects“, die bei mehreren vorhandenen Busverbindungen die Kommunikation regeln.

2.10.2 Datenübertragung in AXI4

Die Datenübertragung in AXI4 ist Adressbasiert oder auch Memory Mapped genannt. Hierbei wird ein fester Bereich im Speicher reserviert. In Folge dessen kann das Master Interface dann in diesen festen Bereich via Speicheradresse, durch den Slave zugreifen, Daten ablegen oder auch abrufen. Außerdem besitzt AXI4 die Burst-Funktionalität und kann bis zu 256 Bursts während einer Datenübertragung durchführen, dabei gibt die Burstgröße die Menge an Datenworten an die mit einer Startadresse übertragen werden, ohne das jeweils erneut eine Adresse übertragen werden muss.

Für die Datenübertragung mittels AXI4 werden insgesamt 5 Kanäle gebraucht um Schreib (Write) und Lese (Read) Vorgänge zu ermöglichen:

- Read Address Channel
- Write Address Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

Schreibvorgang In der folgenden Abbildung ist die Funktionsweise von einem AXI4 Write zu sehen.

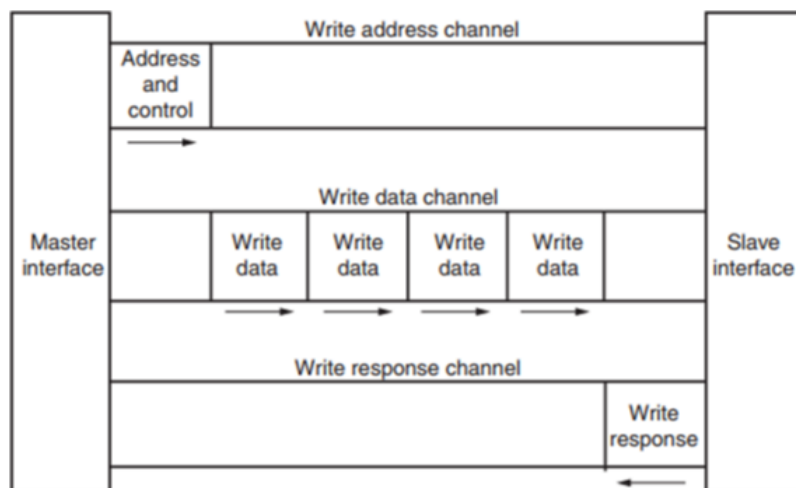


Abbildung 32: AXI4 Write [XIL11]

Für den Schreibvorgang werden die Channel Write Data Channel, Write Address Channel und Write Response Channel benötigt. Im ersten Schritt sendet der Master sowohl Adresse als auch Kontrollinformationen wie Burstgröße, über den Write Address Channel an den Slave und ebenfalls auch die zu schreibenden Daten über den Write Data Channel. Danach quittiert der Slave mit einer Antwort über den Write Response Channel, ob der Vorgang erfolgreich war oder nicht.

Lesevorgang Im folgenden ist die Funktionsweise von einem AXI4 Read zu sehen.

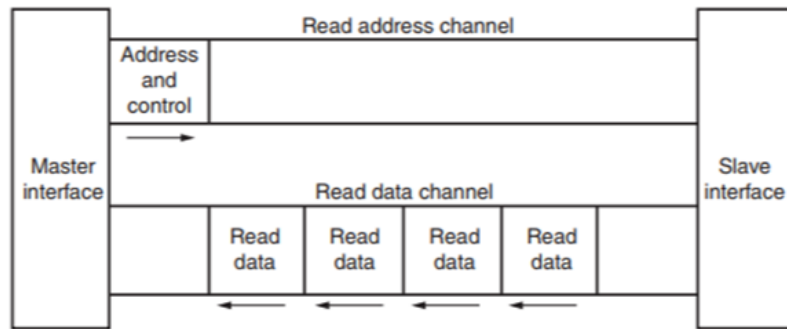


Abbildung 33: AXI4 Read [XIL11]

Für den Lesevorgang werden die Channel Read Address Channel und Read Data Channel benötigt. Im ersten Schritt sendet der Master sowohl Adresse als auch Kontrollinformationen wie Burstgröße über den Read Address Channel an den Slave. Danach sendet der Slave die angeforderten Daten über den Read Data Channel an den Master und der Vorgang ist beendet. Im Gegensatz zum Schreibvorgang werden hier nur zwei Channel benötigt, da der Master hier über den Read Data Channel ohnehin Rückmeldungen vom Slave bekommt. Beim Schreiben braucht es einen weiteren Channel, da es sonst keinen Channel vom Slave in Richtung Master für Antworten geben würde.

2.10.3 AXI4-Lite

AXI4-Lite ist ein Memory Mapped Interface und eine vereinfachte Variante von AXI4. Dabei nutzt AXI4-Lite die selben 5 Kanäle für Lese- und Schreibvorgänge wie auch AXI4. Der Unterschied liegt in den übernommenen Signalen, denn AXI4-Lite übernimmt sehr wenige der von AXI4 vorhandenen Signalen und dementsprechend sinkt die Komplexität auf ein Minimum. Zudem kann AXI4-Lite keine Bursts durchführen, deswegen ist die Datenübertragung langsamer als in AXI4 und zugleich auf eine Übertragung zur gleichen Zeit beschränkt. Durch eben diese Vereinfachungen, dem langsamen Datendurchsatz und der Leichtigkeit des Interfaces eignet sich AXI4-Lite sehr gut für die Übertragung von Konfigurationsdaten.

Die grundlegende Funktionsweise von Lese -und Schreibvorgängen in AXI4-Lite ist identisch zu der in AXI4 (Abbildung 32 und 33), welche im Kapitel „2.10.2 Datenübertragung in AXI4“ beschrieben wurden.

2.10.4 AXI4-Stream

Das AXI4-Stream ist ein Interface das auf die Adressierungsphase verzichtet, somit nicht Memory Mapped ist und eine uneingeschränkte Burst-Funktionalität hat (beliebig wählbare Burstgröße). Deswegen hat AXI4-Stream einen hohen Datendurchsatz und eignet sich sehr gut für die Übertragung von Nutzdaten. Die Funktionsweise der Datenübertragung in AXI4-Stream ist dabei sehr simpel, zusehen in Abbildung 34.

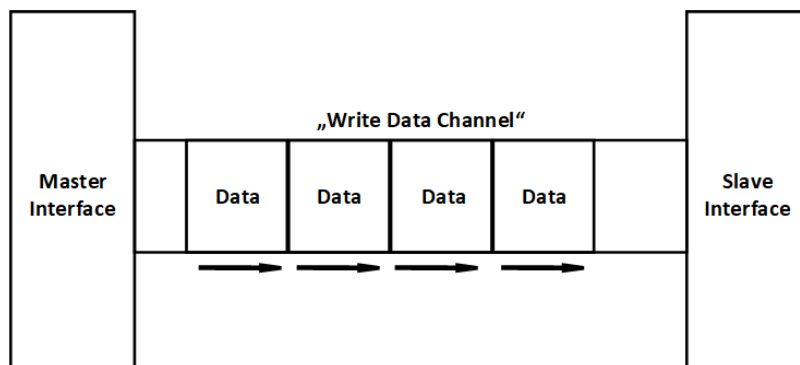


Abbildung 34: Datenübertragung in AXI4-Stream

In AXI4-Stream gibt es keine Lesevorgänge, lediglich Schreibvorgänge, denn es werden einfach Daten vom Master zum Slave verschickt, dabei ähnelt der Kanal zum Verschicken von Daten in AXI4-Stream dem eines Write Data Channel in AXI4.

2.11 Vorhandene Hardware

Geschrieben von Tobias Nießen

In diesem Kapitel wird die, durch die abgeschlossenen Vorgängerprojekte, bereits vorhandene Hardware vorgestellt und deren Anwendungsweise beschrieben.

2.11.1 Digilent PYNQ-Z1

Geschrieben von Tobias Nießen

Durch vorangegangene Entwicklung in zwei Vorgängerprojekten ist bereits ein Board vorhanden: das PYNQ-Z1 von Digilent (siehe Abb. 36). Das PYNQ-Z1 hat einen ARM Cortex-A9 Prozessor, welcher zwei Kerne besitzt. Auf diesem Prozessor läuft ein, von dem Vorgängerprojekt programmiertes, Processing System, welches dafür zuständig ist, das Video in Bilder aufzuteilen und diese an den FPGA weiterzugeben. Außerdem berechnet das Processing System anhand der vom FPGA ausgegebenen Wahrscheinlichkeiten die Fahrtrichtung und steuert die Motoren.

Zudem ist eine Programmable Logic in Form eines Zynq XC7Z020-FPGA verbaut. Programmable Logic bedeutet, dass, im Gegensatz zu integrierten Schaltungen, nicht fest definiert ist, wie die interne Logik dieses Chips verschaltet ist. Es können logische Schaltungen frei entwickelt und mittels des FPGA realisiert werden. Der Vorteil hierbei ist, dass so bestimmte Berechnungen schneller und signifikant energieeffizienter durchgeführt werden können, als mit einer CPU.

Diese Programmable Logic besteht aus einzelnen, einstellbaren Logikblöcken (siehe Abbildung 35) und einer Schaltmatrix, welche zwischen diesen Logikblöcken liegt und frei veränderliche Verbindungen zwischen diesen realisieren kann. Außerdem beinhaltet der FPGA Block-RAM (BRAM), welcher direkt mit den Logikblöcken verschaltet werden kann und somit sehr schnell ansprechbar und lesbar ist.

Das PYNQ-Z1 wurde speziell zur Entwicklung mit den PYNQ-Framework entwickelt. PYNQ ist ein Open-Source Framework von Xilinx, welches als Schnittstelle zwischen der eigentlichen Programmierung der Programmable Logic dient. Die Programmierung des FPGA erfolgt hier mittels Python, statt sonst üblicher, hardwarenahen Programmiersprachen. Außerdem stellt das Board bereits standardmäßig eine Design-Umgebung mittels Jupyter-Notebook bereit, sodass ohne viel Vorahnung direkt programmiert werden kann. Digilent bietet auf deren Website ([Dig21h]) fertige Images mit PYNQ zum Download an.

Des Weiteren bietet dieses Board weitreichende Anschlussmöglichkeiten wie Ethernet, USB, HDMI-Ausgang und insbesondere einem HDMI-Eingang. Es sind zudem zwei Schalter und vier Taster verbaut.

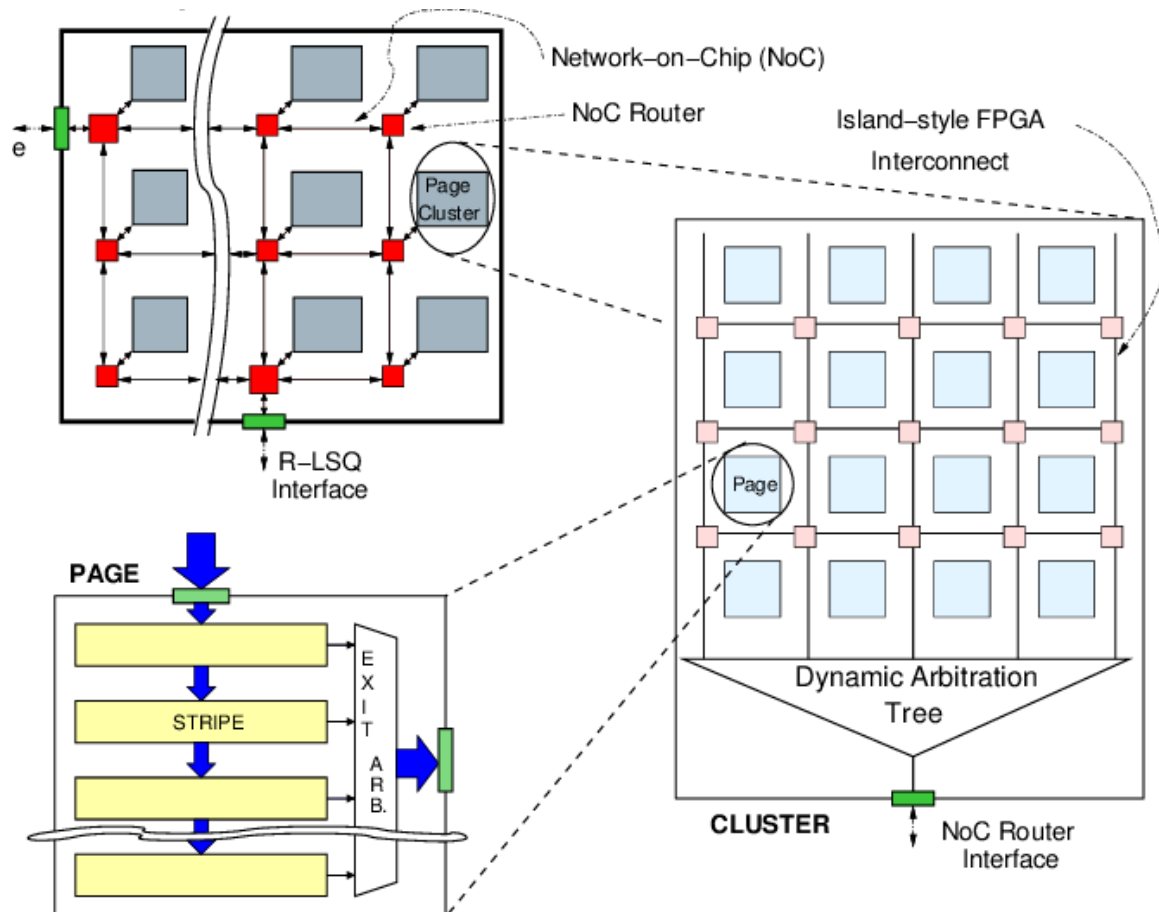


Abbildung 35: Schematischer Aufbau eines FPGA [Mis+06]

Die Versorgung ist mittels USB oder Netzteil möglich.

2.11.2 Fahrzeug

Geschrieben von Felix Müller

In diesem Projekt wurde ein im Vorgängerprojekt erweitertes Modellfahrzeug des Herstellers Elegoo verwendet [Ele]. Es besitzt vier Gleichstrommotoren, die über eine H-Brücke als Verstärker paarweise (links und rechts) angesprochen werden können. Ursprünglich wurde vom Hersteller ein Arduino UNO zur Generierung der Motorsteuerungssignale vorgesehen, welcher im Vorgängerprojekt durch das Digilent PYNQ-Z1 ersetzt wurde. Zusätzlich zur mitgelieferten Stromversorgung der Motoren über zwei Akkuzellen wurde eine Akkubank zum Betrieb der PYNQ-Platine angebracht. Vom Hersteller des Modellfahrzeuges werden noch weitere Sensoren, wie ein Abstandssensor, mitgeliefert, welche allerdings vorerst nicht angebracht wurden.

Als Bilddatenquelle wurde eine einfache USB-Webcam mit Full-HD-Auflösung starr am Fahrzeug montiert und über den USB-A-Anschluss an die CPU des SoC auf der PYNQ-Platine angeschlossen, sodass sie als einfaches Videogerät im verwendeten Linux-Betriebssystem adressiert werden kann.

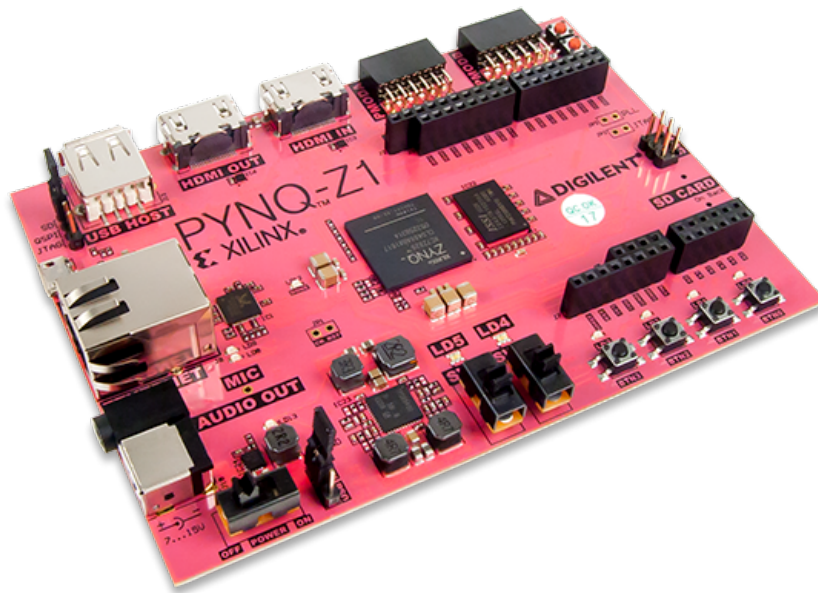


Abbildung 36: Das PYNQ-Z1 FPGA-Entwicklerboard [Dig21h]

2.12 Vorgängerprojekt

Geschrieben von Felix Müller

Im Vorgängerprojekt wurde zum Zwecke der Erforschung von KNN auf FPGAs eine Hardware- und Softwareplattform entwickelt, welche aus vier Teilsystemen besteht, die alle in diesem Projekt weiterverwendet wurden.

Das Teilsystem *Hardware (HW)* ist die Hardwareplattform, welche aus einem Modellfahrzeug, einer Kamera und einem Digilent PYNQ-Z1 mit Zynq-SoC besteht. Alle drei Komponenten wurden bereits in den vorherigen Abschnitten beschrieben.

Das Teilsystem *Programmable Logic (PL)* ist die Hardwareschaltung mit welcher der FPGA programmiert wird. Grundsätzlich sind in dieser zwei Komponenten enthalten: Ein PWM/GPIO-Block zur Ansteuerung des Motorcontrollers auf dem Fahrzeug und das KNN. Beide sind über den AXI4-Bus des SoC mit der CPU verbunden.

Das Teilsystem *Processing System (PS)* ist die Software auf der CPU des SoC, welche die Schaltung zur Steuerung der Motoren und das KNN auf dem FPGA anspricht und Debugginginformationen an eine Basisstation weitergibt. Die Software besteht grundsätzlich aus einer einfachen Hauptschleife, die die folgenden Tasks wiederholt ausführt. Als Erstes werden die Bilddaten der Kamera ausgelesen und für das neuronale Netz vorbereitet. Teil dieser Vorbereitung ist Kachelung des Bildes mithilfe eines Sliding Windows, da die Eingangsauflösung des KNN bedeutend kleiner ist, als die der Kamera. Danach werden die erzeugten Teilbilder mit dem KNN auf dem FPGA ausgewertet und die Einzelergebnisse zu einer Position im Bild weiterverarbeitet. Mithilfe dieser Position und einem PID-Regler werden die Steuersignale für die Motoren generiert, die zuletzt an die Motorsteuerung auf dem FPGA weitergegeben werden.

Das Teilsystem *Base Station (BS)* ist die Basisstation, welche nicht Teil des Fahrzeuges ist und auch nicht für den Betrieb benötigt wird. Stattdessen dient sie zu Debuggingzwecken auf einem entfernten Computer, welcher mit dem Fahrzeug über ein Netzwerk verbunden ist.

3 Ausgewählter Anwendungsfall im Projektgebiet

Geschrieben von Felix Müller

Im Vorgängerprojekt fehlte die Zeit, um ein eigenes Netz zu trainieren, weshalb vorerst die vortrainierte deutsche Kennzeichenerkennung verwendet wurde. In diesem Projekt galt es nun diesen Platzhalter zu ersetzen und eine realitätsnahe Anwendung zu finden. Zu diesem Zwecke wurde die Personenverfolgung in Kombination mit einer Gestenerkennung gewählt, wobei letztere genutzt wird, um die Verfolgung zu starten oder zu stoppen. Als Gesten sollten solche gewählt werden, die leicht mit klassischen Netzen zur Objekterkennung erkannt werden können. Weiterhin galt zu beachten, dass die Auflösung der Bilddaten durch die verwendete Hardware stark begrenzt ist, wodurch die Gesten auch bei sehr geringer Auflösung erkennbar sein müssen.

In den folgenden Abschnitten werden zuerst die nötigen Schritte zur Umsetzung des Anwendungsfalls beschrieben, wonach die verwendeten Git-Repositories der einzelnen Komponenten vorgestellt werden. Zuletzt folgt eine Abwägung, ob der SoC aus dem Vorgängerprojekt zum Zeitpunkt dieses Projektes noch aktuell war oder ersetzt werden musste.

3.1 Konzept zur Umsetzung der Projektziele

Geschrieben von Felix Müller

Für die gewählte Personenverfolgung mit Gestenerkennung zur Steuerung mussten folgende Aufgaben gelöst werden:

- Auswahl/Erstellung eines geeigneten Datensatzes für Gesten und Personen
- Erstellung eines passenden neuronalen Netzes
- Training des erstellten Netzes mit dem Datensatz
- Auswahl eines geeigneten FPGAs
- Auswahl eines Frameworks für neuronale Netze auf FPGAs
- Übersetzung des trainierten Netzes für das FPGA mit dem gewählten Framework
- Anpassung der vorhandenen Fahrzeugsteuerung für das Netz

3.2 Konzept für die Kollaboration der Projektmitglieder

Geschrieben von Felix Müller

Die Zusammenarbeit erfolgte über die Versionsverwaltung Git. Aus dem Vorgängerprojekt wurden drei Repositories übernommen:

- *vehiclepl*: Vivadopjekt für die Hardwareschaltung inkl. neuronalem Netz auf dem FPGA
- *vehicleps*: Software für die CPU auf dem Fahrzeug
- *vehiclebasestation*: Software für die Basisstation zur entfernten Überwachung des Fahrzeugzustandes

In diesem Projekt wurden zwei weitere Repositories hinzugefügt:

- *soc-nn-main_repo*: Gesammeltes Wissen der Projektmitglieder, Skripte zum Training des neuronalen Netzes, Jupyter Notebooks und einheitliche Entwicklungsumgebung in Docker
- *gesten-datensatz*: Rohdaten des Datensatzes und Skripte zur Variation der Bilddaten

3.3 Reevaluation der Boardauswahl

Geschrieben von Tobias Nießen

Zur Verifizierung der Hardwarewahl des PYNQ-Z1 aus Vorgängerprojekten (siehe Kapitel 2.11.1) wird im Folgenden eine Reevaluation dieser Auswahl durchgeführt, um eventuell besser geeignete Entwicklungsboards zu finden. Grund dieser Reevaluation ist, dass das von der Vorgängergruppe benutzte neuronale Netz nahezu den gesamten BRAM belegt, sodass komplexere Netze eventuell auf langsameren Speicher ausweichen müssen.

In dem vorgegebenen Anwendungsszenario „Raumfahrt“ gelten – wie in der Einleitung beschrieben – strengere Auswahlkriterien für die zu benutzende Hardware. Speziell relevant für unser Projekt ist das Verhalten der Hardware bei deren Aussetzung von hoher (Weltraum-)Strahlung. Die verschiedenen Strahlen-Klassen lassen sich in drei Gruppen einordnen:

Keine Strahlungs-Einstufung Für die meisten produzierten Chips ist eine besondere Berücksichtigung von höherer Strahlung nicht von Bedeutung. Bei der normalen Anwendung auf oder nahe der Erdoberfläche werden als aktuell modernste Fertigungstechnik 5 nm-Chips hergestellt [TSM21]. Die Hintergrund-Strahlung gewinnt bei immer kleiner werdenden Transistoren auch immer mehr von Bedeutung. Auf der Erde sind die Auswirkungen momentan hauptsächlich begrenzt auf Fehler in der Speicherung von Bits im RAM (Random Access Memory) [Zie96]. Die neuesten, schnellsten Generationen an Grafikspeicher, DRAM (Dynamic Random Access Memory), sind aufgrund ihres immer höher werden Datendurchsatzes und immer kleiner werdenden Transistoren, so anfällig für Fehler, dass hier standardmäßig bereits Fehlerkorrektur-Mechanismen verwendet werden [Mic20a]. Durch diese Mechanismen werden etwaige Bit-Fehler durch Strahlung ebenfalls korrigiert. Ein Aussetzen dieser modernen, kleinen Chips von starker Strahlung führt nicht nur zu Bit-Fehlern, sondern kann auch dazu führen, dass die kleinen Verbindungen der Transistoren irreparabel durchbrennen.

Strahlungstolerant Die Strahlungsmenge wird mit abnehmender Atmosphäre immer höher, da die schützende Lufthülle schrumpft. Flugzeuge erfahren das 40-fache an Strahlung als die Erdoberfläche. Die Internationale Raumstation (ISS) ist einem 250-fachen Niveau an Strahlung ausgesetzt [ESA19]. Zur Korrektur von Fehlern wird hier, unter anderem, strahlungstolerante Hardware verwendet. Diese Hardware hat Zusatz-Schaltungen verbaut, welche Fehler erkennen, und im Falle dessen den Chip zurücksetzt. Durch dieses Zurücksetzen sind Chips dieser Kategorie nicht für sicherheitskritische Anwendungen geeignet. Um das Durchschlagen einzelner Gates der Transistoren zu verhindern, werden diese nur mit bis zu 20 nm gefertigt, damit der Abstand innerhalb der Transistoren nicht zu klein wird. Diese Fertigung stellt zumindest sicher, dass die Hardware auch nach Aussetzung hoher Strahlung weiterhin betrieben werden kann, wenn auch nach einem Neustart.

Strahlungsfest Damit es in Chips gar nicht erst zu Fehlern durch Strahlung kommt, werden diese mit wesentlich größeren Fertigungstechniken von bis zu minimal 65 nm gefertigt. Außerdem finden besondere Verfahren Anwendung, wie z. B. Silicon-On-Insulator, um den Latch-Up-Effekt ganz zu verhindern. Der Latch-Up-Effekt ist das Durchschalten der Versorgungsspannung eines PN-Überganges durch Übergang in einen niederohmigen Zustand, welcher durch ein gemeinsames, leitendes Substrat ermöglicht wird. Durch Silicon-On-Insulator wird diese eine Schicht mittels einer isolierenden Zwischenschicht in zwei Schichten geteilt, sodass ein Durchschalten nicht möglich ist.

Der Mars hat keine Atmosphäre und bietet somit keinen Schutz vor Weltraumstrahlung [ESA19]. Der einzige strahlungsharte FPGA von Xilinx ist der Xirtex-5QV [Xil11], welcher in 65 nm gefertigt wird. In Vorgängerprojekten wurde aus Preisgründen bereits ein Board ausgewählt, welches möglichst nahe an die Rechenleistung und den Speicher dieses strahlungsharten FPGAs herankommt, ohne jedoch die eigentliche Eigenschaft der Strahlungshärte zu erfüllen: das Pynq-Z1 [Dig21h]. Wie in Tabelle 4 zu sehen, ist der FPGA des Pynq-Z1 in seiner Dimensionierung ähnlich dem Virtex-5QV. Der Pynq hat 65 % der Logic Cells des Virtex und etwa halb so viel BRAM. Da zu wenig BRAM dazu führen kann, dass Daten

auf einen langsameren Speicher zwischengespeichert werden müssen, könnte dieser einen Flaschenhals für neuronale Netze darstellen.

Das Board sollte mit der bereits genutzten Pynq-Bibliothek kompatibel sein. Dies sind alle auf Zynq basierenden Boards. In der weiteren Auswahl einschränkend ist außerdem der Preis der Hardware. Hierdurch wurde die Suche auf Boards für unter 500 € eingeschränkt. Unter diesen Kriterien wurden 14 Boards in Betracht gezogen. Allerdings haben fast alle dieselbe Einschränkung: Der verbaute BRAM ist kleiner, als der des Pynq-Z1. Die Boards, welche mehr BRAM haben, haben direkt signifikant mehr. Ebenso ist die Rechenleistung dieser nicht mehr vergleichbar mit unserem Ausgangskriterium. Aus diesen 14 passen sowohl preislich, als auch leistungstechnisch, am besten das, bereits vorhandene, Pynq-Z1, sowie ein weiteres Board: das Avnet Ultra96V2 UltraScale [Avn21b]. Dieses ist ebenfalls als Vergleich in Tabelle 4 dargestellt.

Das Ultra96V2 hat 50 % mehr BRAM als das Pynq-Z1. Ebenso hat dieses 17 % mehr Logic Cells, sowie 60 % mehr Flip-Flops. Des Weiteren sind hier zwei CPUs verbaut: Ein Quad-Core und ein Dual-Core. Falls sich somit im Laufe der Entwicklung herausstellen sollte, dass der BRAM des Pynq-Z1 einen Engpass darstellt, wäre eine Anschaffung hier eine Erwägung wert.

Da wir allerdings, nach momentanem Stand, die genauen Hardwareanforderungen unserer fertigen neuronalen Netze und sonstiger Software nicht genau abschätzen können, ist eine Entscheidung, neue Hardware anzuschaffen, verfrüht. Außerdem müssten die vorhandenen Hardware-Schnittstellen angepasst werden, da das Ultra96V2 sowohl keinen HDMI-Eingang, als auch weniger (aber wahrscheinlich ausreichend) I/O hat. Für Folgeprojekte könnte es jedoch immer interessanter werden, die Hardware für eine stärkere zu tauschen. Hier wäre ebenfalls möglich, eine Vorerkennung der Bild-Gebiete (Area of Interest) mittels eines zweiten neuronalen Netzes auf der CPU laufen zu lassen.

Im Folgenden evaluierte Boards, welche als nicht geeignet eingestuft wurden. Die Begründung ist jeweils beim Board mit angegeben.

- Digilent Nexys A7 [Dig21f]: langsamer und weniger BRAM als Pynq-Z1
- Digilent Arty S7 [Dig21c]: weniger BRAM als Pynq-Z1
- Digilent Nexys Video Artix-7 [Dig21g]: 2,5x schneller als Pynq-Z1
- Digilent Nexys 4 [Dig21e]: weniger BRAM als Pynq-Z1
- Digilent Basys 3 [Dig21d]: weniger BRAM als Pynq-Z1
- Digilent Arty A7-100T [Dig21b]: ungefähr gleich mit Pynq-Z1, aber weniger I/O
- Digilent ZedBoard [Dig21i]: Fokus auf Audio
- Digilent Anvyl [Dig21a]: stark veraltet
- Digilent Zybo Z7 [Dig21j]: Hardware und I/O identisch mit Pynq-Z1
- Intel Altera [Int21]: zu proprietär
- Avnet MicroZed [Avn21a]: nicht ausreichende I/O
- Numato Mimas A7 [Num21]: langsamer und weniger BRAM als Pynq-Z1

FPGA	(kein Board) Virtex - 5QV (65 nm)	Pynq-Z1 Artix-7 equivalent (28 nm)	Avnet Ultra96V2 UltraScale Zynq ZU3EG (20 nm)
Logic slices	20.480	13.300	n. A.
Logic cells	131.072	85.000	154.000
LUTs (/slice)	6	6	n. A.
LUTs	87.920	79.800	71.000
Flip-Flops (/slice)	4	8	n. A.
Flip-Flops	87.920	106.400	141.000
BRAM	1.312 KB	630 KB	950 KB
DSP slices	320	220	360
RAM	-	512 MB DDR3	2 GB DDR4
Application CPU	-	650MHz dual-core Cortex-A9	1,5 GHz 4-Core ARM-A53
Real-Time CPU	-	-	600 MHz 2-Core ARM-R5F
Controller (fast)	-	1G Ethernet, USB 2.0, SDIO	PCIe x4, USB3.0, WiFi
Controller (slow)	-	SPI, UART, CAN, I2C	GPIO
Misc	-	HDMI input	

Tabelle 4: Vergleich Virtex-5QV FPGA mit ähnlichsten zwei Boards

4 Datensatz für Gestenerkennung

Geschrieben von Philipp Altnickel

Dieses Kapitel beschäftigt sich mit der Findung bzw. Erstellung eines Bilddatensatzes. Dieser soll in der Folge dann zum Training eines eigenen neuronalen Netzes verwendet werden, damit dieses für die Erreichung des Projektzieles der Gestenerkennung genutzt werden kann.

4.1 Wahl des Datensatzes

Geschrieben von Philipp Altnickel

Um das in diesem Projekt gesteckte Ziel, eine realitätsnähere Anwendung umzusetzen zu erreichen, musste neuronales Netz eingesetzt werden, welches u.a. Personen- und Gestenerkennung ermöglicht. Für ein solch spezielles Szenario ist ein fertig vortrainiertes Netz nicht verfügbar. Aus diesem Grund musste ein passendes Netz selbst trainiert werden. Neben dem der Arbeit daran, wie ein solches Netz zu erstellen ist, war ein wesentliches Problem, passende Bilddaten in ausreichender Menge zu bekommen. Je mehr und je besser auf den Anwendungsfall passende Bilder es gibt, desto besser wird auch die Erkennung am Ende funktionieren.

D.h. der erste logische Schritt war, nach bestehenden Datensätzen zu suchen. Die Entscheidung für ein konkretes Anwendungsszenario wurde auch anhand der verfügbaren Bilder verfeinert und auch daran orientiert, wie die Bilder aussehen müssen um ein möglichst genaues Ergebnis bei der Erkennung zu erzielen. Es gibt in fast allen denkbaren Bereichen Datensätze, die auch oft schon fertig sortiert und kategorisiert sind. Verfügbare Bildreihen gibt es beispielsweise für die verschiedensten Objekte, wie Autos, Tiere oder andere Gegenstände. Die Möglichkeit der Verfolgung von Fahrzeugen hat sich als eher unbrauchbar herausgestellt, da es für das hier verwendete, eher spezielle Fahrzeug (siehe Abb. 1) keine vorhandenen Daten gibt, wären hier weitere Schritte, wie das Verändern des Fahrzeugs oder das Verwenden eigener Bilder notwendig gewesen. Als besser und umsetzbar stellte sich die Personenerkennung, bzw. -verfolgung heraus. Rein für diesen Fall sind u.a. einige Datensätze von Fußgängern verfügbar. Bei-

spielsweise von einem Universitätscampus [Uni07] oder aus dem Innenstadtbereich verschiedener Städte [Bra+19]. Um das Szenario in der Komplexität zu erweitern, sollte die Personenerkennung zusammen mit einfacher Gestenerkennung umgesetzt werden. In der Kombination ist es schwierig passende Daten zu finden. Dadurch, dass es aber aufgrund der Leistung des verwendeten Boards auch Begrenzungen bzgl. der Größe der Trainingsbilder gibt, hätten auch Bilder gefunden werden müssen, bei denen die Gesten auf sehr klein skalierten Aufnahmen gut zu erkennen sind. Es gab zwar z.B. auch einen Datensatz, der Personen bei bestimmten Aktionen erkennen könnte [Inf14], aber diese waren relativ zahlreich und schlecht auseinander haltbar. Aus diesem Grund werden in diesem Projekt eigene, selbst erstellte Bilder verwendet.

4.2 Wahl des Inhalts

Geschrieben von Toni Dragojevic, Stanislav Voytas, Mattia Uhlenbrock

Das autonome Fahrzeug soll, durch den ihm zugeführten Kamera-Feed einen Menschen erkennen, diesem folgen und eine begrenzte Anzahl an Gesten/Posen, die dieser Mensch ausführt/einnimmt, kategorisieren. Die zu kategorisierenden Posen müssen dabei auch bei geringen Auflösungen klar unterscheidbar sein. Es wurden folgende Posen festgelegt, welche durch das NN erkannt werden sollen:

- "START" - Beide Arme einer Person stehen horizontal von den Schultern ab. Die Unterarme sind um 90° vertikal nach oben angewinkelt.
- "STOP" - Beide Unterarme einer Person sind vor ihrer Brust überkreuzt.
- "NOPOSE" - Neutral, jede andere erdenkliche Position einer Person.

Es muss also ein Datensatz verwendet werden, der eine Vielzahl von Bildern von Personen; die zumindest die genannten drei Gesten zeigen, enthält und die Bilder in die entsprechenden drei Kategorien einordnet.

Zuletzt wurde eine Version des Datensatzes erstellt (V6), die eine weitere Klasse beinhaltet:

- "UNCLASSIFIED" - Zufällig gewählte Bilder, auf denen keine Person zu sehen ist.

Die Einbindung einer "none of the above"-Klasse (die UNCLASSIFIED darstellt) soll nach Zhang et al. [Zha16] einen regulierenden Effekt auf das Training neuronaler Netze u.A. zur Klassifikation von Bildern haben.

4.3 Erstellung der Bilder

Geschrieben von Toni Dragojevic, Stanislav Voytas, Mattia Uhlenbrock

Es wurden 70 „Start“, 70 „STOP“ und 160 „NOPOSE“ Bilder für den Gesten-Datensatz der ersten Version angefertigt, auf denen jeweils nur eine Person zu sehen ist und die entsprechende Pose einnimmt. Auf den Bildern sind diverse Hintergründe, drei unterschiedliche Personen aus abweichenden Winkeln, mit unterschiedlicher Kleidung und bei unterschiedlichen Lichtverhältnissen abgebildet. Dabei sind die Bilder etwa im Verhältnis 90/10 in Trainings- und Testdatensatz aufgeteilt. Dieses Verhältnis diente auch in den weiteren Versionen als Orientierung. Das Training des ersten Prototyp-CNN mit dem neuen Datensatz lieferte lediglich eine Genauigkeit von ~54%.

In der zweiten Version des Gesten-Datensatzes (V2) zeigen die Bilder minimalen Hintergrund/Kontext. Dabei sind die Bilder als Rechtecke unterschiedlicher Größe anhand der Originale ausgeschnitten. Außerdem wurde die Anzahl der Bilder in den drei Klassen angeglichen. Somit ist der Datensatz in Version zwei ausbalanciert. Die Anpassungen führten nicht zu Verbesserung der Genauigkeit des Netzes.

In einer weiteren Version des Datensatzes (V3) ist erneut der Kontext minimal. Die ausgeschnittenen Bilder liegen dieses Mal als Quadrate also in einem einheitlichen Seitenverhältnis vor.

Die vierte Version (V4) entspricht in der Aufbereitungsweise der Version V3 allerdings enthält diese zusätzliche Bilder für jede Klasse. Die zweite Variante des V4 Datensatzes (V4.1) nutzt ein Skript, welches neben der ursprünglichen Orientierung die Bilder auch in einer jeweils 90, 180 und 270 Grad gedrehten Ausführung erzeugt. Die Bilder in allen vier Rotationen werden darüber hinaus vom selben Skript auf x-Achse und y-Achse gespiegelt abgespeichert. Das Skript erstellt die Varianten für Trainings- und Testdatensatz.

In den vorherigen Versionen des Datensatzes sind Bilder enthalten, die in unterschiedlichen Kriterien und Formen gegen die formulierten Guidelines zum Erstellen der Bilder verstoßen. Die fünfte Version (V5) des Datensatzes enthält ausschließlich Bilder, die nach erneuter manueller Sichtung den Kriterien der erstellten Richtlinien entsprechen. So ist insbesondere die Konformität der Test- aber auch Trainingsbilder zu den Guidelines gewährleistet. Mit dem bereits im vorherigen Absatz beschriebenen Variationskript lässt sich aus dem Datensatz V5 der entsprechende Datensatz V5.1 erzeugen.

Die sechste Version des Datensatzes (V6) ist ein Versuch das Problem eines leeren Kamera-Bildes (ohne Person) im live-Betrieb des Systems zu lösen, und die von unserem neuronalen Netz erreichte Genauigkeit zu steigern. Das Vorgehen wurde von "Univerum Prescription" von Zhang et al. [Zha16] inspiriert. Ein Google Bilder crawler Skript wurde benutzt, um eine bestimmte Anzahl an Bildern einer bestimmten Google-Bilder-Suchanfrage herunterzuladen. Die Bilder bestehen aus Ergebnissen der Suchanfragen "Bremen Stadt" "Hemelingen" und "Bremen Park" und sollen zufällige Hintergründe ohne Personen darstellen.

Version	Anzahl Gesamt	Anzahl NOPOSE	Anzahl START	Anzahl STOP	Masßnahme
V1	299	160	70	69	-
V2	176	58	56	62	Balanciert
V3	176	58	56	62	Quadratisiert
V4	387	127	127	133	Weiterer Bilder
V4.1	3311	1547	1547	1607	Variabilitätsskript
V5	(368, 30)	(123, 10)	(120, 10)	(125, 10)	Konformität Guidelines
V5.1	(4.416, 360)	(1476, 120)	(1440, 120)	(1500, 120)	Variabilitätsskript
V6	507	123	120	125	120 Kein Mensch Klasse

Tabelle 5: Überblick über Versionen des Gestendatensatzes

4.4 Umwandlung in einen PyTorch Datensatz

Geschrieben von Ismail Sastim, Reena Wichmann, Jonas Philipp, Fannese Tchang

Datensätze können verschieden organisiert sein. Da wir nur Bilddaten auswerten, bietet PyTorch hierfür bereits eine fertige generische *Dataset*-Klasse Namens *ImageFolder* an. Sie sieht vor, dass die Bilder je nach Klasse in verschiedene Ordner unterteilt werden müssen. Die Verzeichnisnamen entsprechen den Klassennamen. Es wurde zusätzlich entschieden, die Testbilder von den Trainingsbildern zu trennen, damit immer mit denselben Testbildern die Genauigkeit gemessen werden kann für vergleichbare Ergebnisse. Die Verzeichnisstruktur hat danach in unserem Fall folgendermaßen auszusehen: [Con]

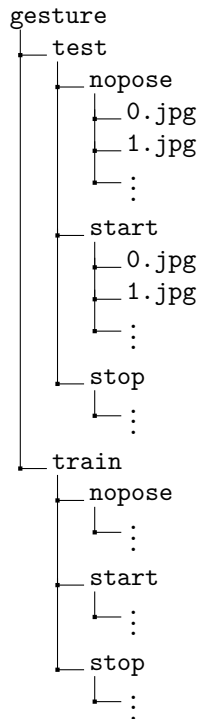


Abbildung 37: Verzeichnisstruktur

Falls es in Zukunft erforderlich sein sollte, exotischere Datensätze einzulesen, wie bestimmte Bildpunkte aus einer CSV-Datei statt ganze Bilder, kann die *Dataset*-Klasse geerbt, die Methoden `__init__()`, `__len__()` und `__getitem__()` überlagert und den Wünschen nach angepasst werden. [Chi] Für das Projekt reicht jedoch derzeit die *ImageFolder*-Klasse aus.

4.5 Wahl eines Modells (Variante 1)

Geschrieben von Ismail Sastim, Reena Wichmann, Jonas Philipp, Stanislav Voytas

Zur Festlegung auf ein Modell, mit dem weiter geforscht werden soll, wurden die Modelle *QuantWeightLeNet* [Papb], *LowPrecisionLeNet* [Papb], ein eigenes Modell und *CNV_1W1A* [Papa] getestet. Vorerst wurde mit dem CIFAR10 Datensatz getestet, da noch kein eigener Datensatz zur Verfügung stand. Im weiteren Verlauf konnte mit dem Gestendatensatz getestet werden. Die Modelle erzielten in den Tests mit den jeweiligen Testdaten eine Genauigkeit, die folgend festgehalten wird.

Quelle	Name des Modells	Anzahl Epochen	Batch Size	Datensatz	Genauigkeit (Testdaten)
[Papb]	QuantWeightLeNet	10	128	CIFAR10	≈ 64%
[Papb]	LowPrecisionLeNet	10	128	CIFAR10	≈ 42%
-	Eigenes Modell	10	128	Gesten Datensatz V2	≈ 73%
[Papa]	CNV_1W1A	1000	32	Gesten Datensatz V2	≈ 80%

Tabelle 6: Genauigkeitsresultate

Es ist zu erkennen, dass im Training mit dem Modell *CNV_1W1A* [Papa] mit den Daten des Gestendatensatzes die höchste Genauigkeit von 80% erzielt wurde.

Letztendlich wurde das *CNV_1W1A*-Modell gewählt, da das *FINN*-Repository ein Notebook zum Aufbereiten bereitstellt. Es optimiert und fügt die HLS Layer in das trainierte *CNV_1W1A*-Modell für die anschließende Hardware Synthese in Vivado ein [FINa]. Ferner ist es ein Binäres Neuronales Netz, welches nach Recherchen sehr geeignete Eigenschaften mitbringt. (siehe auch 2.4). Zudem lassen sich mit ausreichenden Epochen akzeptable Genauigkeiten erzielen.

Zu beachten war, dass die Docker Entwicklungsumgebung die Brevitas Version 0.2.0a0 [Papc] nutzt. Entsprechend existieren im Vergleich zu den aktuellen Versionen einige Funktionen und Klassen nicht oder sind anders benannt. Folglich sind einige Anpassungen nötig, falls Modelle wie *QuantWeightNet* oder *LowPrecisionLeNet* des *master* Branchs [Papb] genutzt werden sollen. Das Modell *CNV_1W1A* wurde aus der Version 0.2.0a0 [Papc] entnommen, damit keine Funktionen oder Klassen aus neueren Brevitas Versionen verwendet werden.

4.6 Wahl eines Modells (Variante 2)

Geschrieben von Fannse Tchang

Quelle	Name des Modells	Epochs	Batch Size	Datensatz	Genauigkeit (Testdaten)
-	Eigenes Modell	int=1	32	Gesten Datensatz V1	≈ 72%
[Papa]	CNV_1W1A	int=1	120	Gesten Datensatz V1	≈ 51%
[Papa]	CNV_1W2	int=5	32	Gesten Datensatz V3	38,09%
[Papa]	CNV_1W2A	int=5	100	Gesten Datensatz V3	33,33%

Tabelle 7: Genauigkeitsresultate

Zu beachten ist, dieses das trainieren im Docker Container anders verläuft als Lokal. Da die Brevitas Version 0.2.0a0 die Module *ExtendedInjector*, *ZeroZeroPoint* nicht unterstützt. Diese wurde entsprechend verändert, außerdem wurde beim Training mit einem *batchSize* von 500 einen Kernel *Tod*(Kernel died) festgestellt da diese mehr Speicherplatz benötigt .

5 Auswahl eines Frameworks für KNN auf FPGAs

Geschrieben von Felix Müller

Da bereits einige Forschungsarbeiten zur Implementation von KNN auf FPGAs existieren, wurde versucht eine solche zu finden, die genug Anpassbarkeit und Dokumentation besitzt, um sie für den gewählten Anwendungsfall zu erweitern. In diesem Kapitel folgt im ersten Abschnitt eine Analyse vorhandener Arbeiten und folgend eine Abwägung zweier potentieller Frameworks zur Erstellung von KNN mit Hinblick auf eine anschließende Hardwaresynthetisierung.

5.1 Vorhandene Frameworks zur Synthetisierung für FPGAs

Geschrieben von Felix Müller, Tobias Nießen

Zur Umsetzung neuronaler Netze auf FPGAs muss deren Topologie auf FPGA-Hardware übersetzt werden, sodass der FGPA mit diesen umgehen und diese ausführen kann. Für unsere ausgewählte Hardware hat Xilinx eine Entwicklungsumgebung basierend auf Python entwickelt. Mittels dieser Umgebung ist es möglich, diese Umsetzungs-Schritte zu beschreiben. Hierzu muss ein Framework entwickelt werden, welches diese Umsetzung tatsächlich durchführt.

Das *Pynq-Z1* Board und die dazugehörige Entwicklungsumgebung „Python Productivity for Zynq (*Pynq*)“ wurde 2016 veröffentlicht. Mitte 2017 wurde *Pynq* auf Version 2.0 erweitert. Mit dieser Erweiterung wurde darauffolgend viel geforscht. Es gibt viele wissenschaftliche Veröffentlichungen, welche das Board und die

Umgebung testen und vergleichen. Alle kommen zu einem ähnlichen Fazit; das Pynq ist gut und schnell. Die neuesten Veröffentlichungen, welche sich mit dem Pynq-Z1 beschäftigen, sind aus 2018. Etwa um diese Zeit ist ein UltraScale Zynq-Board veröffentlicht worden, welches Preislich nahezu gleich ist, aber sowohl einen schnelleren Prozessor, als auch FPGA beinhaltet. Wir nehmen an, dass die darauffolgende Forschung aus diesem Grunde mit diesem oder ähnlichen Boards weitergeführt wurde, und deswegen der Stand der Forschung bei dem Pynq-Z1 mittlerweile drei Jahre zurückliegt.

Nicht alle Autoren beschreiben ihr genaues Vorgehen, wodurch diese Ergebnisse durch uns zuerst selbst nachgestellt werden müssten. Dieses Nachstellen ist prinzipiell möglich, wenn das verwendete Framework und die genaue Implementierung beschrieben wird. Für das jetzige Projekt werden diese Veröffentlichungen aus zeitlichen Gründen nicht weiter betrachtet. Ebenfalls erwähnen einige Autoren zwar das verwendete Framework, geben aber keine Implementierung an. Diese Veröffentlichungen wurden von uns ebenfalls nicht weiter verfolgt. Oft wird zwar die Implementierung und Vorgehensweise genannt, aber nur sehr grob erläutert. Hierdurch ist die Vergleichbarkeit von verschiedenen Forschungsarbeiten nicht immer gegeben.

Im Folgenden eine Auswahl an Veröffentlichungen zu Frameworks, bei welchen immerhin die Vorgehensweise dargestellt ist:

- Comparative Study of Hardware Accelerated Convolution Neural Network on PYNQ Board [Sal+20]
- Convolutional Neural Network Quantisation for Accelerating Inference in Visual Embedded Systems [Leo18]
- Implementation Analysis of Convolutional Neural Networks on FPGAs [Iva19]
- Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs [Lu+17]
- An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs [Lu+19]
- CNN2Gate: An Implementation of Convolutional Neural Networks Inference on FPGAs with Automated Design Space Exploration [GS20]
- On how to efficiently implement Deep Learning algorithms on PYNQ platform [SSS18]
- Pynq-YOLO-Net: An Embedded Quantized Convolutional Neural Network for Face Mask Detection in COVID-19 Pandemic Era [Sai20]
- PYNQ Classification - Python on Zynq FPGA for Neural Networks [Blo+18]
- Face Mask Detection using YOLOv5 for COVID-19 [ICY21]
- A PYNQ-based Framework for Rapid CNN Prototyping [WDC18b]
- Vehicle License Plate Recognition System Based on Deep Learning Deployed to PYNQ [Hou+18]
- The implementation of a Deep Recurrent Neural Network Language Model on a Xilinx FPGA [HQ17]
- PYNQ FPGA Hardware implementation of LeNet-5-Based Traffic Sign Recognition Application [Mar+21]

Allgemein ist zu erkennen, dass die Forschung vor allem in Richtung Genauigkeit und somit in Richtung hoher Erkennungsraten geht. Echtzeitbedingungen, welche eine gewisse Rechenleistung voraussetzen, werden meist, zugunsten der Genauigkeit, vernachlässigt. Ebenfalls werden hauptsächlich RGB-Bilder klassifiziert. Drei volle Farb-Kanäle sind für unsere Anwendung aber nicht zwangsweise notwendig.

DAC-SDC Low Power Object Detection Challenge for UAV Applications [Xu+21] In der 55. Design Automation Conference wurde ein Wettbewerb mit dem Ziel der Objekterkennung durch unbemannte Luftfahrzeuge veranstaltet. Der Fokus lag hier auf der Genauigkeit der Netzwerke, ebenfalls gemessen und bewertet wurde auch der Stromverbrauch. Genauer wurden eine GPU (Nvidia Jetson Nano TX2) und

ein FPGA (Digilent Pynq-Z1) miteinander verglichen. Als Bewertungskriterien wurden die Genauigkeit der Implementierung, die Anzahl an Bildern pro Sekunde, sowie der Stromverbrauch mittels Intersection over Union (IoU) festgelegt. Der Fokus der Bewertung lag auf der Genauigkeit. Der Wettbewerb kommt zu dem Ergebnis, dass die GPU-Lösungen insgesamt schneller sind. Die Bildrate der schnellsten Lösung liegt bei ca. 59 FPS. Allerdings sind die FPGA-Lösungen nicht ungenauer, sondern ausschließlich langsamer. Die Ergebnisse der besten drei Gruppen sind in Tabelle 8 zu sehen. Während die Bildrate langsamer ist, beträgt der Stromverbrauch der FPGA-Lösungen ebenfalls nur ca. 30 % des Verbrauchs der GPU-Lösungen. In dieser Tabelle sieht man zudem den Zusammenhang zwischen Genauigkeit und Durchsatz: je genauer ein Netzwerk, desto geringer ist die Bildrate. Die von den Teilnehmern entworfenen Architekturen sind in Abbildung 38 dargestellt. Hier ist zu erkennen, dass alle drei Gruppen, zumindest in Teilen, in ihrer Architektur Convolutional Neural Networks verwenden. Abschließend wurden aber auch hier die genauen Implementierungen, der Code, nicht veröffentlicht.

Gruppe	Netzwerk	FPS	IoU Score
TGIIF	Single Shot MultiBox Detector (SSD)	11,96	0,6238
SystemETZH	SqueezeNet	25,97	0,4919
iSmart2	MobileNet	7,35	0,5733

Tabelle 8: Top 3 FPGA Teilnehmer [Xu+21]

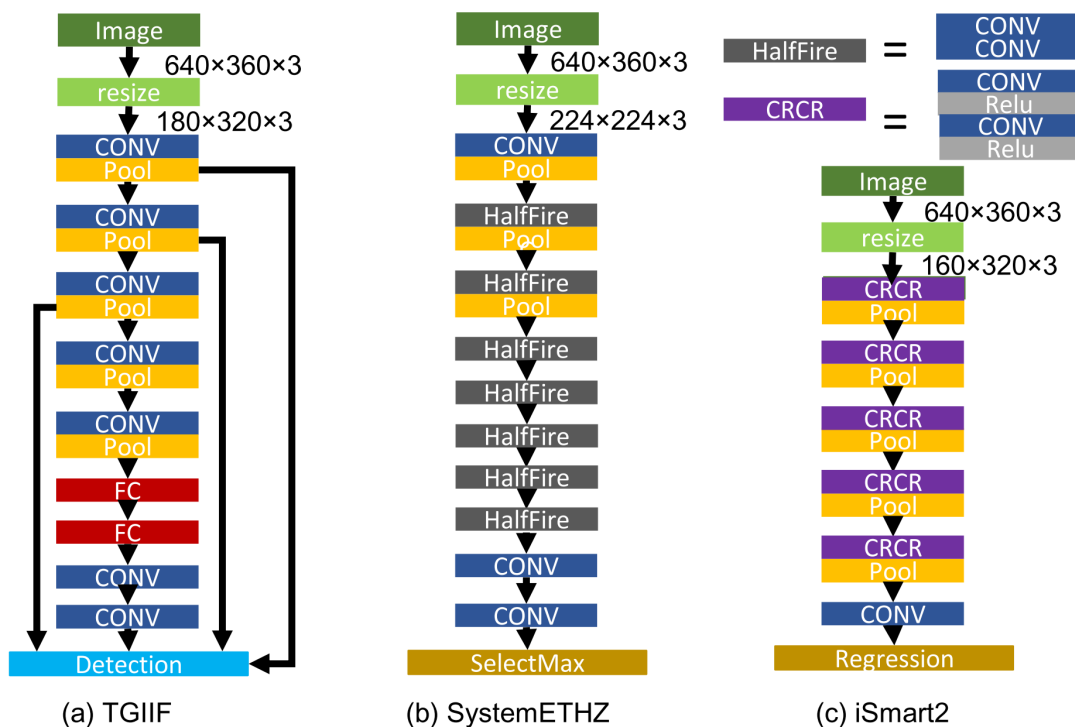


Abbildung 38: Top 3 FPGA Frameworks [Xu+21]

Generell muss das, für unsere Anwendung, gesuchte Framework zur Bildklassifizierung geeignet sein. Im Rahmen der Suche nach passenden und mittels Pynq anwendbaren Frameworks ergeben sich drei vielversprechende: FINN, DNNWeaver und PYNQ Classification. Eine genauere Beschreibung dieser befindet sich in den folgenden Paragraphen. Allgemein benutzen alle Frameworks den FPGA nur zur Ausführung der Neuronalen Netze, nicht aber zum Training.

FINN / BNN-PYNQ [Blo+18] Das FINN Framework wurde in dem Vorgängerprojekt bereits verwendet. Ein bereits fertiges Neuronales Netz läuft bereits hiermit auf dem Pynq-Z1. FINN verarbeitet 32x32 Pixel große Bilder mit bis zu 12.000 Bildern pro Sekunde. Das momentan implementierte Netz verbraucht ca. 80 % des Block RAMs im FPGA. Generell sind mit FINN sowohl Fully-Connected (LFC) Netze, sowie Convolutional Neural Networks (CNN) verarbeitbar. Das momentan laufende Netz ist ein Binäres Neuronales Netz (BNN), welches eine Sonderform von CNN darstellt. Für den Aufbau eigener Netze wird Xilinx Vivado HLS genutzt, welches zuletzt in Vitis 2020.1 enthalten ist. Hier kommt C/C++ zum Einsatz.

DNNWeaver [Sha+16] Ebenfalls für das Pynq-Z1 entworfen wurde DNNWeaver. Dieses Framework wird von einer Einzelperson der University of California betreut. Hierzu sind einige Demonstrationsvideos verfügbar. In diesen ist zu sehen, dass ein CNN hiermit auf dem Pynq-Z1 läuft. Der Hardwareverbrauch in diesem Beispiel liegt bei 94-100 % des Block RAM. Laut Entwickler sind weitere Netzwerk-Arten möglich und teilweise bereits implementiert. Genauer wird hierauf nicht eingegangen. Die Dokumentation zu DNNWeaver ist generell schlecht. Bis auf die Performance in dem Beispiel-Video wird nicht genauer auf den Hardwareverbrauch oder die möglichen Bild-Eingangsgrößen eingegangen. Besonders ist hierbei allerdings, dass nicht Vivado HLS genutzt wird.

PYNQ Classification [WDC18a] Das PYNQ Classification Framework ist ein Framework speziell für die Netzwerkstrukturen von Theano und Caffe. Genau wie mit FINN können hier Eingangsbilder von 32x32 Pixeln verarbeitet werden. Allerdings ist dieses Framework mit bis zu 500 Bildern pro Sekunde signifikant langsamer. Die verarbeitbaren Netze sind hier CNNs. Eine Implementierung von LeNet-5 benötigt hier ca. 99 % des Block RAM, Caffe „quick“ benötigt ca. 73 %. Da BNN eine Sonderform von CNN darstellen, wäre hiermit eventuell ebenfalls eine Implementierung dieser möglich. Etwaige Optimierungsschritte müssten allerdings noch implementiert werden.

Die Qualität der Dokumentation unterscheidet sich zwischen diesen enorm. PYNQ-Classification stammt aus einer Abschlussarbeit einer Person und wird, soweit erkennbar, nicht mehr aktiv entwickelt. „DNN-Weaver“ wird ebenfalls nur noch von einer einzelnen Person betreut. „FINN“ hingegen wird von der Norwegian University of Science and Technology, der University of Sydney und Xilinx selber entwickelt. Die Dokumentation hierzu ist mit Abstand am besten. Aus diesen Gründen haben wir uns für FINN entschieden, da, unserer Meinung nach, hierauf am besten aufgebaut werden kann.

Nicht anwendbar Im Folgenden eine Liste mit Frameworks, welche generell auf dem Pynq laufen, aber für nicht anwendbar befunden wurden. Eine kurze Begründung hierfür ist in Klammern angegeben.

- TinyYOLO-BNN [moh21] (Entwickelt für einen größeren FPGA → Zynq-7045)
- Pynq-YOLO-Net [Sai20] (Quellcode unveröffentlicht)
- LSTM-PYNQ [Ryb21] (Komplett anderer Anwendungsbereich: OCR → Schrifterkennung)
- DEEPhi [Sim18] (Aufkauf durch Xilinx → nur noch größere FPGAs unterstützt, teils kostenpflichtig)
- Xilinx Vitis AI [Xil21] (Nur für größere FPGAs)

5.2 Brevitas als Modellframework

Geschrieben von Stanislav Voytas, Reena Wichmann, Mattia Uhlenbrock

Für die Einarbeitung in die Implementierung und das Trainieren von Neuronalen Netzen erwies sich die Verwendung von TensorFlow als nützlich. Durch die gute und breite Dokumentation des Frameworks lassen sich die Konzepte für die generelle Implementierung von neuronalen Netzen und insbesondere von CNNs nachvollziehen. Aus diesem Grund wurden vorerst CNNs mit TensorFlow erstellt.

Für die Erstellung der Netzmodelle für das FINN-Framework, wird ausdrücklich Brevitas unterstützt.[Xil]

Das aus Brevitas exportierte ONNX-Modell dient dabei als Grundlage für die weitere Verarbeitung durch den FINN-Compiler.

Bei der Gegenüberstellung von TensorFlow und Pytorch mit Brevitas wurde jeweils das gleiche CNN im Jupyter Notebook umgesetzt und mit dem Datensatz CIFAR10 trainiert und getestet. Das Training erfolgte mit zehn Epochen. Durch gleiche Bedingungen lassen sich die Ergebnisse vergleichen. Es erwies sich bei der Validierung mit den CIFAR10 Testdaten bei TensorFlow eine Genauigkeit von 71,07% und bei Pytorch mit Brevitas von 70,08%. Die Genauigkeit ist also sehr ähnlich und wurde somit nicht als entscheidender Faktor festgelegt. Die Entscheidung beruhte hingegen darauf, dass ein von Brevitas im ONNX-Format exportiertes Modell garantiert vom bereits festgelegtes FINN Framework genutzt werden kann. [Xil] Hiermit ist sichergestellt, dass das Netz auf das FPGA übertragen werden kann. Ob ein ONNX-Modell mit einem quantifizierten Netz in TensorFlow mit dem FINN-Compiler genutzt werden kann war nicht sicher und hätte Zeit zur Evaluierung in Anspruch genommen.

6 Evaluation einer Umsetzung von Region Proposals oder Single Shot Detektoren auf FPGAs mit Brevitas

Geschrieben von Lukas Schimanski, Tuncer Catalkaya

Auf die Unterschiede zwischen Region Proposal Networks (RPN) und Single Shot Detektoren (SSD) wurde bereits im Grundlagenkapitel 2.3 eingegangen. Als Zusammenfassung lässt sich folgende Generalisierung treffen: RPN's besitzen eine höhere Genauigkeit kleinere Objekte in Bildern zu identifizieren, während SSD's eine geringere Inferenzzeit aufweisen. Um die angestrebte Gestenerkennung zu ermöglichen musste evaluiert werden, ob mit Brevitas ein SSD oder RPN umgesetzt werden kann oder ob die aus dem Vorgängerprojekt verwendete Methode des Tilings ([Mül+21a] Kapitel 6.4) wiederverwendet wird.

Ein Vorteil des Einsatzes eines RPN's oder SSD's wäre die höhere Genauigkeit die mit solchen Netzen erzielt werden kann. Ein Problem der Tilingimplementierung ist die Konfiguration der Tilegenierung. Hierbei muss immer der Größe der einzelnen Tiles und dem Stride (Überlappung eines Tiles mit dem nächsten) abgewogen werden, da mit vielen Tiles zwar theoretisch genauere Informationen bei der Klassifizierung extrahiert werden können, jedoch mehr Tiles auch mehr Leistung benötigen, wodurch die Anzahl an verarbeiteten Frames stark sinken kann. Zusätzlich ist gerade bei der Erkennung von Gesten der Abstand zum Fahrzeug ausschlaggebend für die Anzahl und Größe der benötigten Tiles, da bei einer geringen Distanz wenige große Tiles ausreichen können um die Geste zu klassifizieren, während bei einem größeren Abstand mehr Tiles und größere Überlappungen notwendig sind. Ein solches Konfigurationsproblem besteht bei einem RPN bzw. SSD nicht, da hier das gesamte Bild in das Netz gegeben und dann ausgewertet werden kann.

Hierzu wurden zunächst zwei frei verfügbare Implementierungen eines SSD's und eines RPN's herausgesucht, um die Komplexität des Codes einschätzen zu können ([Joc] und [Zha+21] (Kapitel 13.7)). Zu den "Zusatzaufwänden", die im Code festgestellt wurde gehört unter anderem die eigenständige Berechnung der IoU auf Basis einer vorhandenen Ground Truth Box und der vorhergesagten Bounding Box des Netzwerkes. Diese Information musste später auch wieder in die Backpropagation des Netzes einfließen, da sonst die Lokalisierung von Objekten nicht trainiert werden könnte.

Es wurde hierbei versucht, die vorhandenen Layer des Netzes in quantisierte Brevitas Layer umzuwandeln. Hierbei stellte sich allerdings heraus, dass sich teilweise Layer nicht 1:1 konvertieren lassen und manche Parameter der ursprünglichen Layer nicht umgesetzt werden konnten. Zusätzlich hat es die mangelnde Doku für Brevitas erschwert die gewünschten Layer entsprechend umzusetzen, wodurch teilweise viel Zeit investiert wurde um triviale Parameter zu setzen.

Ebenso konnte der Zeitaufwand für eine komplette Implementierung eines SSD's bzw. RPN's durch die beschriebenen Probleme innerhalb der Projektzeit nicht abgeschätzt werden, da zur Umsetzung des Netzes auch das Labeln der Trainingsdaten einbezogen werden muss. Hierzu zählt nicht nur das Klassifizieren der Bilder sondern auch das Hinzufügen der Bounding Boxes, damit später während des Trainings die IoU berechnet werden kann. Aus diesen Gründen wurde die Umsetzung eines SSD bzw. RPN in Brevitas für dieses Projekt verzichtet und versucht ein hinreichendes Ergebnis mittels Tiling zu erreichen.

Für ein Nachfolgeprojekt sollte hierbei **frühzeitig** festgelegt werden, mit welchen Parametern das spätere Netz arbeiten muss. Hierzu zählen unter anderem die Anzahl und Qualität der Trainingsdaten. Wie weit eine Person von dem Fahrzeug entfernt ist oder ob die Distanz fest oder variabel sein soll. Als Grundlage könnte folgende Webseite herangezogen werden: [Zha+21]. Die Webseite ist ein interaktives deep learning Buch mit Code. Die Beispiele wurden unter anderem in PyTorch umgesetzt. Hierbei sind besonders die Kapitel 13.7 und 13.8 interessant, bei diesen Kapiteln geht es um SSDs und R-CNNs. Die PyTorch Beispiele müssten in Brevitas umgeschrieben und optimiert werden, da eine volle Umsetzung eines SSDs oder R-CNNs oder auch nur RPNs wahrscheinlich nicht auf dem im Projekt verwendeten FPGA-Board (siehe 2.11.1) laufen würde.

7 Training in der Cloud

Geschrieben von Stanislav Voytas

Das Training auf einer herkömmlichen CPU kann bei einer komplexen Netztopologie sehr viel Zeit in Anspruch nehmen. Mit einer GPU kann dieser Vorgang wesentlich beschleunigt werden. Hierbei stößt man in einer Gruppenarbeit zwangsweise auf das Problem, das nicht jeder Teilnehmer über eine leistungsstarke GPU verfügt.

Um jedem Teilnehmer zu ermöglichen eine GPU in einer einheitlichen Entwicklungsumgebung zu verwenden ohne selbst eine zu besitzen wurde beschlossen das Training in die Cloud zu verschieben, wo Hardware Ressourcen kurzzeitig genutzt werden können.

7.1 Auswahl einer Cloud Plattform

Zur Auswahl standen zunächst die Cloud Plattformen Alibaba Cloud, Amazon Web Services, Google Colab und Microsoft Azure. Für die Erstellung eines Accounts in der Alibaba Cloud und in Amazon Web Services muss zwangsweise eine Kreditkarte hinterlegt werden. Da dies der Idee einer allgemein zugänglichen Arbeitsumgebung widerspricht sind diese Optionen ausgeschlossen. In die engere Auswahl kamen Google Colab und Microsoft Azure, aufgrund der Einfachheit der Einrichtung und Bedienung. Google Colab bietet 2 Varianten zur Nutzung der virtuellen Ressourcen an:

- kostenlose Variante
- Google Colab Pro für 9,25€/Monat

In der kostenlosen Variante wird für das Machine Learning eine NVIDIA Tesla K80 verwendet. In Google Colab Pro kann eine T4- oder P100-GPU verwendet werden.[Goo]

In der Azure Cloud hingegen, kann man als Student einen kostenlosen Account mit jährlich 100€ Guthaben erstellen. Dieses kann dann beliebig für virtuelle Ressourcen ausgegeben werden. Allerdings wird hierbei jede Ressource vom Guthaben abgerechnet. Es gelten dennoch bestimmte Einschränkungen für kostenlose Accounts, so kann man z.B. auch hier nur mit der NVIDIA Tesla K80 trainieren, welche pro laufende Stunde mit 0,97USD/Stunde vom Guthaben abgerechnet wird. Es besteht jedoch die Möglichkeit auf ein kostenpflichtiges Modell welches minutengenau die verwendeten Ressourcen nach Verbrauch abrechnet umzustellen.

Dadurch kann man so genannte Quotas beim Microsoft Support beantragen, die es ermöglichen leistungsstärkere VM Konfigurationen mit besseren GPUs wie z.B. NVIDIA Tesla V100 zu verwenden. Bei Quotas handelt es sich um eine Beschränkung der Anzahl der verwendbaren vCPUs.

Um die Option der stärkeren Hardware gegen Abrechnung nach Verwendung offen zu halten, wurde als Cloud Plattform wurde Microsoft Azure gewählt.

7.2 Microsoft Azure Cloud

In der Azure Cloud sind für unsere Zwecke zwei Cloud-Ressourcen besonders wichtig:

- Speicherkonto
- Machine Learning Ressource

Das Speicherkonto ist zu verstehen als ein Speicher in dem alle für das Training relevanten Dateien abgelegt werden. In unserem Fall liegt hier der Datensatz, das Modell und die entsprechenden Jupyter Notebooks. Die Daten lassen sich über git direkt aus dem Repository klonen.

Die Machine Learning Ressource ist zu verstehen als eigener Arbeitsbereich (auch Machine Learning Studio genannt) innerhalb der Cloud in dem alle für das Machine Learning Projekt relevanten Ressourcen und

Tools verwaltet werden. In diesem Arbeitsbereich wird z.B. die Virtuelle Maschine für das Training (auch Computing-Instanz genannt) erstellt und verwaltet. Die Computing-Instanz ist eine Virtuelle Maschine, auf der der Code ausgeführt wird. Von ihrer Leistung hängt die Geschwindigkeit des Trainings ab. Von den vielen Tools die das Machine Learning Studio zur Verfügung stellt ist besonders Jupyter Notebook hervorzuheben. Das Jupyter Notebook hat automatisch Zugriff auf die Dateien die im zuvor erwähnten Speicherkonto liegen, folglich also auf alle Dateien die aus dem Repository geklont werden.

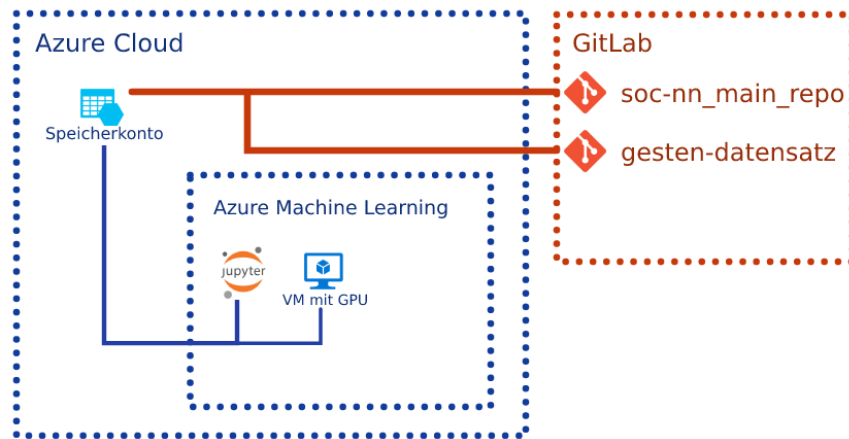


Abbildung 39: Visualisierung der Cloud-Architektur der Ressourcen

Durch diese neue Architektur ist der Dateispeicher nicht mehr an die Computing-Ressource gebunden. Der selbe Code kann also z.B. ohne weiteres auch auf anderen VMs ausgeführt werden.

Durch die Nutzung der Azure Cloud haben sich für unser Projekt folgende Vorteile ergeben:

- Einheitliche Arbeitsumgebung
- Verwendung von Jupyter-Notebook möglich
- Einheitliche Hardware
- Hardware ist schnell austauschbar und skalierbar
- Arbeitsumgebung kann exportiert und importiert werden

Außerdem bietet die Microsoft Azure Cloud Tools zur Erstellung von eigenen Datensätzen an. Die Datensätze können dann innerhalb der Cloud getrennt vom Codespeicher verwaltet und versioniert werden, was auch den Datensatz vom Codespeicher entkoppelt. Dies wurde im Rahmen von diesem Projekt nicht umgesetzt, kann in Zukunft aber eine bessere Lösung für die Versionierung des Datensatzes sein.

7.3 Cloud-Konfiguration und Kostenübersicht

Die Verwendung der Arbeitsumgebung in Machine Learning Studio ist kostenlos, laufende Kosten werden ausschließlich von der Computing-Instanz und vom verbrauchten Speicher im Speicherkonto verbraucht.

Bei der Computing-Instanz wird nach vollen Minuten abgerechnet, die die Instanz aktiv ist. Ist die Instanz ausgeschaltet, verursacht sie keine Kosten.

Beim Speicherkonto ergibt sich der Preis in Abhängigkeit vom verbrauchten Speicher und der Anzahl der Zugriffe. Dieser lässt sich daher individuell im Azure Preisrechner berechnen [Mic].

In der aktuellen Cloud-Konfiguration wird als VM eine *Standard_NC6* [Mic20b] verwendet, welche 0,97US-

D/Stunde verbraucht. Da die benötigte Speicherkapazität insbesondere durch den Datensatz schwanken kann, sollte hier der Azure Preisrechner [Mic] verwendet werden. Aktuell reicht eine Kapazität von 500MB aus, wodurch monatliche Kosten von 0,01€ entstehen. Hinzu kommen 0,50€ für je 10.000 Schreibvorgänge und 0,04€ für je 10.000 Lesevorgänge.

7.4 Vergleich der Trainingszeit

Um zu prüfen ob die Cloud nennenswerte Vorteile im Bezug auf die Trainingszeit bringt, wurde das selbe Brevitas-Modell in 1000 Epochen in einer Docker-Umgebung auf einem lokalen Computer und in der Cloud trainiert. Auf dem Computer wurde mit einer AMD Ryzen 5 1600 CPU trainiert, in der Cloud mit einer NVIDIA Tesla K 80 GPU trainiert. Das Training auf dem Computer dauerte 3946,9 Sekunden, das in der Cloud 3514,705.

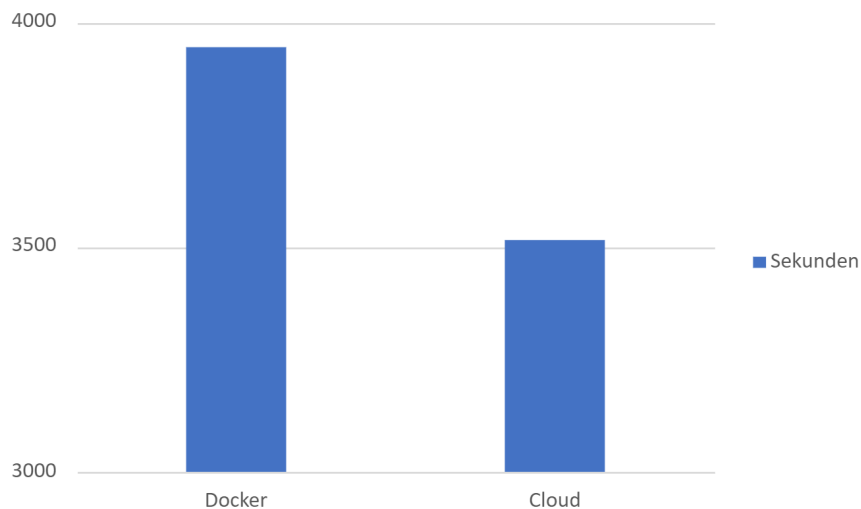


Abbildung 40: Vergleich der Trainings-Dauer Cloud vs. Docker

Hierbei wurde in der Cloud beobachtet, dass nach dem Start des Trainings der Speicherverbrauch der GPU zwar gestiegen ist, die GPU Auslastung jedoch bei 0 blieb. Das selbe Verhalten wurde beobachtet, wenn Brevitas Modelle auf lokalen Computern mit einer GPU trainiert wurden. Daraus kann man schließen, dass das Modell zwar auf das richtige Recheneinheit geladen wird, die Ausführung aber trotzdem immer auf der CPU stattfindet. Dies würde erklären warum die Trainingszeit durch die Verwendung der GPU in der Cloud sich nicht sonderlich gebessert hat. Die Fehlerursache konnte bisher nicht ermittelt werden, sie ist jedoch im Brevitas Code zu vermuten, da das Problem beim Training herkömmlicher PyTorch Modelle nicht auftritt.

Obwohl die Cloud auch ein Training auf VMs mit mehreren GPUs ermöglicht, wurde in diesem Projekt davon nicht Gebrauch gemacht, da zum aktuellen Zeitpunkt kein Bedarf besteht die Prozesse im Trainings-Vorgang zu parallelisieren.

Sollte der Trainings-Vorgang in Zukunft jedoch komplexer werden und die parallele Ausführung von Prozessen erfordern so würde sich diese Option mit Sicherheit auf die Trainingszeit auswirken.

8 Übersetzung der Netztopologien für das FPGA

Geschrieben von Ismail Sastim

Das Trainieren eines Modells in PyTorch mit den quantisierten Brevitas Layern (siehe 2.6) ist nicht ausreichend. Das Exportierte Model im ONNX-Format (siehe 2.7) muss vorerst einige wie in Abbildung 41 dargestellte Übersetzungsschritte durchlaufen, bevor es auf das FPGA Board installiert werden kann.

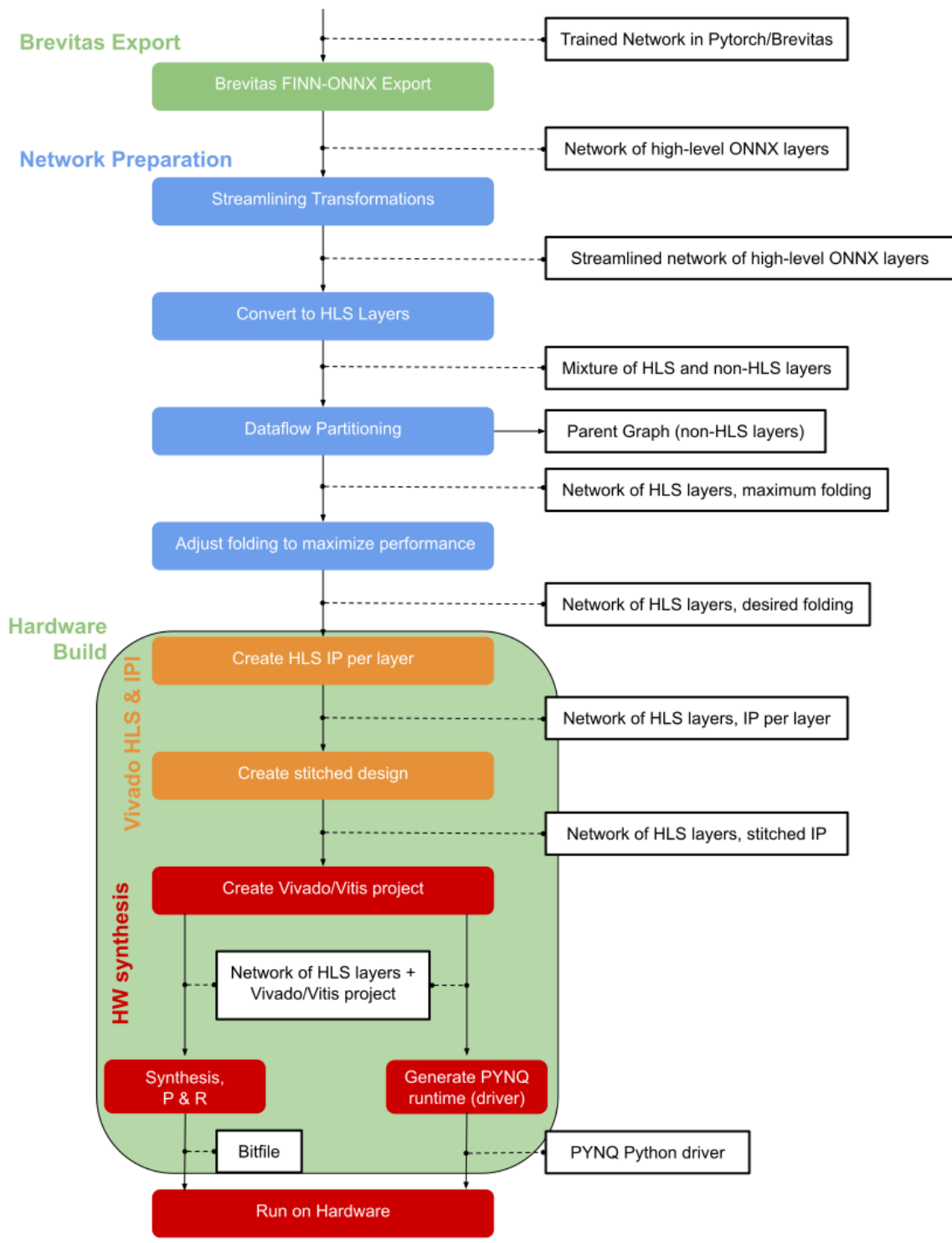


Abbildung 41: FINN End-to-End Flow [Xi]

Im Folgenden sollen die einzelnen Schritte kurz erläutert werden [Xil; FINa]:

Tidy-Up Transformations Dies ist ein Schritt, der in Abbildung 41 nicht explizit aufgezeichnet ist. Er wird in unserem Projekt nach dem Import der ONNX-Datei und nach dem Pre- und Postprocessing durchgeführt. Es werden keine allzu wichtigen Transformationen durchgeführt. Beispielsweise werden Datentypen und Dimensionen gecheckt, Konstanten eingefügt oder jedem Knoten im Model eindeutige Namen vergeben. Die hierfür verwendeten Transformationen sind unter [FINb] dokumentiert.

Adding Pre- and Postprocessing Dies ist wieder ein in Abbildung 41 unaufgezeichneter Schritt. Es ist weder in [Xil] noch in [FINa], woraus es übernommen wurde, dokumentiert, was umgesetzt wird.

Streamlining Transformations Ziel des Streamlining ist die Eliminierung der Fließkommazahlen durch Beispielsweise quantisierte Operatoren und Thresholds. Ferner wird das Netz weiter simplifiziert durch vereinfachte Operatoren wie dem XNOR Popcount (siehe 2.4) Das FINN Framework stellt hierzu verschiedene Transformationsfunktionen bereit [FINb].

Convert to HLS Layers Bestimmte Layer werden hier in ihre analogen HLS Layer umgewandelt. Aus diesen kann nämlich überhaupt die Hardwarebeschreibung erzeugt werden. Am Ende besteht das Model aus einer Mischung von HLS und standard Layern.

Dataflow Partitioning Das Model wird in zweigeteilt zwischen HLS- und Standardlayern. Nur die HLS Layer werden weiterverarbeitet, während die anderen im Datenfluss nicht mehr weitergereicht werden.

Folding Durch Anpassung der PE (Parallelisierung der Eingänge) und SIMD (Parallelisierung der Ausgänge) Parameter von definierten Layern kann die Performance nochmals optimiert werden. Dabei sorgt eine Steigerung der Parameter für höhere Performance in Austausch für mehr Ressourcenverbrauch. [PKK21]

Vivado HLS & IPI Die IPs für jeden Layer werden erzeugt und mit einander vernetzt.

HW synthesis Die erzeugten IPs können in einem Vivado Projekt eingepflegt und anschließend in ein Bitstream umgewandelt werden.

9 Anpassung der Fahrzeugsteuerung des Vorgängerprojektes

Geschrieben von Philipp Altnickel

Am Code der Fahrzeugsteuerung des Vorgängerprojektes wurden zahlreiche Änderungen vorgenommen. Dies umfasste zum einen Änderungen hinsichtlich des in diesem Projekt gewählten Anwendungsfalls. Zum anderen wurden aber auch diverse allgemeine Verbesserungen, Umstrukturierungen und Dokumentationen vorgenommen, um den Code insgesamt besser und übersichtlicher zu machen und auch für zukünftige Erweiterungen vorzubereiten.

9.1 Anbindung des neuronalen Netzes auf dem FPGA

Geschrieben von Philipp Altnickel

Zu einer der ersten und wichtigsten Anpassungen zählt die Einbindung des mittels FINN Framework trainierten Neuronalen Netzes. Das zuvor genutzte BNN-Pynq Netz nutzte eine ganz andere Schnittstelle und auch die Ein- und Ausgabedaten mussten ganz anders verarbeitet werden. D.h. der folgende Abschnitt beschreibt wie die benötigten Schnittstellen gefunden wurden und welche Änderungen am Code vollzogen wurden, um die vorherige Erkennung eines Verkehrszeichens vollständig zu ersetzen.

9.1.1 Probleme und Vorgehen

Geschrieben von Philipp Altnickel

Um ein per FINN Framework kompiliertes Neuronales Netz in das Fahrzeug zu integrieren, ist es natürlich essentiell, die genauen Schnittstellen und die Funktionsweise der Kommunikation zu kennen. Das große Problem dabei ist, dass es kaum offizielle Dokumentation zu diesem Thema gibt. D.h. bevor das Processing System auf die nun benötigte Schnittstelle angepasst werden konnte, musste erst einmal ein Weg gefunden werden, an die notwendigen Informationen heran zu kommen. Dafür waren zwei wesentliche Schritte notwendig. Zum einen wurde das „FINN End2End CNV“ Beispiel herangezogen, um einen Ansatz zu bekommen, die die Framework Entwickler diese Verbindung realisieren. Zum anderen ist es essentiell, die Funktionsweise des zur Kommunikation verwendeten AXI-Busses zu kennen, um anschließend eine eigene Kommunikation umsetzen zu können. D.h. das Vorgehen sah so aus, dass zunächst das Beispiel kompiliert und das dabei erstellte Vivado Projekt untersucht werden musste. Hier konnten die verwendeten IP-Blöcke, sowie die Verschaltung mittels AXI-Bus ermittelt werden. Außerdem sind hier die Adressen auf denen kommuniziert wird erkennbar.

9.1.2 Reverse Engineering

Geschrieben von Viktor Chechulin

Nun war bekannt, wie das neuronale Netz in die Hardware Schaltung integriert werden musste. Um allerdings auch die Ansteuerung mittels Processing System verstehen und nachbilden zu können, war es notwendig den im Beispiel genutzten Code durch Reverse Engineering zu analysieren. Hierfür wurden die automatisch generierten Projekte vom Beispiel „FINN End2End CNV“ noch weiter inspiziert und analysiert, dabei ist aufgefallen, dass in diesem überall Python benutzt wurde und Python durch die Verwendung eines Interpreters langsam ist, im Vergleich dazu ist C++ schneller durch die Verwendung eines Compilers. Außerdem eignet sich das von FINN generierte nicht für Echtzeitanwendungen, dieser Aspekt wird aber in diesem Projekt durch die Nutzung der Kamera auf dem autonomen Modellauto benötigt. Zusätzlich gibt es auch hier von offizieller Seite aus, keine Dokumentation, aus welcher z.B die Intention für die Nutzung von Python herausgeht, sodass deswegen die Logik aus den generierten Projekten extrahiert wurde und als Grundlage für das umprogrammieren in C++ genutzt wurde. Als Resultat wurde die Logik erfolgreich in C++ umprogrammiert und somit zugeschnitten auf den Projektkontext.

9.1.3 Datentransfer zwischen Processing System und neuronalem Netz

Geschrieben von Philipp Altnickel

Der Datentransfer läuft grundsätzlich über zwei DMA-Blöcke (Direct Memory Access). Der IDMA greift auf eine Speicheradresse zu und liest das zu analysierende Bild ein, welches dort abgelegt werden muss. Über AXI4 werden die entsprechenden Daten eingelesen und mittels AXI-Stream an den eigentlichen Logikblock des neuronalen Netzes weitergegeben. Die Konfiguration erfolgt über einen AXI-Lite Bus. Der ODMA empfängt das Ergebnis des neuronalen Netzes via AXI-Stream und schreibt dieses an eine entsprechende Ausgabestelle im Speicher. Auch hier erfolgt die Konfiguration über AXI-Lite.

Konfigurationsregister IDMA Der IDMA-Block wird mittels AXI-Lite wie folgt konfiguriert. Dabei liegt die Konfiguration an einer Basisadresse im Speicher und die Einstellungen werden jeweils mit einem Offset in feste, 4 Byte große Segmente abgelegt.

Offset	Funktion	Beschreibung
0x00	Aktivierung und Status	Durch Schreiben einer 1 wird der Block aktiviert, ist das zweite Bit aktiv, ist der Lesevorgang abgeschlossen
0x10	Speicheradresse der zulesenden Daten	Die Stelle im Speicher, an der die Bilddaten zum einlesen abgelegt sind
0x1C	Batch Size	Die Anzahl, der zu verarbeitenden Bilder. Die Größe eines Bildes ist festgelegt.

Konfigurationsregister ODMA Der ODMA-Block wird ebenfalls mittels AXI-Lite wie folgt konfiguriert. Dabei liegt die Konfiguration an einer Basisadresse im Speicher und die Einstellungen werden jeweils mit einem Offset in feste, 4 Byte große, Segmente abgelegt.

Offset	Funktion	Beschreibung
0x00	Aktivierung und Status	Durch Schreiben einer 1 wird der Block aktiviert, ist das zweite Bit aktiv, ist der Schreibvorgang abgeschlossen
0x10	Speicheradresse der zuschreibenden Daten	Die Stelle im Speicher, an der die Ergebnisse abgelegt werden
0x1C	Batch Size	Die Anzahl, der zu verarbeitenden Bilder. Die Größe eines Bildes ist festgelegt.

9.1.4 Datenformat

Geschrieben von Felix Müller

Folgend werden das Eingabeformat der Bilddaten und Ausgabeformat der Inferenzresultate beschrieben, welche durch Vorkenntnisse aus dem BNN-PYNQ und aufstellen/testen mehrerer Vermutungen bestimmt wurden. Beispielsweise war für die Resultate durch die Umwandlung mit FINN bereits bekannt, dass nur die höchstbewertete Klasse ausgegeben wird. Der ODMA in Hardware hat eine Datenbreite von 8 Bit. Somit war es wahrscheinlich, dass pro verarbeitetem Bild genau ein Byte mit dem Index der Klasse ausgegeben wird. Diese wurde folgend für mehrere Klassen und Bilder getestet und hat sich bewährt.

Eingabeformat der Bilddaten Die Bilddaten müssen in einem kontinuierlichen Bereich im physischen Speicher liegen, deren Anfangsadresse über die oben genannten Konfigurationsregister gesetzt wird. Dort werden sie pixel- und zeilenweise abgelegt. Jeder Pixel ist drei Byte groß in der Reihenfolge Rot-Grün-Blau.

Ausgabeformat der Inferenzresultate Für das Ausgabe der Resultate können verschiedene Formate über die Graphentransformationen bei der Umwandlung mittels FINN gewählt werden. Analog zur Eingabe der Daten wird die Startadresse im physischen Speicher über die Konfigurationsregister gesetzt. Standardmäßig ist die Ausgabe des Netzes eine $1 \times n$ große Matrix aus uint16-Werten, die für jede der n Klassen des Netzes die Konfidenz angibt. Soll nur die beste Klasse ausgegeben werden, kann in FINN eine sogenannte „TopK“ Operation eingesetzt werden, welche die $1 \times n$ Matrix in eine 1×1 Matrix umwandelt, indem sie den Index der Klasse mit der höchsten Konfidenz berechnet, sodass diese Arbeit direkt auf dem FPGA geschieht und weniger Schreiboperationen in den Speicher nötig sind. Alle Resultate werden aneinandergereiht im Speicher abgelegt mit einem Offset basierend auf dem gewählten Format. Mit TopK wird das Resultat für jedes Bild als ein Byte, der Index der Klasse abgelegt, sodass das Offset in Bytes der Index des Bildes ist. Ohne TopK werden bei n Klassen und der Konfidenz als $\text{uint16 } 2 * n$ Bytes pro Bild benötigt. Für unsere Zwecke ist letzteres Format zielführender, da durch TopK einiges an Informationen verloren geht und selbst bei sehr geringer Konfidenz eine Klasse ausgegeben wird, womit die Anzahl der falsch-positiven Ergebnisse deutlich erhöht wird, da vermutlich erst Ergebnisse über einer gewissen Schwelle wirklich interessant sind.

9.2 Modularisierung des Quellcodes

Geschrieben von Felix Müller, Tuncer Catalkaya

Der Quellcode der Fahrzeugsteuerung ist im Vorgängerprojekt unter starken Zeitdruck iterativ entstanden und besitzt daher wenig Struktur durch fehlende Planung. Um mit der Quellcodebasis zukünftig effizient zu arbeiten, war eine Modularisierung und Dokumentation der Architektur notwendig. Der bisherige Quellcode wurde in C++ geschrieben, aber verwendet dessen Erweiterungen gegenüber C kaum, weshalb zum Zwecke der Modularisierung mehr Gebrauch von Klassen/Objekten gemacht werden sollte.

Die bisherigen Dateien wurden dazu in Klassen mit wohldefinierten Schnittstellen gruppiert und in einem Diagramm, das die Beziehungen zwischen den Klassen (Controller) der Fahrzeugsteuerung (Abb. 42) darstellt. Jede der Klassen erfüllt genau eine Aufgabe, deren Ergebnisse von weiteren Klassen verwendet werden können. Dies unterscheidet die neue Struktur von der vorherigen, deren Funktionsaufrufe sehr undurchsichtig waren.

Die blauen Kästchen in Abbildung 42 sind die verschiedenen Klassen, die als Controller umgesetzt wurden. Die Pfeile zwischen den Klassen symbolisieren eine „Benutzung“. Zeigt z.B. Klasse A auf Klasse B, dann bedeutet dies übersetzt: „Klasse A benutzt Klasse B bzw. Methoden der Klasse B“.

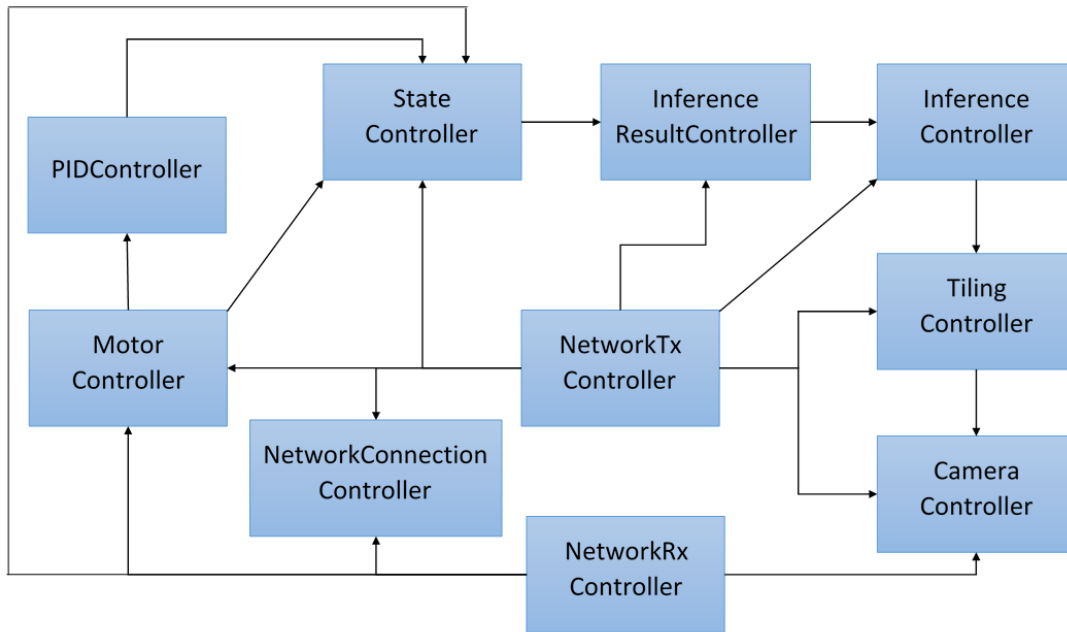


Abbildung 42: Beziehungen zwischen den Controller Klassen der Fahrzeugsteuerung

In Tabelle 9 werden die enthaltenen Methoden von den Klassen (Controller) des obigen Diagramms (Abb. 42) dargestellt. Die Notation ähnelt sehr stark UML. Das «abstract» soll bedeuten, dass es sich um eine abstrakte Klasse/Methode handelt. Die aufgeführten Datentypen/Zeiger entsprechen die von C++.

Klasse	Methoden (+ = public, - = private)
«abstract» Basisklasse: Controller	«abstract» +update(): void
NetworkConnectionController	+update(): void +getServerSocketId(): int +getClientSocketId(): int +getSourceAddress(): sockaddr_in +getRemoteAddress(): sockaddr_in +getSourceAddress(): int -connect(): int
NetworkRxController	+update(): void +base_station_receiveCommandsThread(volatile bool * stop): std::thread -base_station_receiveCommands(volatile bool * stop): void -receiveCommands(): void
CameraController	+update(): void +restart(): void +setResolution(uint32_t width, uint32_t height): void +setTransmitVideo(bool transmitVideo): void +isTransmitVideo(): bool +getImage(): cv::Mat*

Klasse	Methoden
TilingController	+update(): void +getTileBufferRef(): uint8_t* +getTileBufferSize(): uint32_t* +getTileAmount(): uint32_t*
InferenceController	+update(): void +getResults(): std::vector<inference_result_t>
InferenceResultController	+update(): void +isPresent(uint8_t classId): bool +getPosition(std::vector<uint8_t> classIds): position_result_t
StateController	+update(): void +setAutomaticControlMode(bool activated): void +isAutomaticControl(): bool +getVehicleState(): vehicleState -setVehicleState(classification newClassification): void -updateState(vehicleState newState): void
MotorController	+update(): void +setDirection(uint8_t direction): void +setSpeed(uint32_t left, uint32_t right): void +getDirection(): uint8_t +getSpeedLeft(): uint16_t +getSpeedRight(): uint16_t +updateVehicleTurnDirection(vehicleTurnDirection newTurnDirection): void -init(): void -steerVehicle(): void -vehicleHalt(): void -vehicleResume(): void
PIDController	+update(): void +getAngularVelocity(): double -initMotorControl(): void
NetworkTxController	+update(): void +setLastFrameTime(uint32_t lastFrameTime): void -transmitBuffer(uint8_t sendBuffer[]): void -transmitFramebuffer(uint32_t frameTime): void

Tabelle 9: Methoden der Controller Klassen

In den folgenden Abschnitten folgt eine kurze Beschreibung der einzelnen Klassen.

9.2.1 Controller

Geschrieben von Felix Müller

Die Basisklasse *Controller* bietet die Grundlage für alle Schritte der Verarbeitung. Jede erbende Klasse muss die abstrakte Methode *update()* implementieren, welche für jedes Bild aufgerufen wird. Die Reihenfolge der Updates einzelner Controller ist ausschlaggebend, da die Verarbeitung eines Controllers die Ergebnisse eines anderen voraussetzen kann. Nach aktuellem Stand ist diese Ordnung folgendermaßen:

- `NetworkConnectionController`
- `NetworkRxController`
- `CameraController`
- `TilingController`
- `InferenceController`
- `InferenceResultController`
- `StateController`
- `MotorController`
- `PIDController`
- `NetworkTxController`

Die Beschreibung der einzelnen Controller folgt in den kommenden Abschnitten.

9.2.2 NetworkConnectionController

Geschrieben von Tuncer Catalkaya

Der `NetworkConnectionController` dient grundsätzlich dazu eine Netzwerk Verbindung zwischen dem Fahrzeug und der Base Station bereitzustellen und sich zu verbinden. Hierbei handelt es sich um eine UDP (User Datagram Protocol) Verbindung, siehe [Pro80] für genauere Informationen. Es wird stets in der *update()* Methode überprüft, ob die Verbindung hergestellt werden konnte. Falls dies nicht der Fall ist, wird versucht erneut eine Verbindung aufzustellen bis eine hergestellt werden konnte.

9.2.3 NetworkRxController

Geschrieben von Tuncer Catalkaya

Der `NetworkRxController` ist für den Datenaustausch von der Base Station zu dem Vehikel zuständig. Diese Klasse erwartet also eine Eingabe von der Base Station und verarbeitet diese innerhalb der Fahrzeugsteuerung. Die Eingaben werden mittels eines Threads empfangen. Die Eingaben, die empfangen werden, sind Informationen über die Auflösung, dem dargestellten Bild bzw. Video und dem Motor, siehe für genauere Informationen in den Anhang C. Außerdem können Befehle von der Base Station empfangen werden um diese Eingaben „manuell“ zu manipulieren, siehe Tabelle 10 für die verfügbaren Kürzel.

Taste	Funktionalität
R	Wechseln der Bildauflösungen
V	Umschalten(an/aus) des übertragenen Video Signals
M	Wechseln von manuellen und automatischen Modus
Leertaste	Not-Halt (aktiviert manuellen Modus und deaktiviert alle Motoren)
Oben oder W	Bewegt das Fahrzeug vorwärts (im manuellen Modus)
Unten oder S	Bewegt das Fahrzeug rückwärts (im manuellen Modus)
Links oder A	Steuert das Fahrzeug nach links (im manuellen Modus)
Rechts oder D	Steuert das Fahrzeug nach rechts (im manuellen Modus)

Tabelle 10: Base Station Kürzel [Mül+21a]

9.2.4 CameraController

Geschrieben von Felix Müller

Dieser Controller kümmert sich um das Abgreifen von Bildern einer Kamera. Mit jedem Update wird genau ein Bild abgegriffen und zwischengespeichert. Falls kein neues Bild verfügbar ist, bleibt das alte Bild bestehen. Der Begriff „Kamera“ wird in der Implementation sehr abstrakt gehalten und kann die Form einer Bild-/Videodatei oder einer tatsächlichen Kamera einnehmen. Die Anbindung letzterer kann frei gewählt werden, solange ein Treiber für das Video4Linux-Subsystem [Dev21] verfügbar ist. Konkret wurde in diesem Projekt für eine reelle Bildquelle eine über USB angebundene Webcam verwendet (siehe 2.11.2).

9.2.5 TilingController

Geschrieben von Felix Müller

Dieser Controller erzeugt aus dem gepufferten Bild des *CameraControllers* Kacheln unter Verwendung eines oder mehrerer Sliding Windows. Diese Kacheln werden auf die Eingangsgröße des neuronalen Netzes von 32x32 Pixeln skaliert und sequentiell im Speicher abgelegt. Die genaue Erzeugung der Kacheln kann im Bericht des Vorgängerprojektes nachgelesen werden [Mül+21a].

9.2.6 InferenceController

Geschrieben von Felix Müller

Der *InferenceController* kopiert die vom *TilingController* erzeugten Bilder in einen kontinuierlichen Bereich im physischen Speicher, was notwendig ist, da der FPGA auf dem SoC keinerlei Informationen über die Speichereinteilung des Betriebssystems auf der CPU hat. Die Startadresse im physischen Speicher und die Anzahl Bilder wird nun über AXI4-Lite für das neuronale Netz auf dem FPGA konfiguriert. Für die Ausgabe der Resultate wird eine weitere Startadresse und erneut die Anzahl Bilder eingestellt. Nun wird die Ausführung der Inferenz gestartet und auf die Fertigstellung gewartet. Zuletzt werden die Ergebnisse in einen eigenen Datentyp umgewandelt und gepuffert. Dieser Datentyp wurde über ein *struct* definiert und enthält die ursprüngliche Position der Kachel im Gesamtbild, die Höhe/Breite und die Konfidenz der einzelnen Klassen (siehe Listing 4).

```

typedef struct {
    uint16_t x;
    uint16_t y;
    uint16_t width;
    uint16_t height;
    uint16_t[] classConfidences;
} inference_result_t;

```

Listing 4: Datentyp für Inferenzergebnisse der einzelnen Klassen

9.2.7 InferenceResultController

Geschrieben von Felix Müller

Dieser Controller berechnet auf Basis der gepufferten Einzelergebnisse für Kacheln des *InferenceController*s weitere Auswertungen. Eine solche Auswertung ist die Erkennung der Präsenz eines Objektes im Bild, wozu die Ergebnisse aller Kacheln betrachtet werden und ab einem gewissen Schwellwert für die Konfidenz ein interne Zähler erhöht wird. Falls der Zähler einen konfigurierbaren Wert erreicht, wird das Objekt als präsent gewertet, sonst nicht. Eine weitere Auswertung ist die Bestimmung der relativen Position eines Objektes im Gesamtbild über einen gleitenden Durchschnitt der Position von Kacheln, deren Konfidenz für eine gewählte Anzahl Klassen über einem konfigurierten Schwellwert liegt. Die Berechnung für eine Klasse kann im Vorgängerbericht nachgelesen werden [Mül+21a]. In diesem Projekt wurde die Funktionalität um die Verfolgung mehrerer Klassen erweitert. Dies war notwendig, um eine Person unter der Berücksichtigung der Geste zu verfolgen. Konkret sollte die Person verfolgt werden, falls sie sich in der Startpose oder keiner Pose befindet.

Zur Übergabe der Lokalisierungsergebnisse wurde erneut ein eigener Datentyp verwendet, welche bereits im Vorgängerprojekt definiert wurde (siehe Listing 11.5.1). Das Feld *valid* zeigt an, ob das Objekt überhaupt gefunden wurde. Theoretisch könnte dieser Wert auch allein für die Erkennung der Präsenz eines Objektes verwenden, allerdings ist die Lokalisierung deutlich aufwendiger, sodass, wie oben genannt, eine separate Berechnung implementiert wurde. Das Feld *highestConfidence* beschreibt die höchste Konfidenz mit der eines der gesuchten Objekte in einer Kachel erkannt wurde. Passend dazu zeigt *correctClassCount* die Anzahl Kacheln an, welche zumindest für eines der gesuchten Objekte den gesetzten Schwellwert überschritten haben. Zuletzt geben *x* und *y* die berechneten Koordinaten der gesuchten Objekte im Bild an.

```

typedef struct {
    bool valid;
    int highestConfidence;
    int correctClassCount;
    uint32_t x;
    uint32_t y;
} position_result_t;

```

Listing 5: Datentyp für Lokalisierungsergebnisse

9.2.8 StateController

Geschrieben von Lukas Schimanski, Tuncer Catalkaya

Der StateController setzt das in Abbildung 43 dargestellte Zustandsdiagramm um. Beim Aufruf des Updates wird die aktuelle Klassifizierung vom InferenceResultController abgerufen und auf Basis des Klassifizierung und des aktuellen Zustands ein neuer Zustand gesetzt. Abhängig vom aktuellen Zustand werden vom Inference ResultController nur die Klassen abgerufen, die den Zustand des Wagens ändern. Der Wagen startet initial im „Manual Mode“.

Umgesetztes Zustandsdiagramm für die Gestenerkennung Es wurde ein Zustandsdiagramm entwickelt um die möglichen Zustände und Zustandswechsel des Vehikels festzuhalten. Dieses Diagramm wird in Abbildung 43 dargestellt. Hierbei war es wichtig zunächst ein simples Zustandsdiagramm zu implementieren um so erst mal zu zeigen, dass eine Implementation überhaupt möglich ist. Dabei hat sich folgende Zustandsmenge ergeben $Z = \{z_1, z_2, z_3\}$, wobei der Startzustand $z_0 = z_1$ ist. Die Zustände und dessen Beschreibungen werden in Tabelle 11 dargestellt.

Zustand	Name	Beschreibung
z_1	MANUAL_MODE	Vehikel kann manuell von einer Person über der Base Station gesteuert werden
z_2	SEARCH_MODE	Vehikel ist im Stillstand und dreht sich um eine Start Geste zu erkennen
z_3	APPROACH_MODE	Vehikel fährt auf eine Start Geste zu solange die Start Geste erkannt wird

Tabelle 11: Zustandsmenge mit Beschreibung der Zustände

Die Notation der Zustandsdiagramme in diesem Bericht entspricht der UML Notation. Detaillierte Informationen zu der Notation können in [Par21] nachgelesen werden.

Das Zustandsdiagramm (Abb. 43) kann grundsätzlich in zwei verschiedene Bereiche unterteilt werden. Der eine Bereich ist alleinig der MANUAL_MODE in dem das Vehikel manuell über der Base Station gesteuert werden kann (vgl. Tabelle 10). Der zweite Bereiche beinhaltet den SEARCH_MODE und APPROACH_MODE, welche zusammen den autonomen Bereich ergeben. In diesem Bereich wird das neuronale Netz in Einsatz genommen um Entscheidungen zu treffen, ob das Fahrzeug nach einer Geste sucht (SEARCH_MODE) oder sich auf eine Person, oder in diesem Falle Start Geste, zubewegt.

Der Ablauf des Zustandsdiagramm aus Abbildung 43 kann folgendermaßen festgehalten werden. Sollte die Fahrzeugsteuerung, also die vehicleps, gestartet werden, dann ist der initiale Zustand der MANUAL_MODE. Der MANUAL_MODE wurde als Startzustand festgelegt, damit das Vehikel, nach dem Starten der Fahrzeugsteuerung, nicht versehentlich wegfährt, falls z.B. beim deployen der vehicleps das Fahrzeug allein gelassen wird. Denn im MANUAL_MODE ist das Fahrzeug zunächst im Stillstand und bewegt sich erst, wenn explizit ein Kommando von der Base Station empfangen wurde. Vom MANUAL_MODE kann in den SEARCH_MODE gewechselt werden. Dazu muss das Automatik Kommando von der Base Station empfangen werden, die Taste „M“. Die Fahrzeugsteuerung bleibt im SEARCH_MODE bis eine Start Geste erkannt wird. Sollte eine Start Geste erkannt werden, dann wird in den APPROACH_MODE gewechselt. Hierbei verfolgt das Fahrzeug die erkannte Start Geste bis keine mehr erkannt werden konnte. Sollte keine Start Geste mehr erkannt werden, dann wechselt die Fahrzeugsteuerung in den SEARCH_MODE, bei der erneut eine Start Geste gesucht wird. Es kann jederzeit in den MANUAL_MODE gewechselt werden, hierzu muss das manuelle Kommando von der Base Station empfangen werden, die Taste „M“.

Das in Abbildung 43 dargestellte Zustandsdiagramm ist das was letztlich in der Fahrzeugsteuerung (vehicleps) umgesetzt wurde. Bedauerlicherweise konnte das Zustandsdiagramm aber nicht getestet werden, nähere Informationen dazu sind in den späteren Kapiteln (11 und 12).

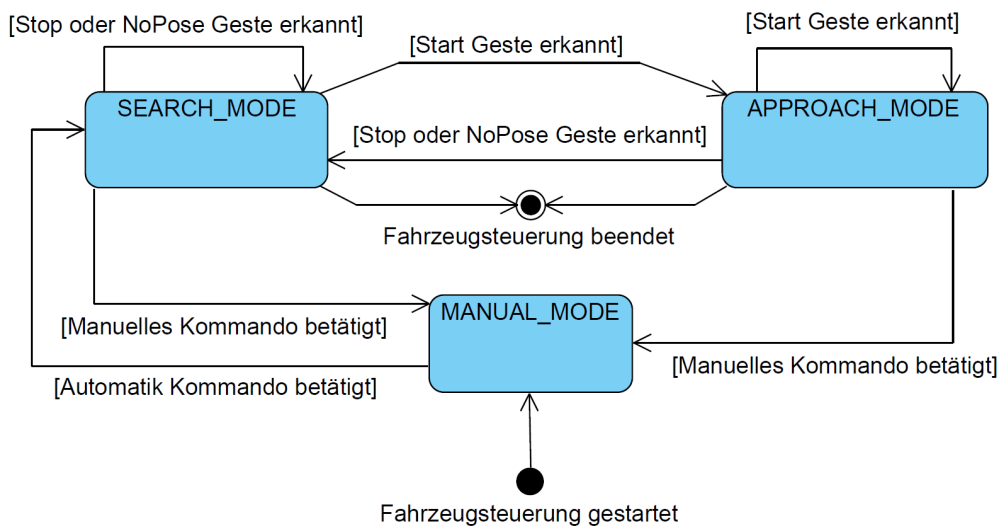


Abbildung 43: Umgesetztes Zustandsdiagramm der Fahrzeugsteuerung mit Gestenerkennung

Ursprünglich war geplant ein anderes Zustandsdiagramm umzusetzen. Auf dieses geplante Zustandsdiagramm wird im nächsten Paragraphen weiter eingegangen.

Eigentlich geplantes Zustandsdiagramm für die Gestenerkennung Ursprünglich war es geplant das Zustandsdiagramm in Abbildung 44 umzusetzen. Hierzu würde eine vierte Klasse bzw. Geste benötigt werden, die vom neuronalen Netz erkannt werden muss. Es handelt sich hierbei um die Klasse „Kein Mensch“ bzw. „Keine Person“ oder kann auch als „Unklassifiziert“ (Unclassified) aufgefasst werden. Also eine Klasse bei der das neuronale Netz klar und deutlich entscheidet, dass innerhalb eines Bilds keine der definierten Posen/Gesten erkannt wurde. Mit der momentanen Implementierung (ohne die zuvor beschriebene vierte Klasse) kommt es leider dazu, dass das neuronale Netz die Gesten, die nicht definiert sind, einfach als „NoPose“ ansieht. Damit fehlt letztlich also die erwähnte vierte Klasse.

Bei dem eigentlich geplanten Zustandsdiagramm (Abb. 44) wird grundsätzlich der autonome Bereich um einen weiteren Zustand namens APPROACH_SEARCH_MODE erweitert. Der APPROACH_SEARCH_MODE verhält sich genau wie der APPROACH_MODE. Das Vehikel dreht sich also, um eine bestimmte Geste zu erkennen. Der Unterschied zu den beiden Zuständen ist der Zweck, der noch im Verlaufe dieser Beschreibung implizit verdeutlicht wird.

Der wesentliche Unterschied zu dem umgesetzten Zustandsdiagramm ist, dass nun die Start Geste dazu verwendet wird um eine Art Tracking Modus zu starten. Der Tracking Modus besteht in diesem Falle aus dem APPROACH_MODE und APPROACH_SEARCH_MODE. Hierbei versucht das Fahrzeug einer Person zu folgen (NoPose oder Start Geste). Sollte er keine Person erkennen (die vierte fehlende Klasse), dann soll das Vehikel sich drehen und erneut versuchen eine Person (NoPose oder Start Geste) aufzufinden. Sollte dies erfolgreich sein, dann „verfolge diese Person“. Dieser Tracking Modus, also der ständige Wechsel von APPROACH_MODE und APPROACH_SEARCH_MODE, wird wiederholt bis eine Stop Geste erkannt wird. Dann geht die Fahrzeugsteuerung zurück in den SEARCH_MODE um erneut eine Start Geste aufzufinden.

Der große Vorteil von dem eigentlich geplanten Zustandsdiagramm ist, dass so die definierten Gesten alle einen eindeutigen Zweck haben. Außerdem kann in einem abstrakten Sinne der SEARCH.MODE als eine Art Wahl Zustand angesehen werden. Stellen wir uns vor, dass die Start Geste umbenannt wird

zu Tracking Geste. Dann würde die Tracking Geste dazu führen, dass die Fahrzeugsteuerung in den zuvor beschriebenen Tracking Modus übergeht, also dem ständigen Wechsel zwischen APPROACH_MODE und APPROACH_SEARCH_MODE. Nun kann aber als Erweiterung eine weitere beliebige Geste definiert werden um so eine weitere Funktionalität umzusetzen. In diese neue Funktionalität durch die neue Geste wird erneut von dem SEARCH_MODE gewechselt. Somit ist also wie Anfangs beschrieben der SEARCH_MODE eine Art Wahl Zustand bei dem entschieden werden kann in welche Funktionalität er übergehen soll. In dem momentanen Falle gibt es natürlich nur eine Funktionalität (den Tracking Modus). Die Stop Geste ist eine Art Unterbrechung, also eine Möglichkeit um jederzeit zurück in den Wahl Zustand zurückzukommen bzw. eine Funktionalität abzubrechen. Zur Optimierung müsste wahrscheinlich noch eine minimale Anzahl an erkannten Stop Gesten in Einsatz genommen werden, damit die Fahrzeugsteuerung nicht ständig, bei nur einer erkannten Stop Geste, in den SEARCH_MODE wechselt.

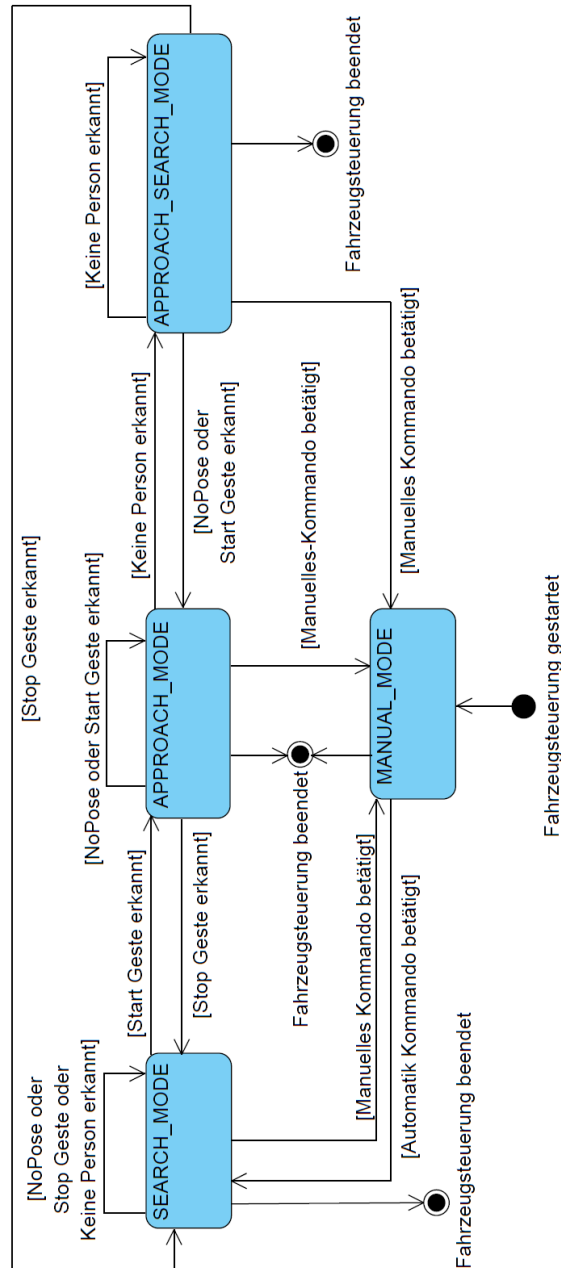


Abbildung 44: Geplantes Zustandsdiagramm der Fahrzeugsteuerung mit Gestenerkennung

9.2.9 MotorController

Geschrieben von Lukas Schimanski

Der Motorcontroller umfasst alle Funktionalitäten die Motoren des Fahrzeugs zu steuern sowie Informationen des Motors dem TxController bereitzustellen.

Hierzu zählen das Abrufen der Geschwindigkeit des linken und rechten Motors sowie deren Richtung. Mit Hilfe des RxControllers und der Basestation können die Motoren einzeln angesprochen werden, sodass das Fahrzeug im Manual Mode über die Basestation gesteuert werden kann.

9.2.10 PIDController

Geschrieben von Lukas Schimanski

Der PIDController spricht den PID Regler an, der den optimalen Winkel berechnet in dem das Fahrzeug das Ziel anfahren soll. Das Ergebnis dieser Berechnung wird dann dem MotorController zur Verfügung gestellt um die Motoren entsprechen anzusteuern. Die Berechnungen des Controllers wurden aus Kapitel 6.5 [Mül+21a] übernommen.

9.2.11 NetworkTxController

Geschrieben von Tuncer Catalkaya

Der NetworkTxController ist für den Datenaustausch von dem Fahrzeug zu der Base Station zuständig. Diese Klasse sendet also die verarbeiteten Eingaben der Fahrzeugsteuerung zu der Base Station, welche wiederum innerhalb der Base Station visuell dargestellt werden. Die genauen Informationen, die zur Base Station übertragen werden, können im Anhang C näher nachgelesen werden. Zusätzlich wird das Bild bzw. Video an die Base Station übertragen. Der *colorMode* des übertragenen Bild bzw. Videos ist konstant RGB565 (2 bytes pro Pixel im Format 5-bit rot, 6-bit grün und 5-bit blau), auf den *colorMode* wird in Kapitel 10 weiter eingegangen.

9.2.12 ConfigController

Geschrieben von Philipp Altnickel

Die Aufgabe des ConfigControllers ist es, eine einheitliche Schnittstelle zum Laden von konfigurierten Einstellungen zu schaffen. Alle Einstellungen werden in der Datei *vehicle.conf* vorgenommen. Der Controller ist als Singleton implementiert, sodass die einzige Instanz vom ganzen Programm aus erreicht werden kann. Die Konfigurationsdatei kann so für das gesamte Programm einmal geladen, aber von jeder Klasse aus erreicht werden. Die Klasse bietet dann die Möglichkeit, Einstellungen anhand ihres Namens und des gewünschten Typs int, unsigned int, double und String abzurufen.

9.2.13 Logger

Geschrieben von Philipp Altnickel

Die Loggerklasse bietet die Möglichkeit, programmweit einen einheitlichen Weg der Textausgabe in die Konsole zu erhalten. Dabei kann für je Klasse ein Objekt mit dessen Namen erstellt werden. Der Logger wird dann die an ihn übergebenen Nachrichten in die Konsole ausgeben und automatisch den Namen der Klasse, sowie wenn gewünscht eine Warnungsmarkierung hinzufügen. Das Logging-Level kann dabei programmweit zwischen Alle, Warnung und Keine gewechselt werden. Dies bietet große Vorteile, da möglicherweise Debugging-Ausgaben über das Programm verteilt und vergessen werden.

9.2.14 DMA

Geschrieben von Philipp Altnickel

Die DMA Klasse übernimmt komplett die Aufgabe der Kommunikation mit dem neuronalen Netz. Sie stellt in Prinzip die Implementierung der Erkenntnisse aus Abschnitt 9.1 dar. Die schon aus dem Vorgängerprojekt bestehenden Klassen MMIO und DMABuffer beinhalten die Allokierung von Speicher sowie das Schreiben auf entsprechende Speicherstellen. D.h. das eigentliche Allokieren von Speicher, sowie das Ansprechen der angeforderten Speicherstellen haben schon immer diese Klassen übernommen. Die DMA Klasse abstrahiert komplett von dieser dahinter liegenden Logik der Speicherstellen und der entsprechenden Adressen. Es stellt Schnittstellen dar, sodass im Programm einfach die im Blockschaltplan der Hardwareschaltung sichtbaren DMA Blöcke als Objekte initialisiert werden können. Es kann speziell für die hier verwendeten DMA Blöcke ein Objekt mit entsprechenden, in Abschnitt 9.1 ermittelten, Speicheradressen erzeugt werden. Die Konfiguration, das Füllen und Auslesen der Speicherbuffer, das Starten der Übertragung und auch das warten auf Ergebnisse übernimmt alles diese Klasse. Außerhalb müssen keine Speicheradressen für den AXI-Bus oder genauere Kenntnisse über Implementierungen u.a. der Warteschleife bekannt sein. Einfache Methodenaufraufe erlauben eine simplere Nutzung der DMA Blöcke und somit eine einfache Kommunikation mit dem neuronalen Netz.

10 Anpassung der Base Station des Vorgängerprojektes

Geschrieben von Tuncer Catalkaya

Der Quellcode der Base Station vom Vorgängerprojekt (siehe [Mül+21a]) musste angepasst werden an das Vorhaben dieses Projekts. Dies beinhaltet die Anpassung an die **Gestenerkennung** und dem „neu“ implementierten **NetworkTxController**. Auf diese Änderungen wird in diesem Kapitel näher eingegangen.

10.1 Anpassung für die Gestenerkennung

Für die Anpassung an die Gestenerkennung musste lediglich das Array in der *framebuffer.hpp* angepasst werden. Hier wurde die Anzahl an Klassen von 44 (vom Vorgängerprojekt) auf 3 (für die umgesetzte Gestenerkennung relevant) gesetzt, siehe Kapitel 9.2.8 für genauere Informationen zu den Zuständen. Hierbei war es wichtig, dass die Klassen alphabetisch sortiert sind um sich so der Ausgabe des umgesetzten neuronalen Netzes anzupassen. Dadurch ist folgende Reihenfolge mit folgenden Bezeichnern entstanden (siehe Kapitel 4.2 für nähere Informationen zu den Gesten):

1. **NoPose**: Der Mensch, wenn er keine definierte Pose einnimmt
2. **Start**: Der Mensch, wenn er die Start Pose einnimmt
3. **Stop**: Der Mensch, wenn er die Stop Pose einnimmt

10.2 Anpassung der Base Station an den NetworkTxController

Bei der Anpassung an den NetworkTxController (siehe Kapitel 9.2.11) mussten lediglich Features bzw. Felder entfernt werden. Hierbei handelt es sich um Felder, die im Vorgängerprojekt implementiert wurden, sich aber als „Unnützlich“ bewährt haben. Dies betrifft folgende Features bzw. Felder:

- **tileToShow**: Zeigt ein Tile in der Base Station an (auf Tastendruck wird durch die Tiles rotiert)
- **colorMode**: Wechselt die Farben des angezeigten Video bzw. Bilds (fest definierte Farboptionen) auf Tastendruck

Das *tileToShow* wurde komplett entfernt. Bei dem *colorMode* wurde statt dem Wechseln der Farben ein fest definierter Farbmodus gewählt, es handelt sich hierbei um *RGB565* (2 bytes pro Pixel im Format 5-bit rot, 6-bit grün und 5-bit blau). Es mussten außerdem die Netzwerkdatenrahmen Nummern angepasst werden. Zuvor gab es 5 Netzwerkdatenrahmen im Vorgängerprojekt, dies hat sich nun durch die Änderungen der beiden Features bzw. Felder auf 3 reduziert (vgl. Anhang C).

11 Evaluation

Geschrieben von Reena Wichmann

Innerhalb der Evaluation werden die Ergebnisse des Projekts untersucht. Es soll hier herausgestellt werden, welche Aspekte erfolgreich verliefen und in ein mögliches Folgeprojekt übernommen werden können und bei welchen es Schwierigkeiten gab und aus welchem Grund. Zudem ließ sich die eigenen Arbeit damit überprüfen und bewerten.

11.1 Genauigkeit der Netztopologie (Variante 1)

Geschrieben von Reena Wichmann, Toni Dragojevic, Mattia Uhlenbrock, Stanislav Voytas, Ismail Sastim, Jonas Philipp

Zur Feststellung der Genauigkeit der Netztopologie wurden Tests vorgenommen, um festzustellen, ob die für das Training mit dem Gestendatensatz festgelegten Parameter korrekt gewählt wurden beziehungsweise mit welchen Parametern es sich lohnt weiter zu forschen.

11.1.1 Evaluationstests mit Version 4 des Datensatzes

Zum Aufbau der Evaluationstests ist festzuhalten, das es sich bei der Komplexität des Netzes als zu zeitaufwendig darstellt, die Layer des Modells neu zusammenzufügen, indem Layer hinzugefügt oder weggelassen werden. Stattdessen wurden die Parameter Epochen und Batch Size und der Datensatz verändert. Hierbei wurden als Grundlage die zuvor verwendeten Parameter von 1000 Epochen, Batch Size 32 und Gesten Datensatz Version 4.0, festgelegt. Um eine erfüllbare Testmenge zu haben, wurde innerhalb jedes Tests in ersten Durchlauf nur einer der Parameter verändert und die restlichen Grundlagenparameter beibehalten, womit sich eine Testmenge von sieben ergibt.

Es wurde sich zudem darauf geeinigt, die aus den trainierten Modellen entstehenden .pth und .onnx Dateien und die zugehörige .html Dateien zu jedem Test zu speichern, um das trainierte Testmodell wiederverwenden zu können. Somit kann das Modell auch für andere Testfotos verwendet werden und aus diesen eine Genauigkeit bestimmt werden. Die Dateien sind unter `src/training/exports/evaluation` im `soc-nn_main_repo`-Repository zu finden [Wic+21]. Zudem lassen sich die innerhalb der Evaluation durchgeführten Tests mithilfe der festgehaltenen Parameter reproduzieren.

Innerhalb der Durchführung wurde das neuronale Netz mit den festgelegten Parametern trainiert und mithilfe der Testfotos aus dem Gestendatensatz eine Genauigkeit festgestellt. Im Folgenden sind die Parameter, die Version des Datensatzes und die Ergebnisse der Testreihe aus dem ersten Durchlauf der Evaluationstests festgehalten:

Test Nr.	Name Modell (Quelle)	Epochen	Batch Size	Datensatz	Genauigkeit
v4.0_1	CNV_1W1A [Papa]	250	32	Gesten Datensatz V4	≈ 61,9%
v4.0_2	CNV_1W1A [Papa]	500	32	Gesten Datensatz V4	≈ 52,38%
v4.0_3	CNV_1W1A [Papa]	1000	32	Gesten Datensatz V4	≈ 61,9%
v4.0_4	CNV_1W1A [Papa]	1000	16	Gesten Datensatz V4	≈ 52.38%
v4.0_5	CNV_1W1A [Papa]	1000	64	Gesten Datensatz V4	≈ 52.38%
v4.1_1	CNV_1W1A [Papa]	1000	32	Gesten Datensatz V4.1	≈ 61,9%
vcifar_1	CNV_1W1A [Papa]	1000	32	CIFAR10	≈ 73.44%

Tabelle 12: Genauigkeitsresultate Evaluationstests mit Datensatz Versionen V4.0 V4.1 und CIFAR10

Es ist zu erkennen, dass der Test mit dem CIFAR10 Datensatz die höchste Genauigkeit erzielt hat, was sich durch die deutlich größere Menge an Daten erklären lässt. Dieser ist für unser Projekt keine Option, da Gesten erkannt werden sollen und sich somit auf den Gestendatensatz festgelegt wurde. Es ist jedoch für die Evaluierung festzuhalten, dass das gewählte Modell bei geeignetem Datensatz eine hohe Genauigkeit von über 70% erreichen kann.

Zudem haben diese Evaluierungstests gezeigt, dass das Training mit sowohl 250 als auch 1000 Epochen die gleiche Genauigkeit erreicht. Dies muss erneut mit einer größeren Anzahl an Testfotos überprüft werden, da aktuell nur mit 21 Fotos getestet wird. Nach den Tests lässt sich zudem festhalten, dass die Batch Size 32 beibehalten werden sollte, da mit den Batch Sizes 16 und 64 keine besseren Ergebnisse erzielt wurden.

11.1.2 Evaluationstests mit Version 5 des Datensatzes

Es wurde ein weiterer Durchlauf benötigt, da der Gestendatensatz zu einer neuer Version gebracht wurde, in dem die Qualität der Fotos überprüft und unpassende Fotos aussortiert wurden, wie in Kapitel 4.3 beschrieben. Ziel der weiteren Tests war es, die beste Zusammenstellung der Parameter Epochen, Batch Size und Datensatzversion rauszustellen, damit mit diesen weiter geforscht werden kann. In diesen Tests wurde die gleiche Vorgehensweise wie zuvor gewählt wurde und zusätzlich wurden die Erkenntnisse des ersten Durchlaufs beachtet.

Zum Aufbau lässt sich sagen, dass jeder Test mit dem gleichen Modell CNV_1W1A [Papa] wie im ersten Durchlauf durchgeführt wurde und die zuvor genannten Parameter verändert wurden. Bei Version 5.1 handelt es sich um die gleichen Fotos wie in Version 5.0 jedoch inklusive rotierten beziehungsweise gespiegelten Versionen, wodurch der Datensatz vergrößert wird.

Das Modell wurde erneut mit den Testfotos, die im Datensatz Version 5.0 beziehungsweise 5.1 enthalten sind, validiert und die Genauigkeit festgehalten. Im Folgenden sind die enthaltenen Tests und dessen Ergebnisse des zweiten Durchlaufes festgehalten:

Test Nr.	Name Modell (Quelle)	Epochen	Batch Size	Datensatz	Genauigkeit
v5.0.1	CNV_1W1A [Papa]	250	32	Gesten Datensatz V5.0	90%
v5.0.2	CNV_1W1A [Papa]	500	32	Gesten Datensatz V5.0	80%
v5.0.3	CNV_1W1A [Papa]	1000	32	Gesten Datensatz V5.0	80%
v5.0.4	CNV_1W1A [Papa]	250	16	Gesten Datensatz V5.0	80%
v5.0.5	CNV_1W1A [Papa]	250	64	Gesten Datensatz V5.0	≈ 83.3%
v5.1.1	CNV_1W1A [Papa]	500	32	Gesten Datensatz V5.1	43,75%
v5.1.2	CNV_1W1A [Papa]	100	32	Gesten Datensatz V5.1	18.75%
v5.1.3	CNV_1W1A [Papa]	250	32	Gesten Datensatz V5.1	34,375%

Tabelle 13: Genauigkeitsresultate Evaluationstests mit Datensatz Versionen 5.0 und 5.1

Werden die Evaluationstests des zweiten Durchgangs untereinander verglichen wird deutlich, dass ein Training des Modells mit 250 Epochen ausreichend ist und keine 500 oder 1000 Epochen benötigt werden, da sich das Ergebnis dadurch nicht verbessert. Das lässt sich als positiv bewerten, da es weniger zeitaufwendig ist. Es wird zudem deutlich, dass ein trainiertes Modell mit der Datensatz Version 5.1 wie es in den Tests v5.1.1, v5.1.2 und v5.1.3 ausgeführt wurde, zu einer sehr geringen Genauigkeit führt. Daraus lässt sich schließen, dass der Ansatz mit rotierten und gespiegelten Versionen der Fotos zu arbeiten in dieser Form keine Verbesserung erzielt. Die Batch Size auf 16 oder 64 wie in Test v5.0.4 und v5.0.5 zu verändern erreicht innerhalb der Ausführung Genauigkeiten von 80% und 83,3% und führt dementsprechend zu keiner Verbesserung im Vergleich zu Test v5.0.1.

11.2 Evaluationstest mit Version 6 des Datensatzes

Es wurde ein Test mit Version 6 des Datensatzes durchgeführt, um die Hypothese eines positiven Lern-Effekts von Zhang et al. [Zha16] nachzuweisen. Hierfür wurde eine weitere Ausgabe-Neurone hinzugefügt: "UNCLASSIFIED".

Test Nr.	Name Modell (Quelle)	Epochen	Batch Size	Datensatz	Genauigkeit
v6.0_1	CNV_1W1A [Papa]	110	32	Gesten Datensatz V6.0	25%

Tabelle 14: Genauigkeitsresultate Evaluationstests mit Datensatz Version 6.0

Die Genauigkeit fiel bei diesem Test mit V6 auf 25%. Wegen der niedrigen ANzahl an Testdurchläufen ist dieses Ergebnis jedoch nicht aussagekräftig.

11.2.1 Auswertung Evaluation

Im Zuge der Auswertung aller Evaluationstests lässt sich erkennen, dass mit der neuen Datensatz Version 5.0 eine deutlich höhere Genauigkeit erreicht werden kann, als mit der Datensatz Version 4.0. Der Test v5.0_1, der mit der Version 5.0 des Datensatzes und nur 250 Epochen trainiert wurde, erreichte mit 90% bei der Durchführung die höchste Genauigkeit. Im Vergleich zu den Evaluationstests mit der Datensatz Version 4.0, bei der höchstens 61,9% erreicht wurden, handelt es sich um eine deutliche Steigerung. Es zeigt sich, dass die Überprüfung der Qualität des Datensatzes zu einer Verbesserung führt und in Zukunft weiterhin die Richtlinien bei der Erstellung neuer Fotos beachtet werden sollten.

In Zukunft kann es sinnvoll sein die Parameter aus dem Test v5.0_1 zu nutzen, wenn ein Modell trainiert werden soll, da mit diesem die höchste Genauigkeit erreicht wurde. Zudem kann mit diesen Parametern getestet werden, wenn durch weitere hinzugefügte Fotos oder andere Veränderungen neue Datensatzversionen entstehen.

Name Modell (Quelle)	Epochen	Batch Size	Datensatz	Genauigkeit
CNV_1W1A [Papa]	250	32	Gesten Datensatz V5.0	90%

Tabelle 15: Parameter zur Erzielung der höchsten Genauigkeit

11.3 Genauigkeit der Netztopologie (Variante 2)

Geschrieben von Fannese Tchang

Für die Evaluierung unseres CNV Model würde ein anderer Aspekt zusätzlich untersucht. Da unserer Modell keine zufrieden stehenden Ergebnisse lieferten, wurde neue Schichten hinzugeführt, um das Modell mit der Hoffnung zu optimieren. Diese Schichten sind: Relu, Droptout quantlineare. Mit Hilfe dieser Schichten kann man die Überanpassung während des Trainingsprozesses verhindern, ein lineares Bild zu einem nichtlinearen Bild führen bzw. die Bilder wie möglich zu grauen und weißen Bilder zu führen. Und eine erweiterte Quantisierte lineare Transformation. Damit wurde beim Training mit der Gesten-Datensatz Version 3 und mit einem Batchsize von 100 eine Genauigkeit von 33,33% erreicht. Die Erhöhung der Anzahl der Schichten oder Neuronen führte nicht zu einer Steigerung der Genauigkeit mit der Gesten-Datensatz Version 3.

Man muss bedenken, dass bei der Optimierung nicht nur die Genauigkeit steigen muss sondern auch die Performance von dem Model damit das Fahrzeug mehrere Bilder pro Sekunde erkennen kann. D.h. dass, das CNV Model eine größere Genauigkeit liefern könnte, die jedoch eine schlechte Performance am Fahrzeug hat.

11.4 Ressourcenverbrauch der erzeugten Schaltung

Geschrieben von Ismail Sastim

Nach der vollständigen Übersetzung von trainiertem Neuronalem Netz bis hin zu IP-Blöcken in Vivado (siehe Anhang J) konnte folgende Hardwarebelegung des verwendeten *PYNQ-Z1* Boards durch Vivado ermittelt werden.

Ressource	Benutzung	Verfügbar	Auslastung in %
LUT	24254	53200	45,59
LUTRAM	1636	17400	9,40
FF	33024	106400	31,04
BRAM	98	140	70,00
IO	6	125	4,80
BUFG	1	32	3,13

Tabelle 16: Ressourcenauslastung laut Vivado Summary

Es ist erkenntlich, dass das Board insgesamt noch genügend Ressourcen zur Verfügung hat. Die größte Auslastung ist am Block RAM festzustellen. Allerdings ist mit 70% auch hier noch genügend Puffer enthalten.

11.5 Systemevaluation

11.5.1 Testvideos

Geschrieben von Lukas Schimanski

Für eine reproduzierbare Evaluation des Gesamtsystems wurden zwei Videos aufgenommen (*gesten.mp4* und *gesten2.mp4*) [Mül+21b]. Das erste Video ist an die Trainingsdaten angelehnt und wurde aus einer Entfernung von ca. 1,5m und einer Höhe von ca. 70cm aufgenommen. Es wurde hierbei darauf geachtet, dass der Kontrast zwischen der Person im Vordergrund und dem Hintergrund so groß wie möglich ist, um eine klare Abgrenzung von Objekten für das Netz zu ermöglichen.

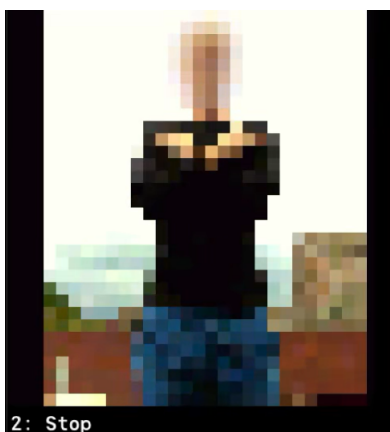


Abbildung 45: Klassifizierung „Stop“ in der Basestation

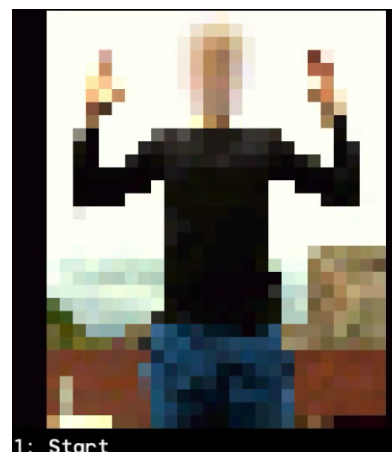


Abbildung 46: Klassifizierung „Start“ in der Basestation

Das zweite Video wurde zur Evaluation des Trackings aufgenommen und entspricht perspektivisch der Sicht, die das Fahrzeug vom Boden aus auf seine Umgebung hat.

Die Entfernung zwischen Kamera und Person entspricht ca. 3m. Es wurde ebenfalls wieder darauf geachtet, dass der Kontrast zwischen Person und Hintergrund so groß wie möglich ist.



Abbildung 47: Tracking der Person in der Baestation mit Person im Bildzentrum



Abbildung 48: Tracking der Person in der Baestation mit Person am Bildrand

Beide Videos wurden auf einer Smartphonekamera in Full-HD aufgezeichnet und im Nachhinein auf eine Auflösung von 800x800 für das Klassifizierungsvideo und 800x450 für das Trackingvideo skaliert.

Das Resultat des Klassifizierungstests sowie des Trackingtests sind im vehicleps-Repository in den Dateien (Klassifizierung.m4v bzw. Tracking.m4v) unter res/videos zu finden. [Mül+21b]

Für den Klassifizierungstests wurde eine Auflösung (INITIAL_RESOLUTION.WIDTH / INITIAL_RESOLUTION.HEIGHT) von 32x32 Pixeln sowie eine Tilekonfiguration von 32x32 mit einem Stride von jeweils 0.25 verwendet.

Für den Trackingtest wurde eine Auflösung von 352x288 Pixeln und folgende Tilekonfiguration verwendet:

```
TILE_SPEC="32,32,.5,.5, 64,64,.25,.25, 128,128,.25,.25"
```

11.5.2 Testergebnisse

Geschrieben von Philipp Altnickel

Das Ziel des ersten Videos war es, ein einziges, genau spezifiziertes Tile in das neuronale Netz herein zugeben und für dieses Tile eine Klassifizierung zu erhalten. Somit sollte die Genauigkeit der Klassifizierung durch das neuronale Netze unabhängig von jeglicher Steuerungslogik des Processing Systems, anhand eines realitätsnahen Beispiels gezeigt werden. Erkennbar ist, dass die Klassifizierung der Pose in der tendenziell funktioniert. Jede der drei Posen wird zum überwiegenden Teil der Versuche korrekt erkannt. Allerdings ist auch sichtbar, dass es durchaus falsche Klassifizierungen gibt. Es gibt sehr häufig kurze Abweichungen zu einer anderen, nicht korrekten Pose. Im Ergebnis kann man zu diesem Test festhalten, dass eine Erkennung einer Pose mit diesem neuronalen Netz durchaus möglich ist. D.h. sowohl die Kommunikation des Processing System mit der digitalen Logik des neuronalen Netzes auf dem FPGA, als auch die Verarbeitung innerhalb des neuronalen Netzen sind grundsätzlich funktionsfähig. Über die reale Einsatzfähigkeit sagt dieser Test allerdings nur bedingt etwas aus, da hier ein Tile vom Ausschnitt der Person und auch von inhaltlichen Eigenschaften, wie dem Kontrast von Person zu Hintergrund, speziell angepasst wurden.

Der Test mit dem zweiten Video sollte dazu dienen, die Fähigkeit der Bestimmung der Position einer Person in einem realitätsnahen Umfeld zu bewerten. Es zeigt sich deutlich, dass das System extreme Schwierigkeiten hat, die Position der Person im Bild zu bestimmen. Das Kreuz befindet sich eigentlich immer sehr mittig im Bild, was dafür spricht, dass das Netz über das ganze Bild verteilt Personen erkennt und der Mittelwert somit immer zentral landet. Es ist also eher als Zufall anzusehen, dass das Kreuz bei einer mittig stehenden Person sehr genau trifft. Sobald sich die Person an den Bildrand bewegt, verändert sich die Position des Kreuzes nicht wirklich. Eine sinnvolle Umsetzung der Projektziele ist also mit dem aktuellen Stand von Software und neuronalem Netz nicht möglich. Sowohl die Genauigkeit des neuronalen Netzes, als auch die Verarbeitung innerhalb der Software müssen verbessert werden, damit ein zuverlässiges Tracking möglich ist.

11.5.3 Versuche zur Optimierung

Geschrieben von Philipp Altnickel

Wie die zuvor beschriebenen Ergebnisse gezeigt haben, ist eine sowohl mit der entwickelten Software, als auch mit der Hardwareschaltung des neuronalen Netzes grundsätzlich möglich, Posen zu identifizieren und auch Personen zu verfolgen. Dies bildet den Grundstein dafür, dass das Fahrzeug die als Projektziel gesetzten Reaktionen durchzuführen. Um zu evaluieren, wie gut diese Erkennung im integrierten System funktioniert und damit schlussendlich auch die Fahrzeugsteuerung agieren kann, wurden verschiedene Versuche unternommen, bestehende in der Software vorgesehene Parameter anzupassen und auch einzelne Verarbeitungsschritte in der Software zu ändern. Ziel war es auch, das beste Ergebnisse aus dem eigens erstellten neuronalen Netz herauszuholen und die Funktion zu optimieren.

Alle folgenden Versuche wurden so durchgeführt, dass entweder aufgenommene Testbilder und Videos oder das live Kamerabild verwendet wurden, um die den Effekt der Änderungen zu beurteilen. Zu Beginn mit dem unoptimierten Programm zeigte sich sehr deutlich, dass die Unterscheidung von Gesten, die eine Person die sich im Bild befindet zeigt, sehr unzuverlässig und eher zufällig einmal richtig war. Auch die Erkennung einer Person im Bild mit der Geste NOPOSE war nicht wirklich zuverlässig. Dies war allerdings auch zu erwarten, da an den Parameters seit dem Vorgängerprojekt nicht viel geändert wurde, sich aber die Situation und das neuronale Netz komplett geändert haben. Die meisten Anpassungen lassen sich über die Datei „vehicle.conf“ im Processing System umsetzen. Hier sind im Wesentlichen die Einstellungen für das Vorgehen beim Tiling (TILE_SPEC), die Größe der Eingangsframes (INITIAL_RESOLUTION_WIDTH und INITIAL_RESOLUTION_HEIGHT), sowie die Mindestsicherheit des neuronalen Netzen für jedes Tile (CONFIDENCE_THRESHOLD) und die Anzahl der korrekt erkannten Klassen (CORRECT_CLASS_THRESHOLD) von Bedeutung. Je kleiner die Eingangsbilder skaliert werden, desto schneller geht die Verarbeitung jedes Frames, aber desto mehr Informationen gehen verloren. Auch der Aufbau der Tiles, d.h. in welche Größen es gibt und wie viele Überlappende Tiles es geben soll hat wesentlichen Einfluss auf die Verarbeitungsgeschwindigkeit, aber auch auf die Zuverlässigkeit der Erkennung. Über die weiteren Parameter kann eingestellt werden, wie sicher sich das neuronale Netz bei der Erkennung eines Tiles sein muss und wie viele erkannte Tiles es geben muss, damit das Programm eine Pose als im Bild vorhanden einstuft.

Als guter Kompromiss für die Eingangsbildgröße hatte sich im Vorgängerprojekt schon 352x288 Pixel ergeben, weshalb dies meist so beibehalten wurde. Bei den Tiles ist es so, dass eine Person idealerweise ein Tile möglichst gut ausfüllt. Somit sind größere Tiles wichtig, wenn eine Geste nah am Fahrzeug erkannt werden soll und kleinere, sowie Zwischengrößen wenn sich die Person entfernt. Auch stärker überlappende Tiles sind von Vorteil, um eine Person mit einem Tile in möglichst perfektem Ausschnitt einzufangen. Bei dem Spiel mit diesen Werten zeigte sich, dass die Genauigkeit des neuronalen Netzes nicht gut genug ist um wirklich akzeptable Ergebnisse bei der Klassifizierung der Tiles zu erzielen. Viele Tiles führten zu deutlich weniger Bildern pro Sekunde bei der Verarbeitung aber auch nicht zu herausragend guten Ergebnissen. Gerade für unterschiedliche Konstellationen von Hintergrund, Person und Kleidung gab es keine Optimierung die bei allen einigermaßen gut funktioniert hat. Das neuronale Netz war sich oft nicht sicher genug, welche Pose es erkennt. Die Sicherheiten lagen oft nah beieinander. Es gab noch einige Versuche, z.B. die Erkennung auf wenige (also z.B. CORRECT_CLASS_THRESHOLD=1) aber dafür sehr

sichere (also z.B. CONFIDENCE_THRESHOLD>400) Tiles zu beschränken oder viele (z.B. CORRECT_CLASS_THRESHOLD>=10) aber dafür weniger sichere (also z.B. CONFIDENCE_THRESHOLD<300) Tiles zu nutzen. Aber auch dies ist in dem Rauschen und der Ungenauigkeit des neuronalen Netzes untergegangen. So stand nach einiger Zeit fest, dass in dieser Konstellation von Software im Processing System und Hardware im neuronalen Netz eine Erkennung der Pose in jeder Bildposition nicht zuverlässig erreichbar ist. Es gab dann noch einige Ideen, wie das Projektziel abgewandelt werden könnte, um zumindest eine gewisse Posenerkennung am Fahrzeug zu verwirklichen. Versucht wurde z.B. ein großes, bildfüllendes Tiles zu erzeugen und die Posenerkennung nur an diesem festzumachen, sodass eine Person die Pose direkt vor der Kamera ausführen muss. Aber auch hier wäre eine sehr exakte Positionierung nötig damit dies einigermaßen funktioniert und trotzdem wären die Schwankungen und Ungenauigkeiten sehr hoch gewesen.

Was dagegen unerwartet gut, wenn auch nicht ganz wie im Projektziel festgelegt, funktioniert hat, war die Erkennung der Position einer Person. Vorgesehen war eigentlich eine Person mit der Geste NOPOSE zu verfolgen. Dies war sehr schwer, da es keine Klasse für „Unklassifiziert“, also keine Person in Tile gibt und somit sehr viele Tiles als NOPOSE klassifiziert werden. Insbesondere die STOP Pose wurde aber vom neuronalen Netz in den Tiles um die Person herum mit einer relativ hohen Sicherheit klassifiziert. Dies war zwar fast unabhängig von der Pose, die die Person wirklich gezeigt hat, war aber deutlich höher als bei Tiles ohne Person. Seit dem Vorgängerprojekt verwendete die Software noch einen Mittelwert aus allen Tiles in denen eine Klasse erkannt wurde, um die wirkliche Position des Objektes zu bestimmen. Es zeigte sich, dass das Tile mit der höchsten Wahrscheinlichkeit für die STOP-Pose in Verbindung mit einem unteren Grenzwert für die Sicherheit, bessere und genauere Positionen erzeugte. An dieser Stelle wäre mit etwas mehr Entwicklungsaufwand vermutlich noch Potential, aber die ermittelten Positionen sind, wie im oben beschriebenen Beispielvideo gezeigt, sehr gut.

Geschrieben von Viktor Chechulin

Zusammengefasst kann das System in der aktuellen Version, selbst nach etlichen Optimierungen nicht eindeutig genug die Gesten von Personen, welche nicht exakt wie die Personen in den Testbildern positioniert sind voneinander unterscheiden. Zudem konnte ein Nebeneffekt beim Optimieren beobachtet werden, nämlich das Menschen an sich gut erkannt werden konnten und somit die Möglichkeit Personen zu verfolgen gegeben wäre, soweit diese nicht weiter als 1-2 Meter entfernt vor der Kamera stehen oder zur Verbesserung der Klassifizierung von NOPOSE zusätzlich die Klasse „Unklassifiziert“ zum neuronalen Netz hinzugefügt wird. Schließlich konnte aufgrund dessen die neue Fahrzeugsteuerung, welche auf Gesten ausgelegt ist, nicht optimal getestet werden.

12 Ergebnisse und Ausblick

Geschrieben von Tuncer Catalkaya

In diesem Kapitel werden zuerst die erreichten Ergebnisse in Kurzform aufgelistet und beschrieben. Danach wird ein Ausblick für die Nachfolgeprojekte gegeben. Der Ausblick besteht im Groben aus Ideen und Vorschlägen für eine Verbesserung des Gesamtsystems.

12.1 Ergebnisse

Geschrieben von Tobias Nießen

In diesem Projekt wurden die Grundlagen und verschiedenen Arten neuronaler Netze erarbeitet. Außerdem wurden verschiedene Möglichkeiten zur Implementierung dieser neuronalen Netze auf FPGAs aufgezeigt. Basierend auf dem so erworbenen Wissen wurde die vorherige Implementierung aus den Vorgängerprojekten, welche eher rudimentär, unübersichtlich und teils aufgebläht ist, überarbeitet.

Eins der Ziele war es, den Einstieg für Folgeprojekte und die weitere Arbeit an diesem Aufbau in Zukunft zu vereinfachen. Hierzu wurde die Fahrzeugsteuerung größtenteils neu geschrieben und in modulare Teile aufgeteilt, wobei auch ein großer Fokus auf der Dokumentation dieses Codes lag. Zu Anfang war ein physikalisches Board notwendig, um Programme mit diesem zu kompilieren und auszuführen. Dies ist nicht mehr der Fall, da ein Cross-Compiler aufgesetzt wurde, welcher ohne das PYNQ-Z1 das entworfene Programm ebenfalls testweise ausführen kann.

Zur Umsetzung des weiteren Ziels der Gestenerkennung wurden hier ebenfalls Änderungen an der Fahrzeugsteuerung und der Basisstation unternommen. Vor allem wurde ein eigener Datensatz generiert, aus welchem ein selbst erstelltes neuronales Netz trainiert wurde.

Diese Erkennung von Gesten funktioniert auf dem FPGA grundsätzlich, die Genauigkeit ist jedoch noch verbesserungswürdig. Im Vergleich zu dem Ausgangspunkt des Projektes, in welchem ein vorgefertigtes neuronales Netz benutzt wurde, welches ebenfalls leichtere Objekte erkennen musste, ist dies ein erheblicher Fortschritt. Es haben sich einige Verbesserungs- und Optimierungsmöglichkeiten ergeben, wie zum Beispiel eine Vorerkennung der Gebiete eines Bildes per CPU, welche jedoch aus Zeitgründen nicht implementiert werden konnten.

Das Fahrzeug fuhr im Ausgangsprojekt holprig. Es wurde versucht, die Ansteuerung der Motoren in Software anzupassen, sodass eine präzisere Steuerung möglich ist. Jedoch ist hier bei gewollter Verbesserung eine Änderung der Hardware nicht abzuwenden, da keine traditionelle Lenkung vorhanden ist. Die Hardware wurde in diesem Projekt nicht verändert.

Im Laufe des Projektes sind mehrere ausführliche Anleitungen entstanden, mittels welcher zukünftige Projekte in der Lage sind, das hier Entworfenen nachzuvollziehen, selbst zu installieren und abzuändern. Insgesamt wurde das Ziel des einfacheren Einstiegs in das Thema neuronale Netze und FPGA Programmierung für zukünftige Projekte definitiv erreicht. Ebenfalls ist die Software signifikant einfacher nachzuvollziehen und durch eine gute Dokumentation einfacher abzuändern. Durch diese Arbeit lassen sich die Grundlagen der neuronalen Netze nachvollziehen, sodass ein Weiterarbeiten an der hier entworfenen Gestenerkennung gut möglich ist.

12.2 Ausblick

Geschrieben von Viktor Chechulin, Tuncer Catalkaya, Felix Müller, Tobias Nießen, Lukas Schimanski

Es folgt eine Liste mit Vorschlägen und Ideen, mit derer man eine Verbesserung des derzeitigen Gesamtsystems erhofft:

- Um eine eindeutigere Erkennung der Posen START, STOP und NOPOSE zu erzielen, könnte es hilfreich sein die Klasse „Unklassifiziert“ in das neuronale Netz aufzunehmen, um zu verhindern, dass die Umgebung als NOPOSE, START oder STOP klassifiziert wird.
- Die vierte Klasse, also „Kein Mensch“, „Keine Person“ oder auch „Unklassifiziert“ (Unclassified) könnte mithilfe eines weiteren neuronalen Netzes umgesetzt werden. Hierzu könnte für die Klassifizierung, ob es sich um eine Person handelt, ein CNN verwendet werden. Im Anschluss wird dieses CNN an das vorhandene Netz, mit den drei Klassen „START“, „STOP“ und „NOPOSE“, angeknüpft.
- Um das Tiling zu verbessern, wäre eine Idee das Verfahren NMS (Non Maximum Suppression) zu implementieren, sodass Tiles die Posen wie START und STOP erkennen, zu einem großen Tile zusammengefasst werden und somit die Klassifizierung erleichtert und gleichermaßen verbessert wird.
- Auf der CPU könnte ebenfalls ein neuronales Netz verwendet werden, welches eine Vorerkennung der Area of Interest verwirklicht.
- Die Hardware-Seite der Fahrzeugsteuerung wurde nicht verändert. Die vorgeschlagenen Verbesserungen zu Anfang des Projektes sind immer noch relevant, die Motoren lassen sich vor allem in der Drehung nicht sehr fein steuern.
- Der Quellcode ist zwar durch die Controller-Hierarchie besser geordnet als zuvor, jedoch könnte die Softwarearchitektur noch weiter ausgebaut werden. Hierzu könnten z.B. die SOLID-Prinzipien angewendet werden, um so „Clean Code“ zu erzeugen und damit zukünftig den Quellcode besser Erweiterbar und Wartbar zu machen.
- Die Base Station könnte als Webserver implementiert werden. Dazu kann das Web Framework [SMb] herangezogen werden. Auf der Seite gibt es ein Beispiel für das Anzeigen eines Livestreams, siehe [SMa] für nähere Informationen zu HLS-Media-Stream (HLS = HTTP-Livestreaming).
- Lokalisierung eines Objektes mittels eines RPN oder SSD anstatt von Tilingüberlappung, um die in Kapitel 6 beschriebenen Konfigurationsprobleme zu vermeiden.
- Zur Erhöhung der Bildwiederholrate könnte das Tiling noch weiter optimiert werden. Beispielsweise könnte sich mehr auf eine horizontale Positionierung konzentriert werden, da die vertikale Achse für das verwendete Fahrzeug wenig interessant ist.
- Zur Erhöhung der Genauigkeit könnte versucht werden die Bildeingangsgröße des Netzes zu erhöhen. In der Evaluation zum Ressourcenverbrauch hat sich gezeigt, dass noch weiterer Platz auf dem FPGA ist, sodass eine Erhöhung der Genauigkeit wahrscheinlich keine Auswirkung auf die Geschwindigkeit hätte.
- Übersetzung der Fahrzeugsteuerung für das PYNQ-Z1 per Crosscompiler auf x86 ermöglichen. Zur testweisen Ausführung ist es bereits möglich die Fahrzeugsteuerung auf x86 auszuführen allerdings muss die Anwendung, die auf dem Fahrzeug läuft, auf diesem übersetzt werden, was einige Minuten benötigt.
- Derzeit wird der Quellcode nicht getestet. Hierzu könnte sich über ein Test Framework für C++ informiert werden. Ein Beispiel für ein Test Framework wäre „GoogleTest“.

12.2.1 Zukünftige Testszenarien

Geschrieben von Philipp Altnickel

Folgender Abschnitt zeigt einige Tests, die hätten durchgeführt werden sollen, wenn die Fahrzeugsteuerung wie geplant funktioniert hätte. Sie geben einen guten Überblick über das geplante Verhalten und kann möglicherweise in ähnlicher Form nach einer Weiterentwicklung durchgeführt werden.

Um sicherzustellen, dass alle Änderungen am Gesamtsystem wie gewünscht zusammen funktionieren, können einige Tests durchgeführt werden. Konkret sollten dabei die wichtigsten und größten Änderungen auf ihre korrekte Funktionalität im Gesamtsystem geprüft werden.

Test der Einbindung des eigenen neuronalen Netzes Ziel des ersten Tests ist die Prüfung, dass die Erfassung der Bilddaten, sowie die Verarbeitung im neuronalen Netz und anschließende Klassifizierung und Bestimmung der Position im Bild korrekt funktioniert. Hierzu wird das System, d.h. das Fahrzeug und die Base Station gestartet. Anschließend hat sich eine Person vor der Kamera des Fahrzeugs positioniert. Diese Person sollte dann folgende Positionen und Gesten einnehmen:

- Es befindet sich keine Person im Bild.
- Die Person füllt nahezu die volle Bildhöhe aus. Sie steht dabei einmal mittig im Bild, einmal sehr nah am linken und einmal sehr nah am rechten Rand. In jeder Position werden die Gesten START, STOP und NOPOSE getestet.
- Die Person füllt etwa die Hälfte der Bildhöhe aus. Auch hier werden alle Kombinationen aus mittig, sehr weit links und sehr weit rechts am Rand mit allen drei Posen getestet.
- Die Person steht sehr weit weg von dem Fahrzeug, die füllt etwa ein Viertel der Bildhöhe aus. Hier werden ebenfalls die Mitte und Randbereich mit allen Posen getestet.

Für jede Kombination wird anhand der Darstellung der Base Station die korrekte Erkennung der Position, sowie Geste geprüft. Die erfolgreiche Ausführung dieser Tests zeigt, dass sowohl die Vorverarbeitung auf dem Processing System mit Aufnahmen des Bildes und Tiling, als auch die Kommunikation und Verarbeitung auf der Programmable Logic und auch die anschließende Verarbeitung der Ergebnisse funktionieren.

Test der Zustände der Fahrzeugsteuerung Ziel des nächsten Tests ist die Verifizierung der korrekten Funktion des Zustandsautomaten für die Fahrzeugsteuerung. Nachdem bereits geprüft wurde, dass das Fahrzeug die Gesten einer Person erkennen kann, solle nun gezeigt werden, dass diese auch zum gewünschten Zustandsübergang führen. Da der Zustandsautomat (Abb. 44) in seiner Größe überschaubar ist, ist eine vollständige Zustandsabdeckung abprüfbar. Außerdem können alle für das wesentliche und neue Verhalten des Fahrzeugs wichtigen Übergänge geprüft werden. Gleichzeitig kann in diesem Test auch das korrekte Verhalten des Fahrzeugs in jedem Zustand geprüft werden.

Folgende Schritte werden der Reihe nach durchgeführt und geprüft:

- Zum Start befindet sich das Fahrzeug immer im MANUAL.MODE. Das Fahrzeug lässt sich manuell über den Rechner, auf dem die Base Station läuft steuern. Es folgt dabei der gewünschten Aktion der jeweiligen Taste.
- Schaltet man an der Base Station in den Automatik-Modus, wird das Fahrzeug in den SEARCH-MODE gehen. Das Fahrzeug dreht sich nun um seine eigene Achse.
- Es stellt sich eine Person vor das Fahrzeug. Keine und die Stopp-Geste ändern nichts am Zustand und Verhalten.
- Wird die Start-Geste gezeigt, wechselt das Fahrzeug in den APPROACH.MODE. Es fährt auf die Person zu.
- Bewegt sich die Person schnell aus dem Bild, sodass das Fahrzeug ihr so schnell nicht folgen kann,

wechselt es in den APPROACH_SEARCH_MODE. Es dreht sich nun erneut um seine eigene Achse.

- Die Person bleibt stehen. Das Fahrzeug sollte sie wiederfinden, in den APPROACH_MODE wechseln und erneut auf sie zufahren.
- Zeigt die Person nun die Stopp-Geste, wird der Zustand zurück zum SEARCH_MODE geändert, das Fahrzeug fängt an, sich um seine eigene Achse zu drehen.

12.2.2 Weitere Ideen für Zustandsdiagramme für die Fahrzeugsteuerung

Geschrieben von Tuncer Catalkaya

Bei der Konzeption und Planung eines Zustandsdiagramms für die Fahrzeugsteuerung sind zwei weitere alternative Ideen entstanden. Es wird nur die grobe Idee ergänzend zu Kapitel 9.2.8 beschrieben. Diese beiden Diagramme sind generell aufwändiger als das umgesetzte und eigentlich geplante Zustandsdiagramm (vgl. Abb. 43 und 44). Die beiden Zustandsdiagramme sollen Denkanstöße geben für eine Erweiterung des bereits implementierten Zustandsdiagramms.

Das Zustandsdiagramm in Abbildung 49 erweitert grundsätzlich die Auswahl der Geste, also den ersten Bereich, der zuvor nur aus dem SEARCH_MODE bestand. Nun kommt ein REQUEST_MODE hinzu. Dieser neue Zustand besagt, dass das Vehikel bei einer erkannten Person (NoPose Geste) sich auf diese Person „fokussieren“ soll. Das Vehikel dreht sich also nicht mehr, sondern bleibt still, versucht aber die Person zu verfolgen und gut im Bild zu halten, um so auf eine Geste zu horchen. Sollte keine Person mehr erkannt werden, dann wird wieder zurück in den SEARCH_MODE gewechselt. Wobei dies von einer Minimum Anzahl an Versuchen eine Person zu erkennen abhängt. Zusammengefasst wird also nicht mehr versucht sofort eine Geste zu erkennen, sondern es wird erst versucht überhaupt eine Person zu erkennen um dann von einer Person eine Geste zu erkennen.

Das Zustandsdiagramm in Abbildung 50 erweitert grundsätzlich den Tracking Mode, also den bisherigen ständigen Wechsel zwischen APPROACH_MODE und APPROACH_SEARCH_MODE. Zu diesem Tracking Mode kommt der Zustand STOP_MODE hinzu. Dieser neue Zustand besagt, dass das Vehikel bei einer Stop Geste, während des Tracking Mode, in Stillstand gehen soll. Das Vehikel dreht sich also nicht mehr, sondern bleibt still, versucht aber die Person zu verfolgen und gut im Bild zu halten, um so optimal eine Geste erkennen zu können. Also das Fahrzeug bleibt still und dreht sich nicht. Sollte im STOP_MODE erneut eine Start Geste erkannt werden, so versucht das Fahrzeug erneut einer Person zu folgen. Falls im STOP_MODE keine Person erkannt wird, dann geht das Fahrzeug zurück in den SEARCH_MODE. Wobei dies von einer Minimum Anzahl an Versuchen eine Person zu erkennen abhängt.

Der REQUEST_MODE und der STOP_MODE sind identisch. Somit könnte, wenn der Ablauf dieses Zustands implementiert wurde, sogar eine Kombinationen beider Zustandsdiagramme umgesetzt werden (vgl. Abb. 49 und 50).

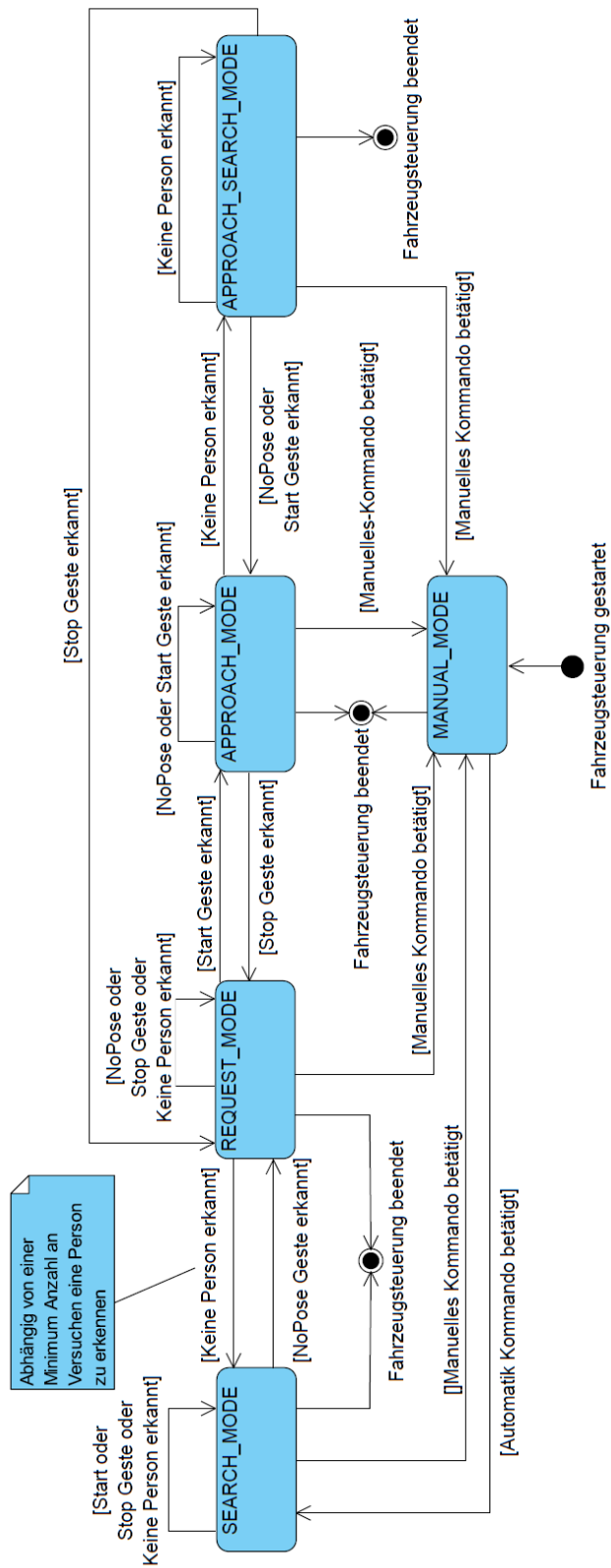


Abbildung 49: Zustandsdiagramm der Fahrzeugsteuerung - Erweiterung der Gesten Auswahl

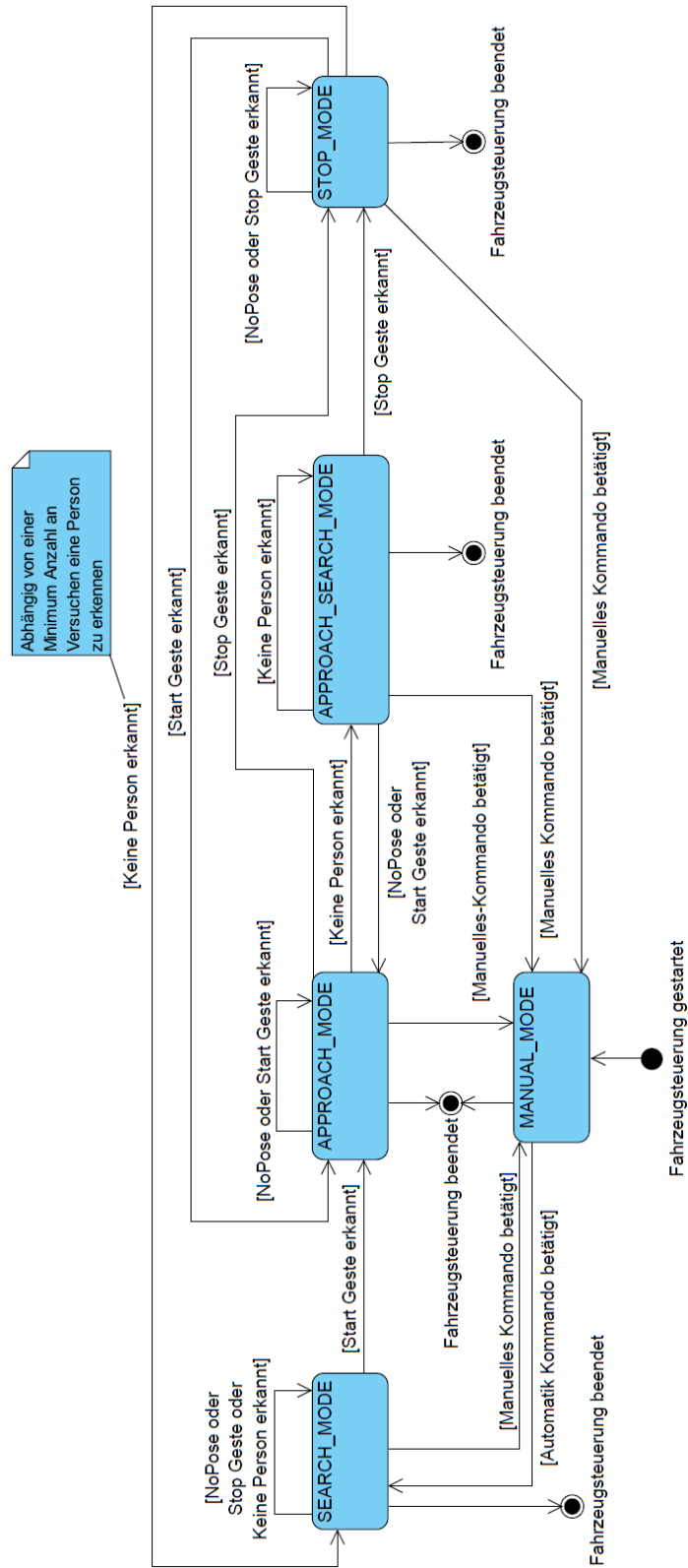


Abbildung 50: Zustandsdiagramm der Fahrzeugsteuerung - Erweiterung des Tracking Modus

Literatur

- [Agg18] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer International Publishing AG, 2018, S. 315–330. ISBN: 978-3-319-94462-3.
- [AH20] Bishwo Adhikari und Heikki Huttunen. „Iterative Bounding Box Annotation for Object Detection“. In: Juli 2020.
- [ARM10] ARM. *AMBA Specifications*. 2010. URL: <https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications> (besucht am 16. 08. 2021).
- [Avn21a] Avnet. *MicroZed | Avnet Boards*. 2021. URL: <https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/microzed/> (besucht am 29. 08. 2021).
- [Avn21b] Avnet. *Ultra96-V2 Board*. 2021. URL: <https://www.avnet.com/wps/portal/us/products/new-product-introductions/npi/aes-ultra96-v2/> (besucht am 23. 08. 2021).
- [Blo+18] Michaela Blott u. a. „FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks“. en. In: *arXiv:1809.04570 [cs]* (Sep. 2018). Comment: to be published in ACM TRETS Special Edition on Deep Learning. arXiv: 1809.04570 [cs].
- [Bra+19] Markus Braun u. a. „EuroCity Persons: A Novel Benchmark for Person Detection in Traffic Scenes“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019), S. 1–1. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2019.2897684.
- [Chi] Sasank Chilamkurthy. *Writing Custom Datasets, DataLoaders and Transforms*. URL: https://pytorch.org/tutorials/beginner/data_loading_tutorial.html (besucht am 2021-05-29).
- [Con] Torch Contributors. *SOURCE CODE FOR TORCHVISION.DATASETS.FOLDER*. URL: https://pytorch.org/vision/stable/_modules/torchvision/datasets/folder.html#ImageFolder (besucht am 2021-06-22).
- [Cou+16] Matthieu Courbariaux u. a. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG].
- [Dat] COCO Dataset. *COCO Dataset*. URL: <https://cocodataset.org/#home> (besucht am 27. 08. 2021).
- [Dev21] LinuxTV Developers. *(Online) Linux Kernel Media Documentation — The Linux Kernel Documentation*. 2021. URL: <https://linuxtv.org/downloads/v4l-dvb-apis/> (besucht am 29. 06. 2021).
- [Dig21a] Digilent. *Anvyl Spartan-6 FPGA Trainer Board*. en. 2021. URL: <https://digilent.com/shop/anvyl-spartan-6-fpga-trainer-board/> (besucht am 29. 08. 2021).
- [Dig21b] Digilent. *Arty A7: Artix-7 FPGA Development Board*. en. 2021. URL: <https://digilent.com/shop/arty-a7-artix-7-fpga-development-board/> (besucht am 29. 08. 2021).
- [Dig21c] Digilent. *Arty S7: Spartan-7 FPGA Development Board*. en. 2021. URL: <https://digilent.com/shop/arty-s7-spartan-7-fpga-development-board/> (besucht am 29. 08. 2021).
- [Dig21d] Digilent. *Basys 3 Artix-7 FPGA Trainer Board: Recommended for Introductory Users*. en. 2021. URL: <https://digilent.com/shop/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/> (besucht am 29. 08. 2021).
- [Dig21e] Digilent. *Nexys 4 Artix-7 FPGA Trainer Board*. en. 2021. URL: <https://digilent.com/shop/nexys-4-artix-7-fpga-trainer-board/> (besucht am 29. 08. 2021).
- [Dig21f] Digilent. *Nexys A7: FPGA Trainer Board Recommended for ECE Curriculum*. en. 2021. URL: <https://digilent.com/shop/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/> (besucht am 29. 08. 2021).

- [Dig21g] Digilent. *Nexys Video Artix-7 FPGA: Trainer Board for Multimedia Applications*. en. 2021. URL: <https://digilent.com/shop/nexys-video-artix-7-fpga-trainer-board-for-multimedia-applications/> (besucht am 29.08.2021).
- [Dig21h] Digilent. *PYNQ-Z1 - Digilent Reference*. 2021. URL: <https://digilent.com/reference/programmable-logic/pynq-z1/start> (besucht am 23.08.2021).
- [Dig21i] Digilent. *ZedBoard Zynq-7000 ARM/FPGA SoC Development Board*. en. 2021. URL: <https://digilent.com/shop/zedboard-zynq-7000-arm-fpga-soc-development-board/> (besucht am 29.08.2021).
- [Dig21j] Digilent. *Zybo Z7 - Digilent Reference*. 2021. URL: <https://digilent.com/reference/programmable-logic/zybo-z7/start?redirect=1> (besucht am 29.08.2021).
- [DW18] Bahar Salehpour Daniel H. Noronha und Steven J.E. Wilton. „LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks“. In: (2018). URL: <https://ieeexplore.ieee.org/abstract/document/8470462>.
- [EGW] Mark Everingham, Luc van Gool und Chris Williams. *The PASCAL Visual Object Classes Challenge 2007*. URL: <http://host.robots.ox.ac.uk/pascal/VOC/> (besucht am 27.08.2021).
- [Ele] Inc. Elegoo. *Elegoo Smart Robot Car Kit V3.0.2019.10.12.zip*. URL: <https://www.elegoo.com/pages/arduino-kits-support-files> (besucht am 02.08.2021).
- [ESA19] ESA. *The radiation showstopper for Mars exploration*. en. Mai 2019. URL: https://www.esa.int/Science_Exploration/Human_and_Robotic_Exploration/The_radiation_showstopper_for_Mars_exploration (besucht am 23.08.2021).
- [ESC20] Ermal Elbasani, Pattamaset Siriporn und Jae Sung Choi. „A Survey on RFID in Industry 4.0“. en. In: *Internet of Things for Industry 4.0: Design, Challenges and Solutions*. Hrsg. von G. R. Kanagachidambaresan u. a. EAI/Springer Innovations in Communication and Computing. Cham: Springer International Publishing, 2020, S. 1–16. ISBN: 978-3-030-32530-5. DOI: 10.1007/978-3-030-32530-5_1. URL: https://doi.org/10.1007/978-3-030-32530-5_1 (besucht am 29.08.2021).
- [FINa] FINN. *End-to-End FINN Flow for a Simple Convolutional Net*. URL: https://github.com/Xilinx/finn/blob/master/notebooks/end2end_example/bnn-pynq/cnv_end2end_example.ipynb (besucht am 23.06.2021).
- [FINb] FINN. *Transformation - Streamline*. URL: https://finn.readthedocs.io/en/latest/source_code/finn.transformation.streamline.html (besucht am 06.07.2021).
- [Fou21] The Linux Foundation. *ONNX - Website*. 2021. URL: <https://onnx.ai/> (besucht am 27.05.2021).
- [Fur21] Antonio Furioso. 2021. URL: <https://medium.com/analytics-vidhya/convolutional-1-neural-network-22a9649cee8>.
- [Gir+14] Ross Girshick u. a. „Rich feature hierarchies for accurate object detection and semantic segmentation“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, S. 580–587.
- [Gir15] Ross Girshick. „Fast r-cnn“. In: *Proceedings of the IEEE international conference on computer vision*. 2015, S. 1440–1448.
- [Goo] Google. URL: <https://colab.research.google.com/signup#>.
- [GS20] Alireza Ghaffari und Yvon Savaria. „CNN2Gate: An Implementation of Convolutional Neural Networks Inference on FPGAs with Automated Design Space Exploration“. en. In: *Electronics* 9.12 (Dez. 2020). Number: 12 Publisher: Multidisciplinary Digital Publishing Institute, S. 2200. DOI: 10.3390/electronics9122200. URL: <https://www.mdpi.com/2079-9292/9/12/2200> (besucht am 25.08.2021).
- [Gun+98] Steve R Gunn u. a. „Support vector machines for classification and regression“. In: *ISIS technical report* 14.1 (1998), S. 5–16.

- [Gün08] Karlf.Wender Günter Daniel Rey. *Neuronale Netze Eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*. 2008. ISBN: 978-3-456-84513-5.
- [Guo+18] Peng Guo u. a. „FBNA: A Fully Binarized Neural Network Accelerator“. In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, S. 51–513. DOI: 10.1109/FPL.2018.00016.
- [He+16] Kaiming He u. a. „Deep residual learning for image recognition“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, S. 770–778.
- [He+17] Kaiming He u. a. „Mask r-cnn“. In: *Proceedings of the IEEE international conference on computer vision*. 2017, S. 2961–2969.
- [Hei17] Sebastian Heinz. *Einführung TensorFlow*. 2017. URL: <https://www.statworx.com/de/blog/data-science/einfuehrung-tensorflow/>.
- [Hou+18] Xiaoying Hou u. a. „Vehicle License Plate Recognition System Based on Deep Learning Deployed to PYNQ“. In: *2018 18th International Symposium on Communications and Information Technologies (ISCIT)*. Sep. 2018, S. 79–84. DOI: 10.1109/ISCIT.2018.8587934.
- [HQ17] Yufeng Hao und Steven Quigley. „The implementation of a Deep Recurrent Neural Network Language Model on a Xilinx FPGA“. In: *arXiv:1710.10296 [cs]* (Nov. 2017). arXiv: 1710.10296. URL: <http://arxiv.org/abs/1710.10296> (besucht am 25.08.2021).
- [Hut+20] Colin von Huth u. a. *Bericht zum Projekt ‚Neuronale Netze auf strahlungstoleranten FPGAs für die Raumfahrt‘*. Techn. Ber. Hochschule Bremen, 14. Feb. 2020. 88 S. URL: <http://homepages.hs-bremen.de/~jbrederede/de/forschung/veroeffentlichungen/neuronale-netze-fpgas-projekt-1920.html> (besucht am 21.06.2021).
- [ICY21] Jirarat leamsaard, Surapon Nathanael Charoensook und Suchart Yammen. „Deep Learning-based Face Mask Detection Using YoloV5“. In: *2021 9th International Electrical Engineering Congress (iEECON)*. März 2021, S. 428–431. DOI: 10.1109/iEECON51072.2021.9440346.
- [Inf14] Max Planck Institut Informatik. *MPII Human Pose Dataset*. 2014. URL: <http://human-pose.mpi-inf.mpg.de/>.
- [Int21] Intel. *DK-DEV-4CGX150N Intel / Altera | Mouser*. de-de. 2021. URL: <https://www.mouser.de/ProductDetail/989-DK-DEV-4CGX150N> (besucht am 29.08.2021).
- [IT-19] Ancud IT-Beratung. 2019. URL: <https://www.ancud.de/neuronale-netze-mit-python-und-tensorflow/>.
- [Iva19] Ivanovs. *Implementation analysis of convolutional neural networks on FPGAs*. nl. 2019. URL: <https://research.tue.nl/nl/studentTheses/implementation-analysis-of-convolutional-neural-networks-on-fpgas> (besucht am 25.08.2021).
- [Joc] Glenn Jocher. *YOLOV5*. URL: <https://github.com/ultralytics/yolov5> (besucht am 29.08.2021).
- [KH+09] Alex Krizhevsky, Geoffrey Hinton u. a. „Learning multiple layers of features from tiny images“. In: (2009).
- [Kim+12] Kyekyung Kim u. a. „Object recognition for cell manufacturing system“. In: *2012 9th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*. Nov. 2012, S. 512–514. DOI: 10.1109/URAI.2012.6463056.
- [KSH12] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. „Imagenet classification with deep convolutional neural networks“. In: *Advances in neural information processing systems 25* (2012), S. 1097–1105.
- [LeC+98] Yann LeCun u. a. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (1998), S. 2278–2324.
- [Lem+19] César B. Lemos u. a. „Convolutional Neural Network Based Object Detection for Additive Manufacturing“. In: *2019 19th International Conference on Advanced Robotics (ICAR)*. Dez. 2019, S. 420–425. DOI: 10.1109/ICAR46387.2019.8981618.

- [Leo18] Andrea Leopardi. *Convolutional Neural Network Quantisation for Accelerating Inference in Visual Embedded Systems*. eng. 2018. URL: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-240404> (besucht am 25. 08. 2021).
- [Liu+16] Wei Liu u. a. „SSD: Single Shot MultiBox Detector“. In: *Computer Vision – ECCV 2016*. Hrsg. von Bastian Leibe u. a. Cham: Springer International Publishing, 2016, S. 21–37. ISBN: 978-3-319-46448-0.
- [Lu+17] Liqiang Lu u. a. „Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs“. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2017, S. 101–108. DOI: 10.1109/FCCM.2017.64.
- [Lu+19] Liqiang Lu u. a. „An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs“. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. ISSN: 2576-2621. Apr. 2019, S. 17–25. DOI: 10.1109/FCCM.2019.00013.
- [Mar+21] Amna Maraoui u. a. „PYNQ FPGA Hardware implementation of LeNet-5-Based Traffic Sign Recognition Application“. In: *2021 18th International Multi-Conference on Systems, Signals Devices (SSD)*. ISSN: 2474-0446. März 2021, S. 1004–1009. DOI: 10.1109/SSD52085.2021.9429480.
- [Mic] Microsoft. URL: <https://azure.microsoft.com/de-de/pricing/calculator/>.
- [Mic20a] Micron Technology. *Doubling I/O Performance with PAM4*. 2020. URL: https://media-www.micron.com/-/media/client/global/documents/products/technical-marketing-brief/gddr6x_pam4_2x_speed_tech_brief (besucht am 23. 08. 2021).
- [Mic20b] Microsoft. 2020. URL: <https://docs.microsoft.com/en-us/azure/virtual-machines/nc-series>.
- [Mir21] Sebastian Raschka / Vahid Mirjalili. *Machine Learning mit Python und Keras, TensorFlow 2 und Scikit-learn - Das umfassende Praxis-Handbuch für Data Science, Deep Learning und Predictive Analytics*. Heidelberg: MITP-Verlags GmbH & Co. KG, 2021. ISBN: 978-3-747-50215-0.
- [Mis+06] Mahim Mishra u. a. „Tartan: Evaluating spatial computation for whole program execution“. In: Bd. 41. Nov. 2006, S. 163–174. DOI: 10.1145/1168857.1168878.
- [moh21] mohdumar644. *TinyYOLO-BNN*. original-date: 2019-04-05T04:15:49Z. Mai 2021. URL: <https://github.com/mohdumar644/TinyYOLO-BNN> (besucht am 25. 08. 2021).
- [Mül+21a] Felix Müller u. a. *Applying Binarized Neural Networks on FPGAs to an Autonomous Driving Problem*. Englisch. Techn. Ber. Hochschule Bremen, 31. März 2021. 50 S. URL: <http://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/bnns-on-fpgas-driving-projekt-2021.html> (besucht am 21. 06. 2021).
- [Mül+21b] Felix Müller u. a. *Gestenvideos*. 2021. URL: <https://gitlab.com/soc-nn/vehicleps/-/tree/master/res/videos> (besucht am 29. 08. 2021).
- [Ngu+19] Anh-Duc Nguyen u. a. „Distribution Padding in Convolutional Neural Networks“. In: *2019 IEEE International Conference on Image Processing (ICIP)*. 2019, S. 4275–4279. DOI: 10.1109/ICIP.2019.8803537.
- [Num21] Numato Lab. *Mimas A7 – Artix 7 FPGA Development Board*. en-US. 2021. URL: <https://numato.com/product/mimas-a7-artix-7-fpga-development-board/> (besucht am 29. 08. 2021).
- [Papa] Alessandro Pappalardo. *CNV model*. Version 0.2.0a0. URL: https://github.com/Xilinx/brevitas/blob/aff49758ec445d77c75721c7de3091a2a1797ca8/brevitas_examples/bnn_pynq/models/CNV.py (besucht am 2021-07-04).
- [Papb] Alessandro Pappalardo. *Xilinx/brevitas*. Version latest. DOI: 10.5281/zenodo.3333552. URL: <https://doi.org/10.5281/zenodo.3333552> (besucht am 2021-07-04).

- [Papc] Alessandro Pappalardo. *Xilinx/brevitas*. Version 0.2.0a0. URL: <https://github.com/Xilinx/brevitas/tree/aff49758ec445d77c75721c7de3091a2a1797ca8> (besucht am 2021-07-04).
- [Par21] Visual Paradigm. *What is State Machine Diagram?* 2021. URL: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-state-machine-diagram/> (besucht am 27.08.2021).
- [PKK21] Dominika Przewlocka-Rus, Marcin Kowalczyk und Tomasz Kryjak. *Exploration of Hardware Acceleration Methods for an XNOR Traffic Signs Classifier*. 2021. arXiv: 2104.02303 [cs.CV].
- [PMB13] Razvan Pascanu, Tomas Mikolov und Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. 2013. arXiv: 1211.5063 [cs.LG].
- [Pro80] User Datagram Protocol. „Rfc 768 j. postel isi 28 august 1980“. In: *Isi* (1980).
- [Qin+20] Haotong Qin u. a. „Binary neural networks: A survey“. In: *Pattern Recognition* 105 (Sep. 2020), S. 107281. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2020.107281. URL: <http://dx.doi.org/10.1016/j.patcog.2020.107281>.
- [Ras+16] Mohammad Rastegari u. a. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*. 2016. arXiv: 1603.05279 [cs.CV].
- [Red+16] Joseph Redmon u. a. „You Only Look Once: Unified, Real-Time Object Detection“. In: IEEE, 2016. ISBN: 978-1-4673-8852-8. DOI: 10.1109/CVPR.2016.91. URL: <https://pjreddie.com/media/files/papers/yolo.pdf>.
- [Red16] Joseph Redmon. *YOLO: Real Time Object Detection*. 2016. URL: <https://github.com/pjreddie/darknet/wiki/YOLO:-Real-Time-Object-Detection> (besucht am 28.08.2021).
- [Ren+16] Shaoqing Ren u. a. „Faster R-CNN: towards real-time object detection with region proposal networks“. In: *IEEE transactions on pattern analysis and machine intelligence* 39.6 (2016), S. 1137–1149.
- [Rez+19] Hamid Rezatofighi u. a. „Generalized Intersection over Union“. In: (Juni 2019).
- [RF17] Joseph Redmon und Ali Farhadi. „YOLO9000: Better, Faster, Stronger“. In: IEEE, Juli 2017. ISBN: 978-1-5386-0458-8. DOI: 10.1109/CVPR.2017.690. URL: <https://ieeexplore.ieee.org/document/8100173>.
- [RF18] Joseph Redmon und Ali Farhadi. „YOLOv3: An Incremental Improvement“. In: Apr. 2018. URL: <https://arxiv.org/abs/1804.02767v1>.
- [RGV14] Rasmus Rothe, Matthieu Guillaumin und Luc Van Gool. „Non-maximum suppression for object detection by passing messages between windows“. In: *Asian conference on computer vision*. Springer. 2014, S. 290–306.
- [Ryb21] Vladimir Rybalkin. *LSTM-PYNQ Pip Installable Package*. original-date: 2018-02-21T10:28:44Z. Apr. 2021. URL: <https://github.com/tuk1-msd/LSTM-PYNQ> (besucht am 25.08.2021).
- [Sai20] Yahia Said. „Pynq-YOLO-Net: An embedded quantized convolutional neural network for face mask detection in COVID-19 pandemic era“. In: (2020). DOI: 10.14569/ijacsa.2020.0110912.
- [Sal+20] Alaa M. Salman u. a. „Comparative Study of Hardware Accelerated Convolution Neural Network on PYNQ Board“. In: *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*. Okt. 2020, S. 578–582. DOI: 10.1109/NILES50944.2020.9257899.
- [Sha+16] H. Sharma u. a. In: *From High-Level Deep Neural Models to FPGAs*. 2016.
- [Sim18] Tara Sims. *Xilinx Announces the Acquisition of DeePhi Tech*. en. 2018. URL: <https://www.xilinx.com/news/press/2018/xilinx-announces-the-acquisition-of-deephi-tech.html> (besucht am 25.08.2021).

- [SL19] Taylor Simons und Dah-Jye Lee. „A Review of Binarized Neural Networks“. In: *Electronics* 8.6 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8060661. URL: <https://www.mdpi.com/2079-9292/8/6/661>.
- [SMa] Leonid Stryzhevskiy und Benedikt-Alexander Mokroß. *Example-HLS-Media-Stream*. URL: <https://oatpp.io/examples/hls-media-stream/> (besucht am 29.08.2021).
- [SMb] Leonid Stryzhevskiy und Benedikt-Alexander Mokroß. *Oat++ An Open Source C++ Web Framework*. URL: <https://oatpp.io/> (besucht am 29.08.2021).
- [SSS18] Luca Stornaiuolo, Marco Santambrogio und Donatella Sciuto. „On How to Efficiently Implement Deep Learning Algorithms on PYNQ Platform“. In: *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. ISSN: 2159-3477. Juli 2018, S. 587–590. DOI: 10.1109/ISVLSI.2018.00112.
- [Sub] Vineeth S Subramanyam. *IOU (Intersection over Union)*. URL: <https://medium.com/analytics-vidhya/iou-intersection-over-union-705a39e7acef> (besucht am 28.08.2021).
- [SZ14] Karen Simonyan und Andrew Zisserman. „Very deep convolutional networks for large-scale image recognition“. In: *arXiv preprint arXiv:1409.1556* (2014).
- [Sze+15] Christian Szegedy u. a. „Going deeper with convolutions“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, S. 1–9.
- [Tea21] TensorFlow Team. *Convolutional Neural Network (CNN)*. 19. März 2021. URL: <https://www.tensorflow.org/tutorials/images/cnn> (besucht am 24.04.2021).
- [THW17] Wei Tang, Gang Hua und Liang Wang. „How to Train a Compact Binary Neural Network with High Accuracy?“. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI'17. San Francisco, California, USA: AAAI Press, 2017, S. 2625–2631.
- [TSM21] TSMC. *5nm Technology - Taiwan Semiconductor Manufacturing Company Limited*. en. 2021. URL: https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_5nm (besucht am 23.08.2021).
- [Uij+13] Jasper RR Uijlings u. a. „Selective search for object recognition“. In: *International journal of computer vision* 104.2 (2013), S. 154–171.
- [Umu+16] Yaman Umuroglu u. a. „FINN: A Framework for Fast, Scalable Binarized Neural Network Inference“. In: *CoRR* abs/1612.07119 (2016). arXiv: 1612.07119. URL: <http://arxiv.org/abs/1612.07119>.
- [Uni07] Fudan University University of Pennsylvania. *Penn-Fudan Database for Pedestrian Detection and Segmentation*. 2007. URL: https://www.cis.upenn.edu/~jshi/ped_html/.
- [Voi19] Mario Voithofer. „Verwendung von Deep Learning für die Erkennung von Werbung am Straßenrand in Dashcam-Videos“. Magisterarb. Fachhochschul-Masterstudiengang Interactive Media in Hagenberg, Juni 2019. URL: <https://theses.fh-hagenberg.at/system/files/pdf/Voithofer19.pdf>.
- [Wan+19] Jiaqi Wang u. a. „Region proposal by guided anchoring“. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, S. 2965–2974.
- [WDC18a] E. Wang, J. J. Davis und P. Y. K. Cheung. „A PYNQ-based Framework for Rapid CNN Prototyping“. In: *IEEE Symposium on Field-programmable Custom Computing Machines (FCCM)*. 2018.
- [WDC18b] Erwei Wang, James J. Davis und Peter Y. K. Cheung. „A PYNQ-Based Framework for Rapid CNN Prototyping“. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. ISSN: 2576-2621. Apr. 2018, S. 223–223. DOI: 10.1109/FCCM.2018.00057.
- [Wic+21] Reena Wichmann u. a. *Evaluationstests*. 2021. URL: https://gitlab.com/soc-nn/soc-nn_main_repo/-/tree/master/src/training/exports/evaluation (besucht am 31.08.2021).

- [WS20] P. Warden und D. Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O'Reilly Media, Incorporated, 2020. ISBN: 9781492052043. URL: <https://books.google.de/books?id=sB3mxQEACAAJ>.
- [Xil] Xilinx. *FINN Documentation*. URL: <https://finn.readthedocs.io/> (besucht am 30.05.2021).
- [XIL11] XILINX. *AXI Reference Guide*. User Guide. UG 761. XILINX. 2011. URL: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.
- [Xil11] Xilinx. *Space-grade Virtex-5QV FPGA*. en. 2011. URL: <https://www.xilinx.com/products/silicon-devices/fpga/virtex-5qv.html> (besucht am 23.08.2021).
- [Xil21] Xilinx. *Vitis AI*. en. 2021. URL: <https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html> (besucht am 25.08.2021).
- [Xu+21] Xiaowei Xu u. a. *DAC-SDC Low Power Object Detection Challenge for UAV Applications*. 2021. DOI: 10.1109/TPAMI.2019.2932429. URL: <https://ieeexplore.ieee.org/document/8787881> (besucht am 01.02.2021).
- [Zha+21] Aston Zhang u. a. *Dive into Deep Learning*. 2021. URL: <https://d2l.ai/> (besucht am 29.08.2021).
- [Zha16] LeCun Yann Zhang Xiang. „Universum Prescription: Regularization Using Unlabeled Data“. In: *International journal of computer vision* 108.2 (2016), S. 100–110.
- [Zhu04] Mu Zhu. „Recall, Precision and Average Precision“. In: Aug. 2004.
- [Zie96] James Ziegler. „Terrestrial cosmic rays“. In: *IBM Journal of Research and Development* 40 (Jan. 1996), S. 19–40. DOI: 10.1147/rd.401.0019. (Besucht am 23.08.2021).

A Abbildungsverzeichnis

1	Das im Vorgängerprojekt realisierte Fahrzeug mit optischer Objekterkennung auf einem FPGA. Foto: Niklas Krekel	8
2	Der Aufbau eines Neural Network [IT-19]	10
3	Der Basisaufbau eines CNN [Agg18]	11
4	Anwendung von Filtern im Convolutional Layer [Agg18]	12
5	Funktionsweise eines Filters [Agg18]	13
6	Beispiel für das Max-Pooling einer Featuremap [Agg18]	14
7	Darstellung der Berechnung der IoU [Sub]	15
8	Beispiele aus dem MNIST Datensatz [LeC+98]	17
9	LeNet-5 Architektur [LeC+98]	18
10	Training und Test Fehlerrate bei CNN Netzwerken vor ResNet (CIFAR-10) [He+16]	18
11	Prinzip von ResNet (Residual Layer) [He+16]	19
12	Fehlerrate bei CNN Netzwerken vor ResNet und ResNet (ImageNet) [He+16]	19
13	R-CNN Architektur [Gir+14]	20
14	Selective Search mit Greedy Algorithm [Uij+13]	21
15	Fast R-CNN Architektur [Gir15]	22
16	Zeitvergleich zwischen R-CNN und Fast R-CNN [Voi19]	22
17	Faster R-CNN Architektur [Ren+16]	23
18	RPN Architektur [Ren+16]	25
19	Zeitvergleich zwischen R-CNN, Fast R-CNN und Faster R-CNN [Voi19]	25
20	Mask R-CNN Architektur [He+17]	26
21	Beispiele von Mask R-CNN Resultaten (COCO) [He+17]	26
22	Bildverarbeitung in YOLO [Red+16]	27
23	Aufbau YOLO V1 [Red+16]	28
24	Vergleich des Aufbaus von SSD und YOLO V1 [Liu+16]	29
25	Signum Funktionsverlauf	32
26	Sigmoid Funktionsverlauf	32
27	Odd-Even-Padding [Guo+18]	34
28	ONNX Beispielgraph	39
29	Einordnung von ONNX	40
30	FINN Softwarestack [Xil]	40
31	FINN Datenfluss [Xil]	41
32	AXI4 Write [XIL11]	43
33	AXI4 Read [XIL11]	44
34	Datenübertragung in AXI4-Stream	45
35	Schematischer Aufbau eines FPGA [Mis+06]	46
36	Das PYNQ-Z1 FPGA-Entwicklerboard [Dig21h]	47
37	Verzeichnisstruktur	54
38	Top 3 FPGA Frameworks [Xu+21]	57
39	Visualisierung der Cloud-Architektur der Ressourcen	62
40	Vergleich der Trainings-Dauer Cloud vs. Docker	63
41	FINN End-to-End Flow [Xil]	65
42	Beziehungen zwischen den Controller Klassen der Fahrzeugsteuerung	70
43	Umgesetztes Zustandsdiagramm der Fahrzeugsteuerung mit Gestenerkennung	76
44	Geplantes Zustandsdiagramm der Fahrzeugsteuerung mit Gestenerkennung	77
45	Klassifizierung „Stop“ in der Basestation	84
46	Klassifizierung „Start“ in der Basestation	84
47	Tracking der Person in der Basestation mit Person im Bildzentrum	85
48	Tracking der Person in der Basestation mit Person am Bildrand	85
49	Zustandsdiagramm der Fahrzeugsteuerung - Erweiterung der Gesten Auswahl	92
50	Zustandsdiagramm der Fahrzeugsteuerung - Erweiterung des Tracking Modus	93
51	Verzeichnisstruktur des Datensatzes	106

52	Projekt öffnen	134
53	vehiclepl/src/vehicle.xpr	134
54	Block Design öffnen	135
55	Gesamtansicht des Block Design Teil 1	136
56	Gesamtansicht des Block Design Teil 2	137
57	Relevante IP-Blöcke	138
58	IP-Status	139
59	Address Editor öffnen	139
60	Unzugewiesene Adressen	140
61	Adresse zuweisen	140
62	Korrekt zugewiesene Adressen	140

B Tabellenverzeichnis

1	Vergleich YOLOv2 zu YOLOv3 (Daten basieren auf [RF18])	28
2	XNOR	33
3	Fehlerrate zwischen Padding-Methoden in BNNs [Guo+18]	35
4	Vergleich Virtex-5QV FPGA mit ähnlichsten zwei Boards	51
5	Überblick über Versionen des Gestendatensatzes	53
6	Genauigkeitsresultate	54
7	Genauigkeitsresultate	55
8	Top 3 FPGA Teilnehmer [Xu+21]	57
9	Methoden der Controller Klassen	71
10	Base Station Kürzel [Mül+21a]	73
11	Zustandsmenge mit Beschreibung der Zustände	75
12	Genauigkeitsresultate Evaluationstests mit Datensatz Versionen V4.0 V4.1 und CIFAR10	81
13	Genauigkeitsresultate Evaluationstests mit Datensatz Versionen 5.0 und 5.1	82
14	Genauigkeitsresultate Evaluationstests mit Datensatz Version 6.0	83
15	Parameter zur Erzielung der höchsten Genauigkeit	83
16	Ressourcenauslastung laut Vivado Summary	84

C Network Dataframes (Netzwerkdatenrahmen)

Geschrieben von Tuncer Catalkaya

Register C.1: CONTROL DATAFRAME (VEHICLE TO BASE STATION)

22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
																							0

Inferred Y Position (High Byte)
Inferred Y Position (Low Byte)
Inferred X Position (High Byte)
Inferred X Position (Low Byte)
Frame Class
Frame Processing Time (Low Byte)
Frame Processing Time (High Byte)
Motor Processing Time (Byte 1)
Motor Processing Time (Byte 2)
Motor Right Speed (High Byte)
Motor Right Speed (Low Byte)
Motor Left Speed (High Byte)
Motor Left Speed (Low Byte)
Motor Direction
Tile Count (High Byte)
Tile Count (Low Byte)
Video Feed Enable
Image Height (High Byte)
Image Height (Low Byte)
Image Width (High Byte)
Image Width (Low Byte)
Frame ID

Register C.2: SET RESOLUTION DATAFRAME (BASE STATION TO VEHICLE)

4	3	2	1	0
				0

Height (High Byte)
Height (Low Byte)
Width (High Byte)
Width (Low Byte)
Frame ID

Register C.3: SET VIDEO ENABLED DATAFRAME (BASE STATION TO VEHICLE)

1	0
	1

Video Feed Enable
Frame ID

Register C.4: SET MOTORS DATAFRAME (BASE STATION TO VEHICLE)

6	5	4	3	2	1	0
						2

Speed Right (High Byte)
Speed Right (Low Byte)
Speed Left (High Byte)
Speed Left (Low Byte)
Direction
Manual Mode
Frame ID

D Anleitung: Installation und Nutzung des Gestendatensatzes

Geschrieben von Mattia Uhlenbrock

Der Gestendatensatz liegt in einem projektinternen Repository. Dieses Repository enthält die Bilder und Python-Skripte zur Verarbeitung. Außerdem enthält es Skripte zur Installation von Abhängigkeit zu weiteren Python-Modulen und Skripte zum automatischen Installieren der Abhängigkeiten.

D.1 Installation

Für die Verwendung der Python-Skripte zur Verarbeitung der Bilder ist eine Installation von Python vorausgesetzt. Außerdem müssen zum Ausführen der Skripte `buildimagevariability.py` und `cutoutboundingbox.py` die Abhängigkeiten der im Repository enthaltenen `requirements.txt` installiert werden. Dies kann manuell oder automatisch mit `installrequirements.bat` (Windows) und `installrequirements.sh` (Linux) durchgeführt werden.

D.2 Nutzung

Zunächst muss das Repository geklont werden:

```
git clone git@gitlab.com:soc-nn/gesten-datensatz.git
```

D.2.1 Struktur

Im Verzeichnis `gesture/source` befinden sich nach vollständigen klonen die Bilder des Datensatzes in der aus Abbildung 51a hervorgehenden Struktur. Bei der Verwendung der Skripte ist die Struktur gemäß Abbildung 51b vorgesehen.

D.2.2 Variationen des Datensatzes erzeugen

Durch die Nutzung des Variationsskriptes `buildimagevariability.py` werden die Bilder im `gesture/source` Verzeichnis verzeichnisweise eingelesen und in einer analogen Verzeichnisstruktur im Verzeichnis `gesture/dist` abgelegt. Dabei wird für jedes Bild zusätzlich eine um 90, 180 und 270 Grad rotierte Variante abgespeichert. Zu jeder Rotationsvariante wird außerdem jeweils eine auf der x-Achse und eine auf der y-Achse gespiegelte Variante gespeichert.

```
py scripts/buildimagevariability.py
```

Anschließend kann das `gesture/dist` Verzeichnis als root-Verzeichnis für das Training eines CNN im Hauptrepository SoC-NN genutzt werden.

D.2.3 Ausschneiden von Bildern

Um Bilder in ein per XML definiertes Rechteck auszuschneiden, werden diese aus dem Verzeichnis `gesture/original` durch das Skript `cutoutboundingbox.py` verarbeitet. Die Ordnerstruktur für `gesture/original` ist ebenfalls analog zu `gesture/source`. Die XML-Datei wird mit der Software `labelimg` (weitere Informationen zur Software: <https://github.com/tzutalin/labelimg>) erzeugt. Sie muss den gleichen Name, wie die auszuschneidende Datei haben und in das entsprechende Verzeichnis im `gesture/boundings` Verzeichnis, welches ebenfalls der selben Verzeichnisstruktur folgt, eingefügt sein.

```
py scrips/cutoutbuindingbox.py
```

Die Ergebnisdatei wird mit fortlaufender Nummerierung automatisch in das entsprechende Verzeichnis im Verzeichnis `gesture/source` eingefügt.

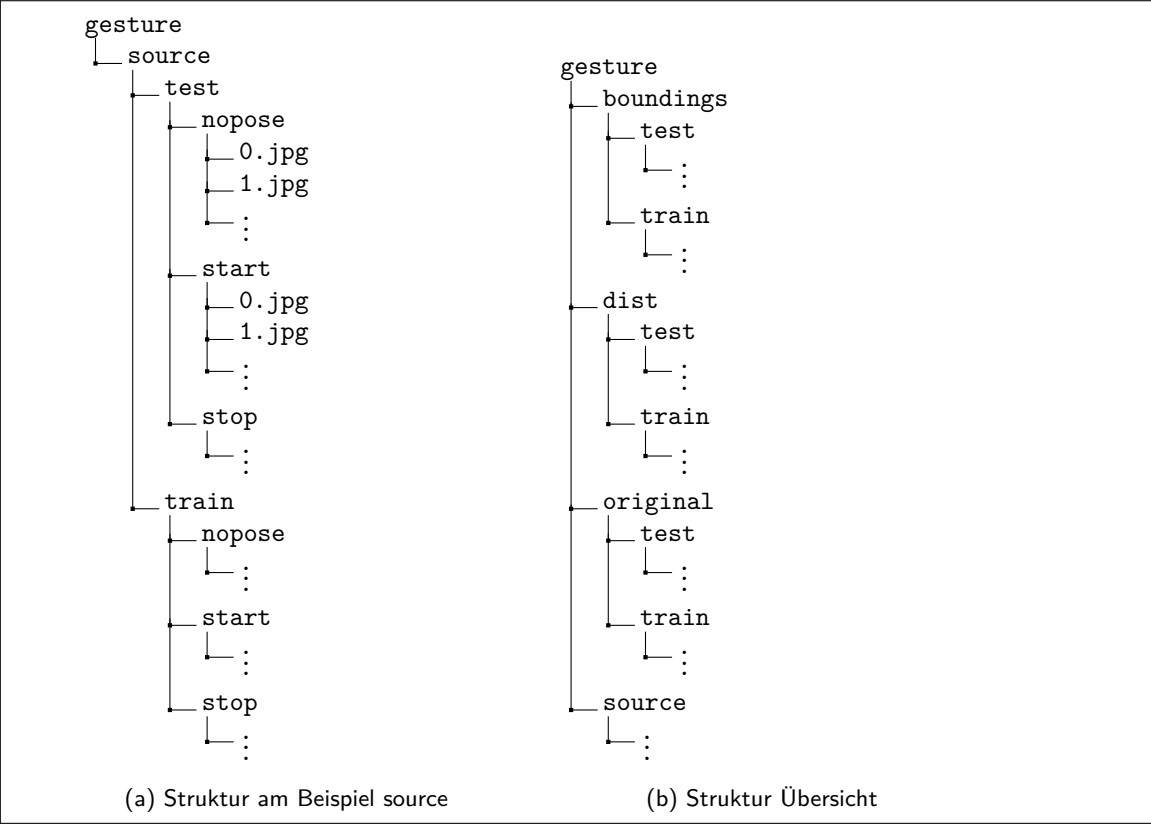


Abbildung 51: Verzeichnisstruktur des Datensatzes

E Anleitung: Trainieren des neuronalen Netzes

Geschrieben von Reena Wichmann

Um ein Modell trainieren zu können müssen die beiden Verzeichnisse `gesten-datensatz` und `soc-nn-main_repo` geklont werden. Im Verzeichnis `soc-nn-main_repo` befindet sich unter `src/training` die Datei `Train.ipynb`, mit welcher das Netz trainiert werden kann. Im Verzeichnis `gesten-datensatz` befinden sich unter `gesture/source` sowohl Trainings- als auch Testfotos, mit Hilfe dessen das neuronale Netz trainiert wird.

E.1 Clonen der benötigten Git Repositories

Zum Clonen der beiden GitLab Repositories `gesten-datensatz` und `soc-nn-main_repo` müssen folgende Befehle ausgeführt werden (in diesem über HTTPS, auch über SSH möglich):

```
git clone https://gitlab.com/soc-nn/soc-nn\_main\_repo.git
```

```
git clone https://gitlab.com/soc-nn/gesten-datensatz.git
```

E.2 Train.ipynb Prepare Environment

Um ein Modell zu trainieren, müssen nun unter Umständen bestimmte Variablen in der `Train.ipynb` angepasst werden. Um diese Variablen zu den richtigen Pfaden anpassen zu können, werden sie im Folgenden einzeln erklärt.

`from models.cnv import CNV` Durch diesen Import wird das Netz importiert, welches in diesem Fall in der `cnv.py` festgelegt ist.

build_dir = `'/Pfad zum geclonten gesten-datensatz Repository/gesten-datensatz'`

data_dir: Hier muss `'data'` zu `'gesture'` verändert werden.

```
data_dir = path.join(build_dir, 'data')
```

model_name: Soll ein Modell neu trainiert werden kann hier der gewünschte Name unter dem das Modell gespeichert werden soll eingetragen werden.

Soll ein bereits trainiertes Modell validiert werden, muss hier der gespeicherte Name der `.pth` Datei eingetragen werden. Diese Datei ist unter `build_dir` gespeichert.

```
model_name = 'CNV'
```

model_path: Unter diesem Pfad wird das trainierte Modell gespeichert beziehungsweise ein bereits trainiertes Modell geladen. Dieser Pfad muss in der Regel nicht verändert werden.

```
model_path = path.join(build_dir, '{model_name}.pth')
```

`num_classes` legt die Anzahl der Klassen fest. In diesem Fall drei für `start`, `stop` und `nopose`.

`weight_bit_width` - Weight Quantization

`act_bit_width` - Activation Quantization

`in_bit_width` - Input Quantization

`in_ch` - in channels (Drei für Rot, Grün und Blau)

`seed` - `torch.seed()`

-> Die beiden Codezellen innerhalb von *Prepare Environment* müssen sowohl zum Trainieren als auch zum Validieren ausgeführt werden, damit alle Variablen gesetzt werden.

E.3 Train.ipynb Training

Die Codezelle innerhalb von *Training* sollte ausgeführt werden, wenn ein neues Modell trainiert werden soll. Hier werden vorher festgelegte Variablen übergeben, die nun nicht mehr verändert werden müssen. Wichtig innerhalb dieser Zelle ist es, die Epochen festzulegen, mit denen das Modell trainiert werden soll. Diese können in *epochs* festgelegt werden.

Durch Ausführen dieser Codezelle wird das zuvor importierte Modell CNV mit den festgelegten Epochen trainiert. Innerhalb des Training erfolgen Ausgaben über den aktuellen Fortschritt. Nach Beenden des Trainings wird das Modell im *.pth* Format im *build_dir* gespeichert.

E.4 Train.ipynb Measure Accuracy

Mittels dieser Codezelle kann die Genauigkeit eines neu trainierten oder bereits gespeicherten Modells anhand der Testfotos gemessen werden. Soll ein zuvor gespeichertes Modell validiert werden, muss die Codezelle in *Training* ausgelassen werden und in den die Variable *model_name* das richtige Modell eingetragen werden. Gegebenenfalls muss zudem der *model_path* angepasst werden.

E.5 Train.ipynb Finn Export

Die hier enthaltene Codezelle dient dazu das zuvor trainierte Modell als *.onnx* Datei abzuspeichern. Diese wird ebenso unter dem in *build_dir* angegebenen Verzeichnis gespeichert.

F Leitfaden zur Erstellung von Datensatz-Bildern

Der folgende Leitfaden ist eine Anleitung zur Erstellung von Bildern für den Datensatz.

F.1 Vorgaben

F.1.1 Posen

Die aufgenommenen Bilder sollten in etwa gleichen Anteilen auf drei Klassen von Posen verteilt sein:

- "START" - Beide Arme einer Person stehen horizontal von den Schultern ab. Die Unterarme sind um 90° vertikal nach oben angewinkelt.

Beispiele:



- "STOP" - Beide Unterarme einer Person sind vor ihrer Brust überkreuzt.

Beispiele:



- "NOPOSE" - Neutral, jede andere erdenkliche natürliche Position einer Person.

Beispiele:



Hier kann man kreativ werden.

F.1.2 Konstante Parameter

Folgende Parameter sollen konstant gehalten werden:

- Die Person soll vertikal und mittig im Bild stehen.
- Es sollen Kopf, Arme und Oberkörper der Person zu sehen sein.
- Das Seitenverhältnis soll 1:1 betragen (Quadratisch).¹
- Es soll möglichst wenig Kontext (Hintergrund) links, rechts und oberhalb der Person zu sehen sein.

Negativbeispiel:



F.1.3 Variable Parameter

Wenn möglich, sollen folgende Parameter in einer Bildreihe variiert werden:

- Bekleidung
- Hintergrund
- Ausrichtung zur Kamera (geringfügige Variation)
- Belichtung (zu verschiedenen Tageszeiten oder Wetterbedingungen)

¹Die meisten Smartphones unterstützen Aufnahmen in 1:1 in den Einstellungen der Kamera-App. Sollte dies nicht möglich sein, können die Bilder auch in anderen Seitenverhältnissen hochgeladen werden. In dem Fall sollte dies im Verzeichnisnamen gekennzeichnet werden (z.B. "toni.16_9.zip"), um die Nachbearbeitung zu erleichtern.

F.2 Ergebnisse hochladen

Bitte beachten Möglichst folgende Ordnerstruktur beibehalten: Die START, STOP und NOPOSE Bilder in einzelne Ordner aufteilen. Dann alles als eine zip-Datei "name.zip" zusammenfassen.

Hochladen

- In der SoC_NN Aulis Gruppe unter "Datensatz für Gestenerkennung" als zip

oder

- per E-Mail (s.u.) als zip

Bei Fragen E-Mail an:

tdragojevic@stud.hs-bremen.de oder
muhlenbrock@stud.hs-bremen.de

G Anleitung: Installation von FINN, Vitis (Vivado) und Xilinx- Runtime (Xrt) in Ubuntu

Geschrieben von Tobias Nießen

Im Folgenden wird die Installation von FINN und den benötigten weiteren Programmen Vitis, Vivado und Xrt beschrieben. Diese Anleitung beschränkt sich auf die Installation in Linux, da diese wesentlich einfacher ist. Rein technisch sind alle Voraussetzungen ebenfalls vorhanden, um diese Programme in Windows Subsystem for Linux (WSL) zu installieren. Zum Zeitpunkt der Textfassung gab es hier allerdings Probleme bei Windows. Laut den Patchnotes im August sollten diese mittlerweile aber behoben und zudem das Ausführen von grafischen Anwendungen mittlerweile möglich sein. Ein Versuch in diese Richtung könnte in Zukunft sinnvoll sein. Hierbei sollte man bedenken, dass WSL keine Virtuelle Maschine darstellt, sondern direkt unter Windows auf einem speziellen Kernel läuft. Fehler könnten hier im Zweifel mehr Schaden anrichten.

Das FINN-Repository ist sehr aktiv. Seit Erstellung dieser Anleitung musste diese bereits mehrfach an neue Workflows angepasst werden. Deswegen arbeiten wir ausschließlich mit **Version 0.5b**. In der folgenden Anleitung wird auch nur exakt diese Version kopiert.

Ein genereller Hinweis vorweg: Docker-Container schließt man, indem im Terminal zweimal hintereinander Ctrl+C gedrückt wird. Nicht durch das Kreuz vom Terminal.

Des Weiteren ist die Darstellung des Codes in Latex nicht immer optimal. Wenn eine Code-Zeile zu lang ist, fügt Latex hier Umbrüche hinzu. Beim einfachen Kopieren der Befehle sollte darauf geachtet werden, dass nicht aus Versehen ein Umbruch mit kopiert wird, wo eigentlich keiner sein sollte. Man sollte Zeile für Zeile kopieren, um die Befehle dann im Terminal wieder zusammenzufügen. Zusammen gehörende Zeilen sind durch einen Pfeil gekennzeichnet. Das folgende Beispiel ist als eine Zeile, also ein Befehl, zu lesen:

```
curl -s -L  
→ https://nvidia.github.io/nvidia-docker/${distribution}/nvidia-docker.list |  
→ sudo tee /etc/apt/sources.list.d/nvidia-docker.list
```

Version wählen An dieser Stelle sei erwähnt, dass falls man Ubuntu sowieso neu installiert, 18.04 LTS installieren werden sollte, da Xilinx Xrt (noch) nicht auf 20.04 installiert werden kann. Für FINN an sich ist das egal, da hierfür einfach nur die passenden Dateien vorhanden sein müssen, bzw. kopiert werden. Will man allerdings generell mit dieser Installation arbeiten, bietet es sich eher an, direkt alles funktionierend zu installieren. Erst recht, falls es nicht in einer VM installiert wird. In einer VM ist eine GPU-Beschleunigung nur mit einer zweiten Grafikkarte möglich. GPU-Beschleunigung ist zum Trainieren quasi zwingend erforderlich, da dies sonst statt Stunden Tage braucht

Bei anderen Linux-Installationen sollte die Anleitung nach 18.04 befolgt werden und falls Fehler/Dependencies nicht lösbar sind, den korrespondierenden Teil der Anleitung für 20.04 ausprobieren.

Falls Linux neu installiert wird, sollte dieses mindestens 200 GB Speicher zugewiesen bekommen, damit Vivado und Vitis installiert werden können. Falls Vivado und Vitis nicht installiert werden soll, sind 30 GB ausreichend. Das Übersetzen in Hardware ist dann nicht möglich; dies ist aber auch nicht immer zwingend zur Entwicklung erforderlich.

Eine Anleitung zur Installation einer VM mit Ubuntu findet sich beispielsweise bei Heise.de: <https://www.heise.de/tipps-tricks/Ubuntu-in-VirtualBox-nutzen-so-klappt-s-4203333.html> (Abgerufen am 23.08.2021)

G.1 Ubuntu 18.04.5 LTS

Ubuntu Version 18.04.5 LTS installieren:

<https://ubuntu.com/download/desktop>

Ubuntu-Versionen-Upgrades ablehnen. Etwaige Software-Updates ausführen.

G.1.1 Vitis (und Vivado) installieren

Die Vitis-Installation enthält Vivado. FINN läuft mit Version 2020.1 stabil. Version 2020.1.1 ist nicht getestet worden. Falls irgendwas mit der neueren Version nicht funktionieren sollte, die alte installieren.

Vitis Version 2020.1 downloaden (**nicht** Version 2020.1.1 „Update 1“):

Xilinx Unified Installer 2020.1: Linux Self Extracting Web Installer

<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2020-1.html>

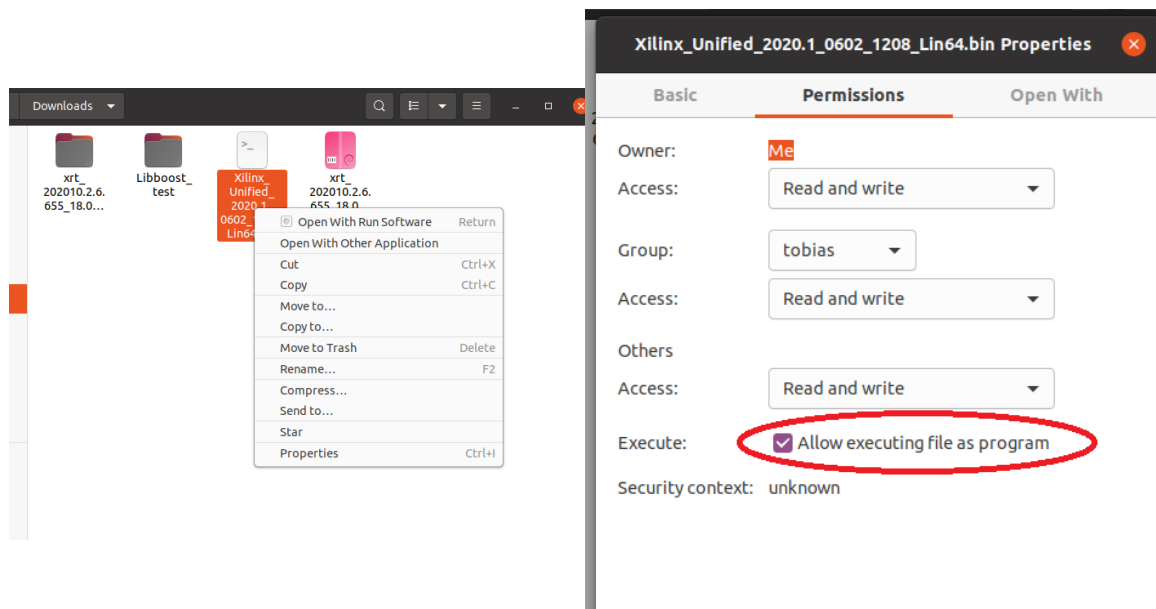
Terminal öffnen und eingeben:

```
sudo gedit /etc/os-release
```

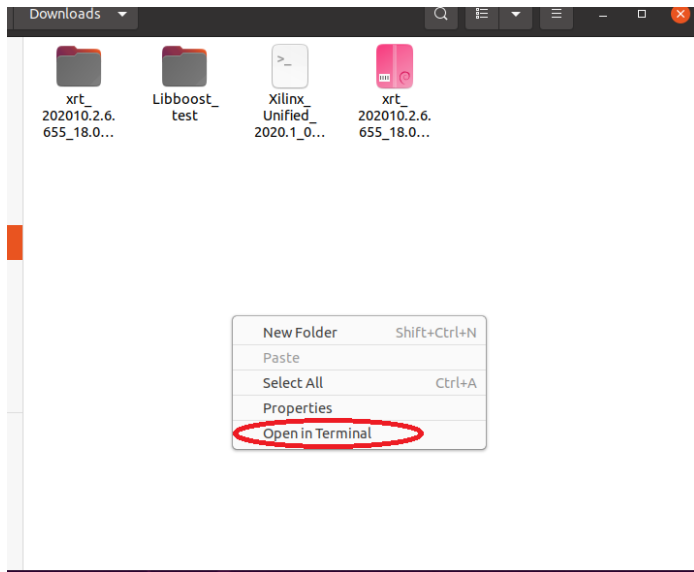
Version ändern in:

```
VERSION="18.04.4 LTS (Bionic Beaver)"
```

Download-Ordner öffnen und in den Eigenschaften vom Installer das Ausführen erlauben:



Neues Terminal öffnen:



Vitis installieren:

```
sudo ./Xilinx_Unified_2020.1_0602_1208_Lin64.bin
```

Nach der Installation und dessen Download von ca. 31 GB, die Version wieder in die Richtige ändern:

```
sudo gedit /etc/os-release
```

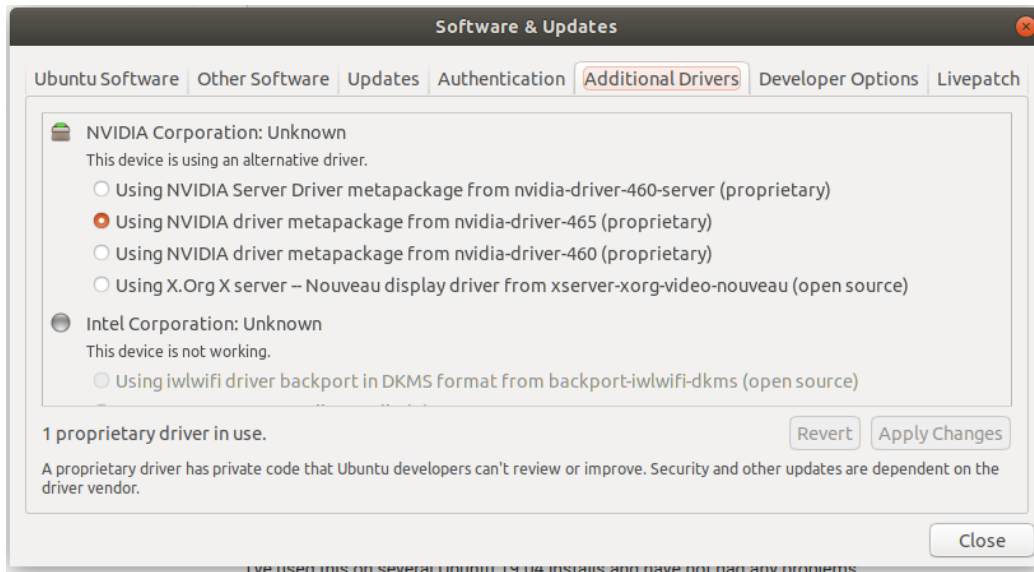
Version ändern in:

```
VERSION="18.04.5 LTS (Bionic Beaver)"
```

G.1.2 Optional: Nvidia Toolkit mit angepasstem Docker

Um die GPU-Beschleunigung in Docker zu aktivieren, wird „nvidia-container-toolkit“ benötigt. „nvidia-docker2“ ist eine veraltete Version, welche nicht mehr unterstützt wird. Anleitungen zu dieser werden nicht funktionieren.

Der Grafikkarten-Treiber sollte neuer als Version 432 sein, und ist im Software-Updater zu finden:



Wenn dieser installiert ist, muss das nvidia-toolkit und das zugehörige Docker installiert werden:

```
sudo add-apt-repository ppa:graphics-drivers/ppa
sudo apt-get update
sudo apt-get install build-essential dkms
sudo apt-get install apt-transport-https
sudo apt-get install ca-certificates
sudo apt-get install curl
sudo apt-get install gnupg-agent
sudo apt-get install software-properties-common
```

Angepasste Docker-Version installieren:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo apt-key fingerprint 0EBFCD88
sudo add-apt-repository "deb [arch=amd64]
↳ https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
sudo groupadd docker
sudo gpasswd -a $USER docker
newgrp docker
```

NVIDIA Container Toolkit installieren:

```
distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add -
curl -s -L
↳ https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list |
↳ sudo tee /etc/apt/sources.list.d/nvidia-docker.list
sudo apt-get update
sudo apt-get install nvidia-container-toolkit
```

Docker neustarten:

```
systemctl restart docker
```

Das Ganze testen:

```
sudo docker run --gpus all --rm nvidia/cuda:11.0-base nvidia-smi
```

G.1.3 Docker installieren

Falls das Nvidia-Toolkit installiert wurde, ist das hier überflüssig. Dann geht's im nächsten Unterkapitel weiter mit „Xrt installieren“.

```
sudo apt-get install docker.io  
sudo groupadd docker  
sudo gpasswd -a $USER docker  
newgrp docker
```

Aus Ubuntu ausloggen und neu einloggen.

G.1.4 Xrt installieren

Xrt downloaden:

Xilinx Runtime - 2020.1 **Ubuntu 18.04**

<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-platforms/2020-1.html>

Auf die heruntergeladene Datei doppelklicken. Dann auf „Installieren“ klicken.

G.1.5 FINN-Framework installieren

Im SoC-NN Repository befindet sich ebenfalls unter `/lib/finn/` die selbe Version. Diese wird jedoch nicht funktionieren. Deswegen muss das FINN Git nochmals separat geklont werden.

Im FINN Repository gibt es oft Updates oder Änderungen. Damit diese uns nicht betreffen, wird eine festgelegte Version geklont. FINN Repository Version 0.5b klonen:

```
git clone -b v0.5b https://github.com/Xilinx/finn
```

Umgebungsvariablen setzen:

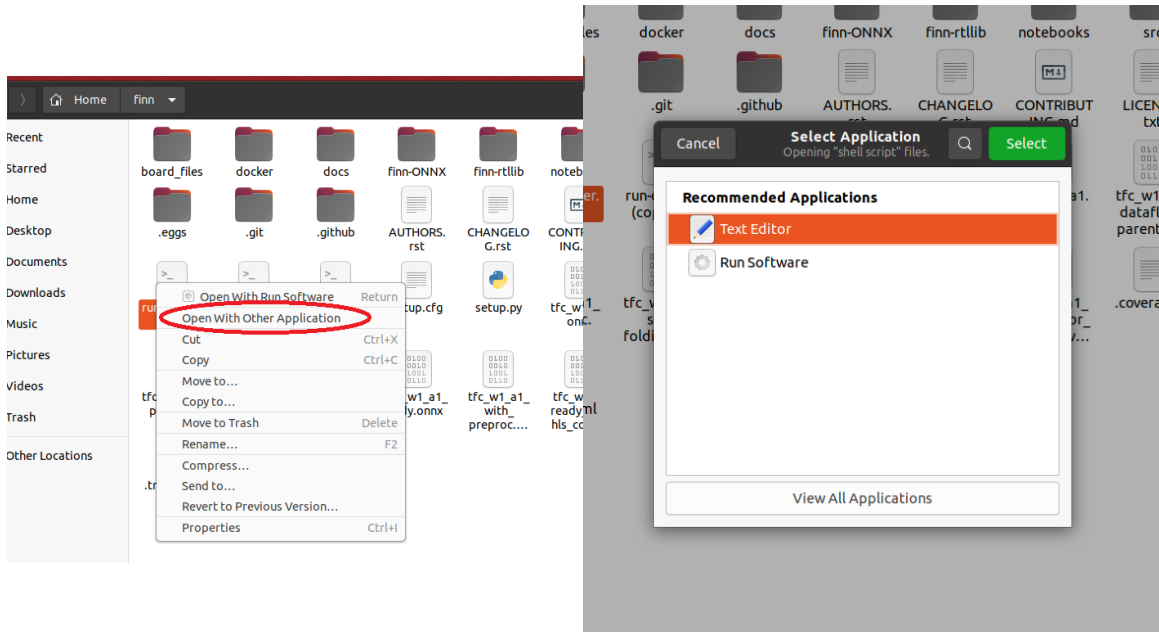
```
sudo -H gedit /etc/environment
```

Dort als neue Zeilen hinzufügen:


```
VIVADO_PATH="/tools/Xilinx/Vivado/2020.1/"
VITIS_PATH="/tools/Xilinx/Vitis/2020.1/"
PLATFORM_REPO_PATHS="/opt/xilinx/platforms"
XILINX_XRT="/opt/xilinx/xrt/"
```

Optional:

In „run-docker.sh“ kann, je nach vorhandener GPU-Beschleunigung, diese aktiviert und deaktiviert werden. Die Datei mit Text Editor öffnen:



Diese Zeile suchen:

```
DOCKER_EXEC="docker run -t --rm $DOCKER_INTERACTIVE --init "
```

Falls Ubuntu in einer VM läuft, wird die GPU-Beschleunigung nicht funktionieren. Falls Ubuntu nicht in einer VM läuft, die gesuchte Zeile ändern in:

```
DOCKER_EXEC="docker run --gpus all -t --rm $DOCKER_INTERACTIVE --init "
```

Ebenfalls müssen dann eventuell noch Treiber von Nvidia installiert werden.

Ubuntu neu starten.

G.1.6 FINN starten

Im „finn“ Ordner:

```
./run-docker.sh notebook
```

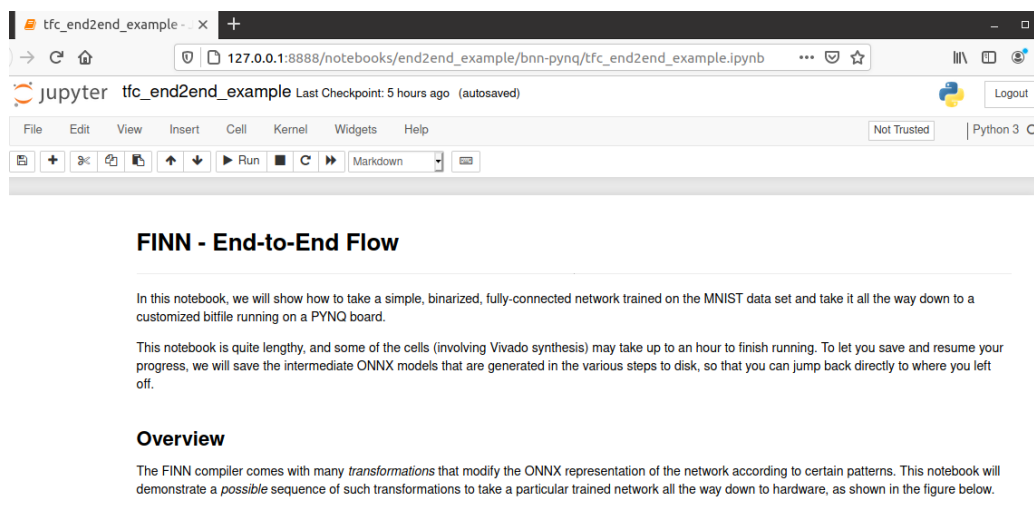
ausführen. (Dauert ca. 20 Minuten)

Am Ende steht dort ein Link in der Form:

„<http://127.0.0.1:8888/?token=90b01d249bb75cc9df8c20b65383ea472206e6bdef5d55d3>“

Diesen in Firefox einfügen.

Hier kann man sich nun z. B. in `/end2end_example/bnn-pynq/` das „tfc_end2end_example“ angucken. Das sollte dann so aussehen:



The screenshot shows a Jupyter Notebook interface in a browser. The title bar reads "tfc_end2end_example". The address bar shows the URL "127.0.0.1:8888/notebooks/end2end_example/bnn-pynq/tfc_end2end_example.ipynb". The Jupyter logo and "tfc_end2end_example" are visible, along with "Last Checkpoint: 5 hours ago (autosaved)" and a "Logout" button. The notebook content includes a title "FINN - End-to-End Flow" and two paragraphs of text. The first paragraph states: "In this notebook, we will show how to take a simple, binarized, fully-connected network trained on the MNIST data set and take it all the way down to a customized bitfile running on a PYNQ board." The second paragraph states: "This notebook is quite lengthy, and some of the cells (involving Vivado synthesis) may take up to an hour to finish running. To let you save and resume your progress, we will save the intermediate ONNX models that are generated in the various steps to disk, so that you can jump back directly to where you left off." Below this is an "Overview" section with the text: "The FINN compiler comes with many *transformations* that modify the ONNX representation of the network according to certain patterns. This notebook will demonstrate a *possible* sequence of such transformations to take a particular trained network all the way down to hardware, as shown in the figure below."

Hier muss allerdings die Netron-Version noch angepasst werden:



The white fields show the state of the network representation in the respective step. The colored fields represent the transformations that are applied to the network to achieve a certain result. The diagram is divided into 5 sections represented by a different color, each of it includes several flow steps. The flow starts in top left corner with Brevitas export (green section), followed by the preparation of the network (blue section) for the Vivado HLS synthesis and Vivado IPI stitching (orange section), and finally building a PYNQ overlay bitfile and testing it on a PYNQ board (yellow section). There is an additional section for functional verification (red section) on the right side of the diagram, which we will not cover in this notebook. For details please take a look in the verification notebook which you can find [here](#)

This Jupyter notebook is organized based on the sections described above. We will use the following helper functions, `showSrc` to show source code of FINN library calls and `showInNetron` to show the ONNX model at the current transformation step. The Netron displays are interactive, but they only work when running the notebook actively and not on GitHub (i.e. if you are viewing this on GitHub you'll only see blank squares).

```
In [1]: from finn.util.visualization import showSrc, showInNetron
        from finn.util.basic import make_build_dir

        build_dir = "/workspace/finn"
```

Outline

```
from finn.util.basic import make_build_dir
from finn.util.visualization import showInNetron

build_dir = "/workspace/finn"
```

Wird geändert in (richtige Einrückung/Abstände sind wichtig):

```
import netron
from IPython.display import IFrame
from finn.util.visualization import showSrc
from finn.util.basic import make_build_dir

def showInNetron(model_filename):
    netron.start(model_filename, address=('0.0.0.0', 8081), browse=False)
    return IFrame(src="http://127.0.0.1:8081/", width="100%", height=400)

build_dir = "/workspace/finn"
```

Das sollte dann so aussehen:

when running the notebook actively and not on GitHub (i.e. if you are viewing this on GitHub you'll only see blank squares).

```
In [12]: import netron
         from IPython.display import IFrame
         from finn.util.visualization import showSrc
         from finn.util.basic import make_build_dir

         def showInNetron(model_filename):
             netron.start(model_filename, address=('0.0.0.0', 8081), browse=False)
             return IFrame(src="http://127.0.0.1:8081/", width="100%", height=400)

         build_dir = "/workspace/finn"
```

Jetzt sollte das Notebook funktionieren.

G.2 Ubuntu 20.04.2.0 LTS

Ubuntu Version 20.04.2.0 LTS installieren:
<https://ubuntu.com/download/desktop>

```
https://ubuntu.com/download/desktop
```

Ubuntu-Versionen-Upgrades ablehnen. Etwaige Software-Updates ausführen.

G.2.1 Vitis (und Vivado) installieren

Die Vitis-installation enthält Vivado.

Vitis Version 2020.1 downloaden (**nicht** Version 2020.1.1 „Update 1“):

Xilinx Unified Installer 2020.1: Linux Self Extracting Web Installer

<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2020-1.html>

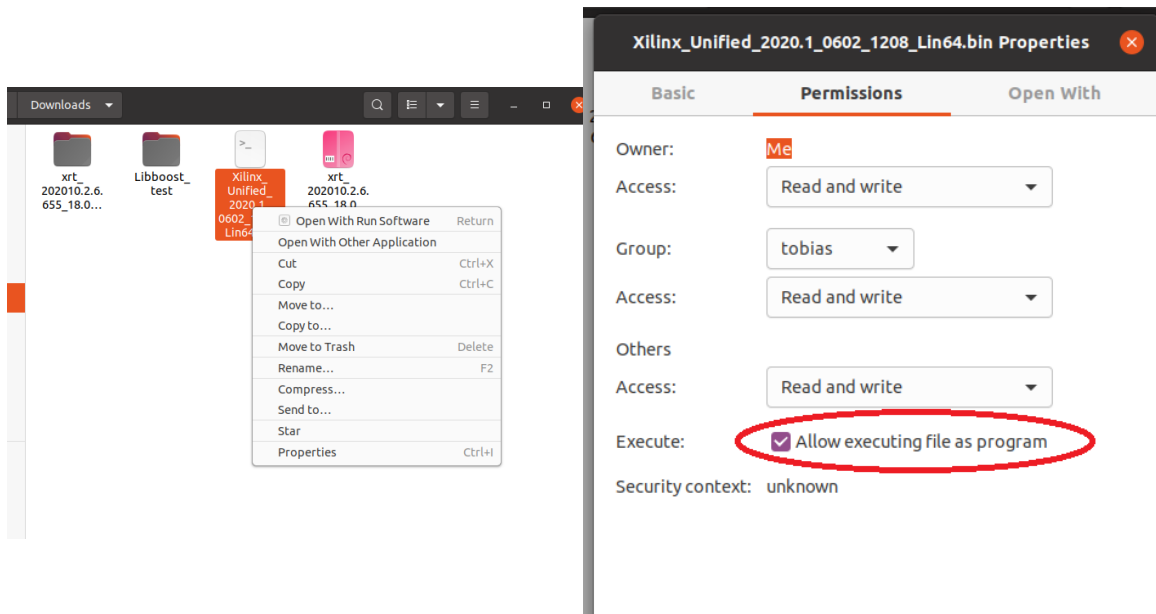
Terminal öffnen und eingeben:

```
sudo gedit /etc/os-release
```

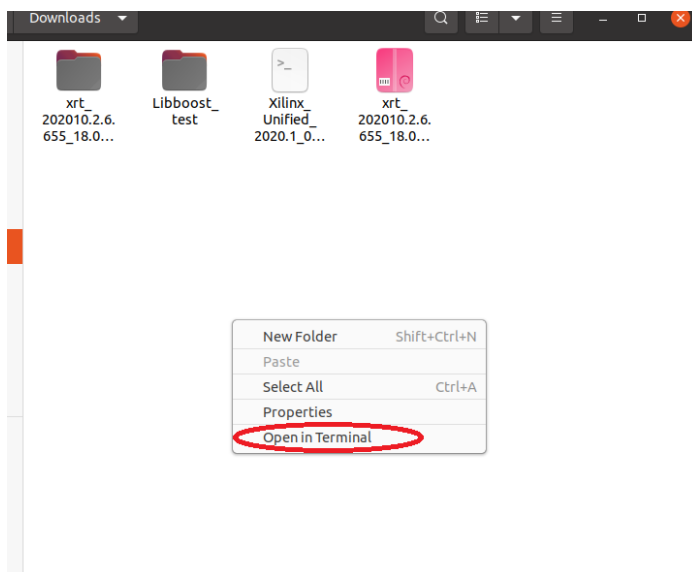
Version ändern in:

```
VERSION="18.04.4 LTS (Bionic Beaver)"
```

Download-Ordner öffnen und in den Eigenschaften vom Installer das Ausführen erlauben:



Neues Terminal öffnen:



Vitis installieren:

```
sudo ./Xilinx_Unified_2020.1_0602_1208_Lin64.bin
```

Nach der Installation und dessen Download von ca. 31 GB, die Version wieder in die Richtige ändern:

```
sudo gedit /etc/os-release
```

Version ändern in:

```
VERSION="20.04.2 LTS (Focal Fossa)"
```

G.2.2 Docker installieren

Falls das Nvidia-Toolkit und damit die GPU-Beschleunigung installiert werden soll, dieses Kapitel nicht beachten und stattdessen die Anleitung aus Unterunterabschnitt G.1.2 befolgen.

```
sudo apt-get install docker.io
sudo groupadd docker
sudo gpasswd -a $USER docker
newgrp docker
```

Aus Ubuntu ausloggen und neu einloggen.

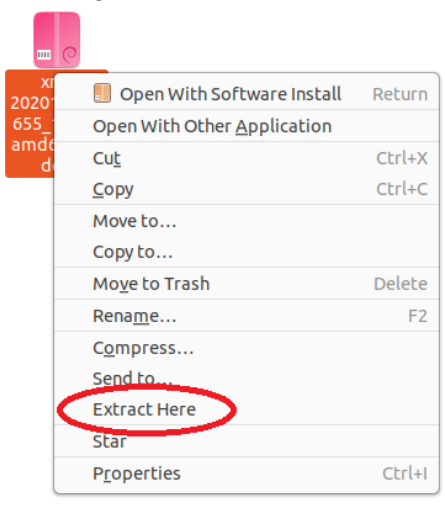
G.2.3 Xrt installieren

Xrt downloaden:

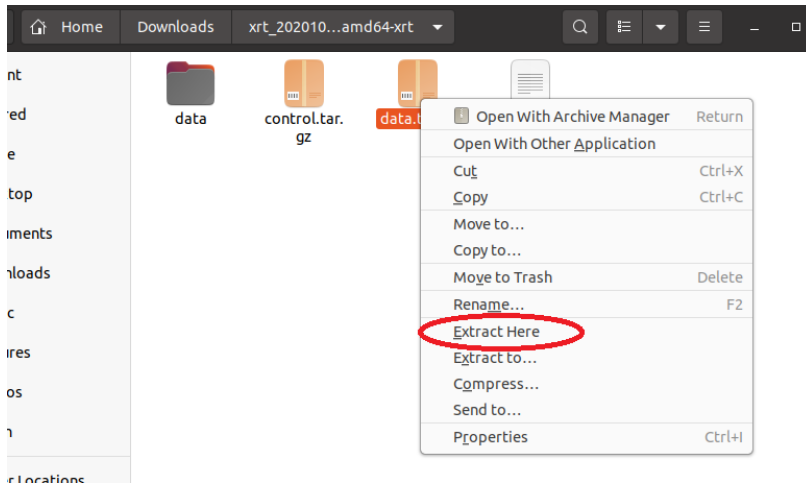
Xilinx Runtime - 2020.1 **Ubuntu 18.04**

<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-platforms/2020-1.html>

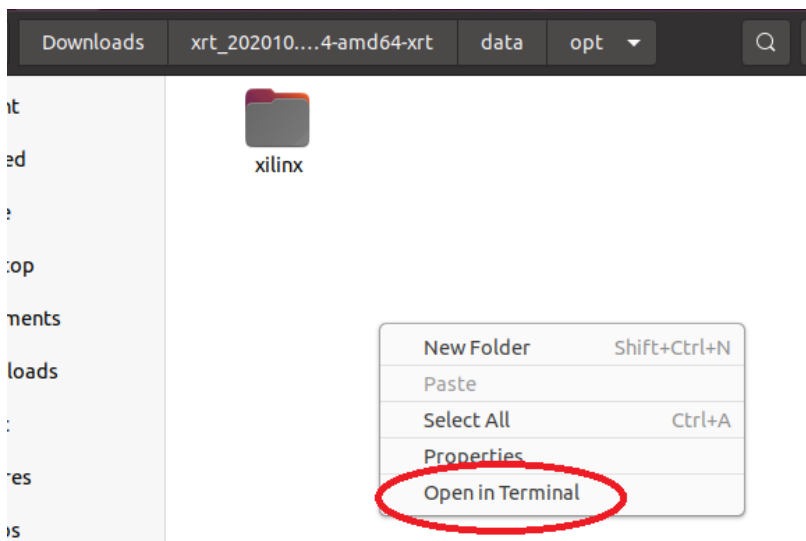
Heruntergeladene Datei extrahieren:



In das extrahierte Verzeichnis wechseln und „data.tar.gz“ ebenfalls extrahieren:



In data/opt ein neues Terminal öffnen:



Im geöffneten Terminal eingeben:

```
sudo cp -r xilinx /opt
```

G.2.4 FINN-Framework installieren

Im FINN Repository gibt es oft Updates oder Änderungen. Damit diese uns nicht betreffen, wird eine festgelegte Version geklont. FINN Repo Version 0.5b klonen:

```
git clone -b v0.5b https://github.com/Xilinx/finn
```

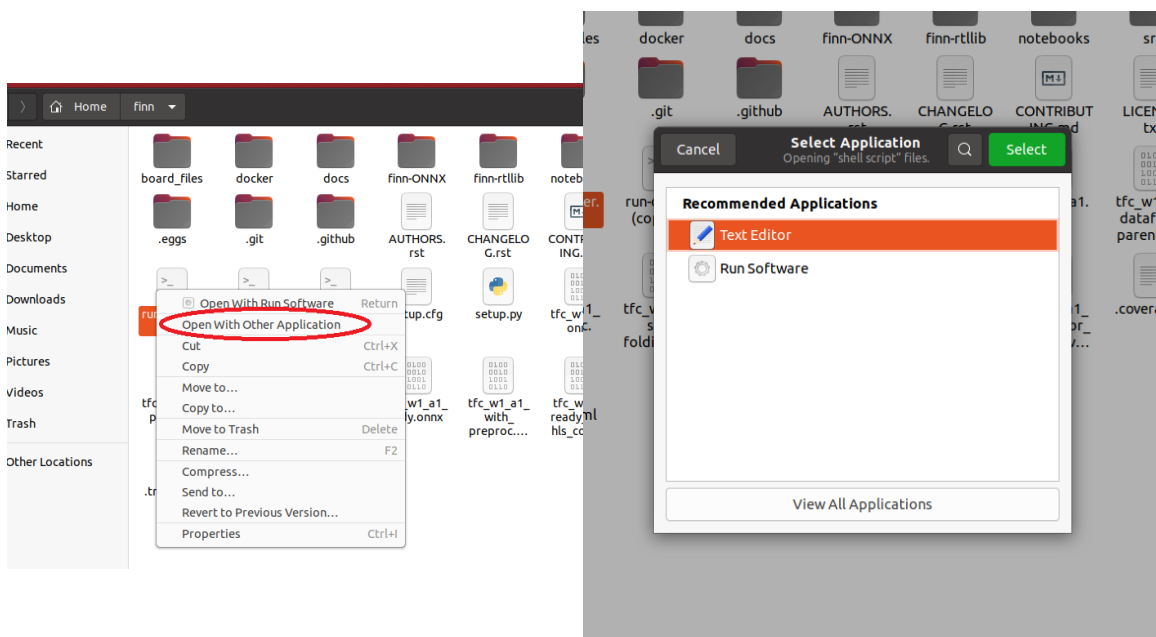
Umgebungsvariablen setzen:

```
sudo -H gedit /etc/environment
```

Dort als neue Zeilen hinzufügen:

```
VIVADO_PATH="/tools/Xilinx/Vivado/2020.1/"  
VITIS_PATH="/tools/Xilinx/Vitis/2020.1/"  
PLATFORM_REPO_PATHS="/opt/xilinx/platforms"  
XILINX_XRT="/opt/xilinx/xrt/"
```

In „run-docker.sh“ kann, je nach vorhandener GPU-Beschleunigung, diese aktiviert und deaktiviert werden. Die Datei mit Text Editor öffnen:



Diese Zeile suchen:

```
DOCKER_EXEC="docker run -t --rm $DOCKER_INTERACTIVE --init "
```

Optional: Falls Ubuntu in einer VM läuft, wird die GPU-Beschleunigung nicht funktionieren. Falls Ubuntu nicht in einer VM läuft, die gesuchte Zeile ändern in:

```
DOCKER_EXEC="docker run --gpus all -t --rm $DOCKER_INTERACTIVE --init "
```

Ebenfalls müssen dann eventuell noch Treiber von Nvidia installiert werden.

Ubuntu neu starten.

G.2.5 FINN starten

Im „finn“ Ordner:

```
./run-docker.sh notebook
```

ausführen. (Dauert 20 Minuten)

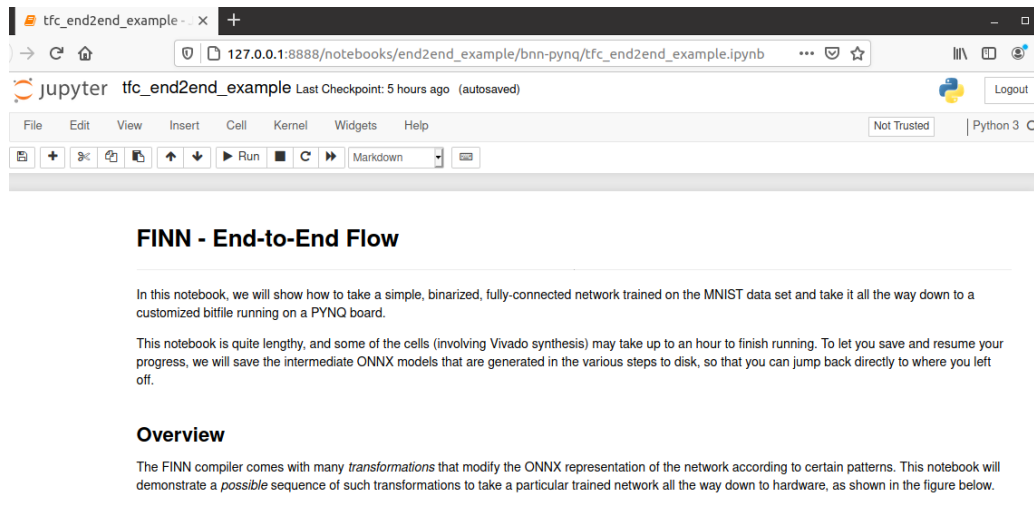
Am Ende steht dort ein Link in der Form:

„<http://127.0.0.1:8888/?token=90b01d249bb75cc9df8c20b65383ea472206e6bdef5d55d3>“

Diesen in Firefox einfügen.

Hier kann man sich nun z. B. in `/end2end_example/bnn-pynq/` das „tfc_end2end_example“ angucken.

Das sollte dann so aussehen:



Hier muss allerdings die Netron-Version noch angepasst werden:



The white fields show the state of the network representation in the respective step. The colored fields represent the transformations that are applied to the network to achieve a certain result. The diagram is divided into 5 sections represented by a different color, each of it includes several flow steps. The flow starts in top left corner with Brevitas export (green section), followed by the preparation of the network (blue section) for the Vivado HLS synthesis and Vivado IPI stitching (orange section), and finally building a PYNQ overlay bitfile and testing it on a PYNQ board (yellow section). There is an additional section for functional verification (red section) on the right side of the diagram, which we will not cover in this notebook. For details please take a look in the verification notebook which you can find [here](#)

This Jupyter notebook is organized based on the sections described above. We will use the following helper functions, `showSrc` to show source code of FINN library calls and `showInNetron` to show the ONNX model at the current transformation step. The Netron displays are interactive, but they only work when running the notebook actively and not on GitHub (i.e. if you are viewing this on GitHub you'll only see blank squares).

```
In [1]: from finn.util.visualization import showSrc, showInNetron
        from finn.util.basic import make_build_dir

        build_dir = "/workspace/finn"
```

Outline

```
from finn.util.basic import make_build_dir
from finn.util.visualization import showInNetron

build_dir = "/workspace/finn"
```

Wird geändert in (richtige Einrückung/Abstände sind wichtig):

```
import netron
from IPython.display import IFrame
from finn.util.visualization import showSrc
from finn.util.basic import make_build_dir

def showInNetron(model_filename):
    netron.start(model_filename, address=('0.0.0.0', 8081), browse=False)
    return IFrame(src="http://127.0.0.1:8081/", width="100%", height=400)

build_dir = "/workspace/finn"
```

Das sollte dann so aussehen:

when running the notebook actively and not on GitHub (i.e. if you are viewing this on GitHub you'll only see blank squares).

```
In [12]: import netron
         from IPython.display import IFrame
         from finn.util.visualization import showSrc
         from finn.util.basic import make_build_dir

         def showInNetron(model_filename):
             netron.start(model_filename, address=('0.0.0.0', 8081), browse=False)
             return IFrame(src="http://127.0.0.1:8081/", width="100%", height=400)

         build_dir = "/workspace/finn"
```

Jetzt sollte das Notebook funktionieren.

G.3 Sonstiges / Mögliche Fehler

Einen Docker-Container schließen Docker-Container schließt man, indem im Terminal zweimal hintereinander Ctrl+C gedrückt wird. Nicht durch das Kreuz vom Terminal.

Alle Docker beenden

```
docker kill $(docker ps -q)
```

Could not resolve host github.com

Docker beenden und Schließen. Dann in einem neuen Terminal:

```
git config --global --unset http.proxy  
git config --global --unset https.proxy
```

Ubuntu neu starten und erneut probieren.

H Anleitung: Synthese bei Änderung der FPGA-Hardware

Geschrieben von Tobias Nießen

Falls die an den FPGA angeschlossene Hardware geändert werden soll, muss dieser mit einem anderen Bitstream beschrieben werden. Dieser Bitstream wird in Vivado generiert und anschließend in die Fahrzeugsteuerung kopiert, welche ihn dann schlussendlich auf den FPGA schreibt.

Vivado kennt Das Pynq-Z1 standardmäßig nicht. Die Board-Files müssen heruntergeladen werden: https://github.com/cathalmccabe/pynq-z1_board_files/raw/master/pynq-z1.zip

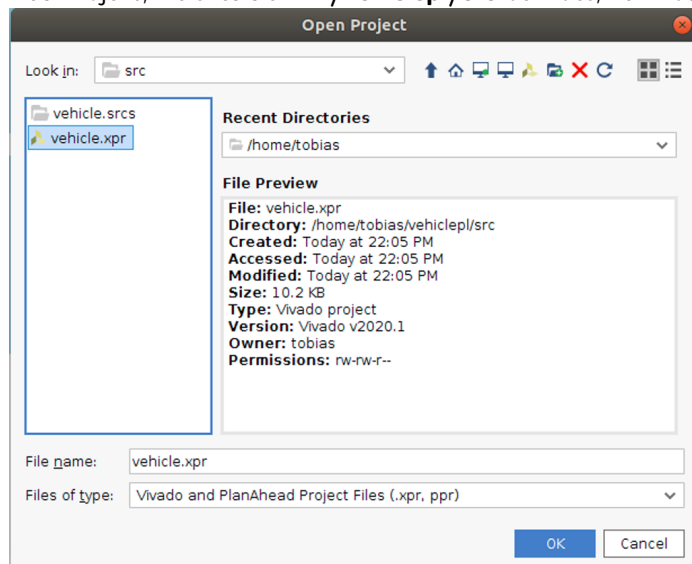
Anschließend muss der entzippte Ordner „pynq-z1“ kopiert werden nach:

```
tools/Xilinx/Vivado/2020.1/data/boards/board_files
```

Dann muss das Projekt gedownloadet werden:

```
git clone https://gitlab.com/soc-nn/vehiclepl
```

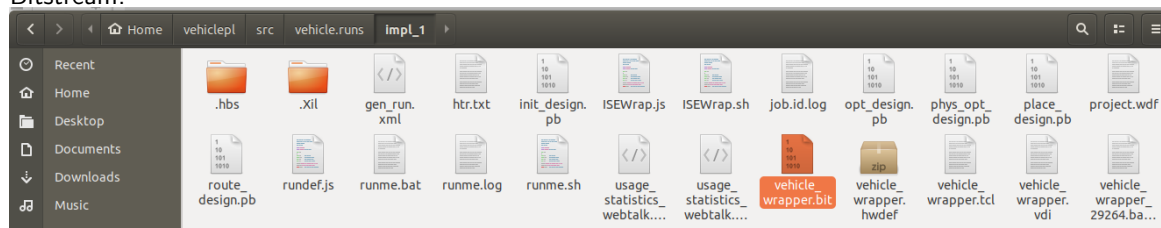
Das Projekt, welches sich in **/vehiclepl/src** befindet, kann dann in Vivado geöffnet werden:



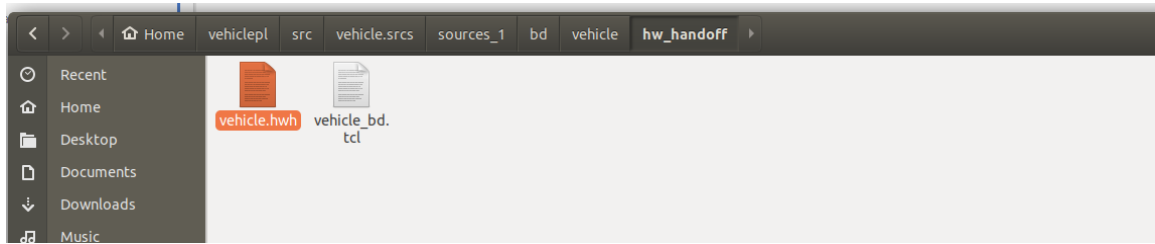
Unter „Open Block Design“ ist das Block Design der verwendeten Hardware dargestellt.

Hat man das Design oder die Hardware-Belegung geändert, so muss eine neuer Bitstream generiert werden, welchen das Board benötigt. Hierzu muss zuerst „Run Synthesis“ ausgeführt werden, anschließend „Run Implementation“ und zuletzt „Generate Bitstream“.

Ist dies abgeschlossen, so findet sich in **vehiclepl/src/vehicle.runs/impl_1/vehicle_wrapper.bit** der Bitstream:



und in **vehiclepl/src/vehicle.srcs/sources_1/bd/vehicle/hw_handoff/vehicle.hwh** die Hardwarekonfiguration:



Damit die neue Konfiguration auf das Board gespielt wird, müssen diese beiden in **vehicleps/hw** eingefügt werden und anschließend das Projekt neu auf das Board übertragen werden.

Die Hardware-Konfiguration ist momentan wie folgt:

L298N	OV7670	PYNQ-Z1	Funktion
	3.3V	3.3V	3.3V
GND	GND	GND	Ground
	SCL	J2 - SCL	I ² C Clock
	SDA	J2 - SDA	I ² C Data
	VS	J4 - IO00	Vertical-Sync
	HS	J4 - IO26	Horizontal-Reference
	PLK	J4 - IO01	Pixel Clock
	XLK	J4 - IO27	Input Clock
	D7	J4 - IO02	Data Bit 7
	D6	J4 - IO28	Data Bit 6
	D5	J4 - IO03	Data Bit 5
	D4	J4 - IO29	Data Bit 4
	D3	J4 - IO04	Data Bit 3
	D2	J4 - IO30	Data Bit 2
	D1	J4 - IO05	Data Bit 1
	D0	J4 - IO31	Data Bit 0
	RET	J4 - IO06	Reset (0: Reset, 1: Normal)
	PWDN	J4 - IO32	Power Down (0: Normal, 1: Power down)
ENA		J3 - IO34	Motor Enable A
IN1		J3 - IO35	Motor Input 1
IN2		J3 - IO36	Motor Input 2
IN3		J3 - IO37	Motor Input 3
IN4		J3 - IO38	Motor Input 4
ENB		J3 - IO39	Motor Enable B

I Anleitung: Inbetriebnahme Pynq-Z1 und Installation vom Cross-Compiler

Geschrieben von Tobias Nießen

Diese Anleitung beschreibt die Inbetriebnahme des Pynq-Fahrzeuges und das Einbinden dieses in ein WLAN-Netz, sodass eine Fernsteuerung möglich wird. Außerdem wird das Kompilieren sowohl per Cross-Compiler, als auch direkt auf dem Board erklärt, wobei der Cross-Compiler meist zu bevorzugen ist, da dieser keine Nachteile hat, solange das Netzwerk richtig konfiguriert ist.

Das Pynq-Z1 kann mittels eingebauter Powerbank oder externem 12V-Netzteil mit Strom versorgt werden. Anfangs muss das Board mittels Lan-Kabel mit dem Netzwerk verbunden werden. Das Board ist so konfiguriert, dass es den Hostname „pynq“ anfordert. Wenn der Router das richtig interpretiert hat, so sollte das Board unter pynq im Browser aufrufbar sein. Hier öffnet sich das bereits enthaltene Jupyter-Notebook. Das Passwort ist „xilinx“.

Eine Verbindung per ssh ist ebenfalls möglich mit xilinx@pynq, Benutzer:xilinx, Passwort:xilinx.

Um WLAN zu konfigurieren, sollte nach Eingabe von

```
ifconfig
```

ein Eintrag mit wlan0 erscheinen. Falls dem nicht so ist, muss das WLAN gestartet werden:

```
sudo ifconfig wlan0 up
```

Anschließend kann man zwecks Verifizierung nach den vorhandenen WLAN-Netzen suchen:

```
sudo iwlist wlan0 s | grep WLAN-NETZ-NAME
```

Zur Automatischen Verbindung wird der WPA-Passkey benötigt:

```
wpa_passphrase WLAN-NETZ-NAME PASSWORT
```

Aus dem Output muss das hinter „psk=“ kopiert werden:

```
xilinx@pynq:~$ wpa_passphrase [REDACTED]
network={
    ssid="[REDACTED]"
    #psk="[REDACTED]"
    psk=[REDACTED]
}
```

Anschließend wird die Interface-Konfiguration geöffnet:

```
sudo nano /etc/network/interfaces.d/wlan0
```

Dort wird folgendes eingetragen, wobei die SSID und der psk ergänzt werden muss:

```
auto wlan0
iface wlan0 inet dhcp
wpa-ssid
wpa-psk
```

```
auto wlan0
iface wlan0 inet dhcp
wpa-ssid [REDACTED]
wpa-psk [REDACTED]
```

Nach einem Neustart sollte das Pynq sich per WLAN verbinden:

```
sudo reboot
```

Für die eigentliche Entwicklung kann natürlich direkt auf dem Board kompiliert werden. Der Prozessor ist jedoch nicht sehr leistungsstark. So ziemlich jeder PC sollte signifikant besser hierzu geeignet sein. Aus diesem Grunde wurde ein Cross-Compiler implementiert, welcher sich automatisch mit dem Pynq verbindet und die kompilierte Software hochladen und ausführen kann.

1.1 Cross-Compiler

Das Processing System muss geklont werden:

```
git clone https://gitlab.com/soc-nn/vehicleps
```

In den Ordner wechseln und das Skript „run.sh“ ausführbar machen:

```
cd ~/vehicleps
sudo chmod 774 run.sh
```

Der Cross-Compiler wird mittels eines Docker-Containers über das Skript „run.sh“ gestartet:

```
./run.sh build_x86
```

Das Kompilierte kann mittels

```
./run.sh run
```

direkt auf das Board kopiert und ausgeführt werden. Hierzu sollte beachtet werden, dass in der „vehicle.conf“ die richtige IP-Adresse bei **BASE_STATION_IP** eingetragen wird. In dieser Datei kann ebenfalls ausgewählt werden, ob die Kamera oder eines der Testvideos genutzt werden soll.

Ebenfalls kann das fertig kompilierte Programm auch direkt in Docker gestartet werden:

```
./run.sh run_x86
```

Eine Kommandozeile in dem Docker-Container wird geöffnet mit:

```
./run.sh
```

I.2 Kompilieren direkt auf dem Board

Das Processing System muss geklont werden:

```
git clone https://gitlab.com/soc-nn/vehicleps
```

In den Ordner **vehicleps/lib/vivado_include** muss noch der Inhalt aus einer Vivado-Installation aus dem Ordner „include“ kopiert werden. Anschließend wird die Kompilierung in **vehicleps** gestartet:

```
cd ~/vehicleps  
sudo make run
```

Dies dauert pro Kompilierung ca. 5 bis 15 Minuten.

J Anleitung: Übersetzung der Netztopologie für das FPGA

Geschrieben von Ismail Sastim

Das *Train.ipynb* Notebook legt innerhalb der Dockerumgebung unter */workspace/build/* die *MODEL_NAME.onnx* Datei der trainierten Models ab. Um aus diesen die IP-Blöcke für die Bitstream Generierung in Vivado zu erstellen, steht das *Synthesize.ipynb* Notebook zur Verfügung, welches an [FINa] angelehnt ist aber sich auf das wesentliche beschränkt.

Hier sind ggfs. einige Anpassung nötig.

- Zum einen muss der zu übersetzende Model-Name angepasst werden, welcher dem der *.onnx*-Datei gleich ist.
- Zum anderen der folding-Parameter entsprechend der Netztopologie. Hier wurde außer der letzten Tupel nichts verändert, da an der *CNV_IWIA*-Tpologie keine Anpassungen vorgenommen wurden. Eine wichtige Außnahme ist hier die Anzahl der Ausgabeklassen. Da wir nur drei Klassen ("*START*", "*STOP*", "*NOPOSE*") haben statt den üblichen zehn aus den bekannten Testdatensätzen wie CIFAR10, muss für den ersten Wert in der letzten Tupel eine durch die Anzahl der Klassen teilbare Zahl eingetragen werden. Bei zehn Klassen könnten es (5,1,3), beim Gestendatensatz (3,1,3) sein.

Der Rest kann so verbleiben. Es reicht, wenn das *Synthesize.ipynb* bis zum Schritt der Hardware Generierung kommt. Alles danach ist nicht wichtig. Vor der Ausführung des Notebooks empfiehlt es sich auf dem Hostsystem das Verzeichnis */tmp/finn_dev_\${USER}/* zu entleeren, um den Überblick beizubehalten. Danach kann das Notebook angestartet werden. Der Prozess bis zur Hardware Generierung kann mehrere Stunden beanspruchen. Danach sollten im erwähnten Verzeichnis unter Anderem drei Verzeichnisse in dem Format */tmp/finn_dev_\${USER}/vivado_stitch_proj_** abgelegt sein. Nur diese sind für den weiteren Verlauf von Interesse. Zusätzlich wird das *vehiclepl*-Repository benötigt. Nach dem Klonen des Repositories ist *vehiclepl/ip_repo/finn/* zu entleeren und die drei *stitch*-Verzeichnisse dort hin zu kopieren.

```
git clone https://gitlab.com/soc-nn/vehiclepl.git
rm -rf vehiclepl/ip_repo/finn/*
cp -r /tmp/finn_dev_${USER}/vivado_stitch_proj_* vehiclepl/ip_repo/finn/
```

Listing 6: vehiclepl Repository Klonen und stitch-Verzeichnisse reinkopieren

Für das weitere Verfahren wird eine Vivado 2020.1 Installation benötigt (z.B. in */opt/Xilinx/*). Zudem sollten die *PYNQ-Z1* Board-Dateien im Installationsverzeichnis unter */opt/Xilinx/Vivado/2020.1/data/boards/board_files/* eingefügt werden.

```
cd /opt/Xilinx/Vivado/2020.1/data/boards/board_files/
wget https://github.com/cathalmccabe/pynq-z1_board_files/raw/master/pynq-z1.zip
unzip pynq-z1.zip
rm pynq-z1.zip
```

Listing 7: PYNQ Z1 Board Files installieren

Nun kann die Projektdatei *vehiclepl/src/vehicle.xpr* in Vivado geöffnet werden.

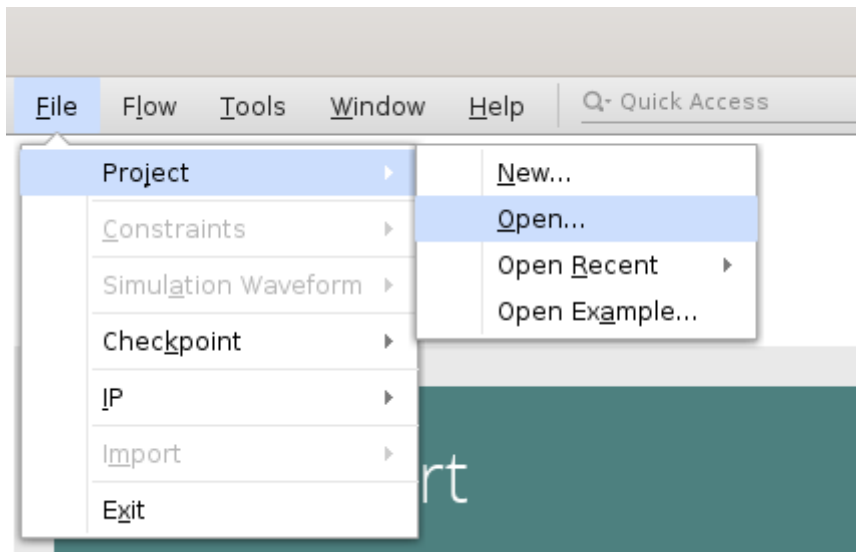


Abbildung 52: Projekt öffnen

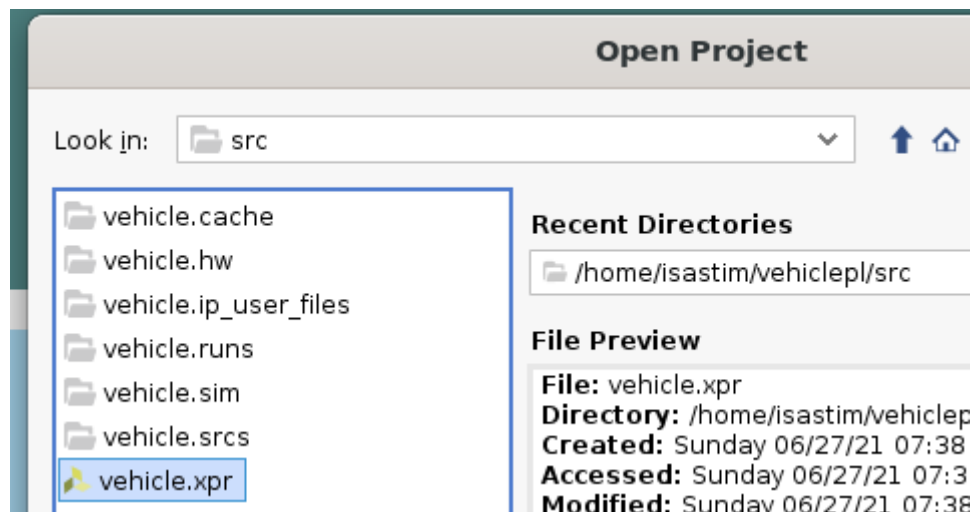


Abbildung 53: vehiclepl/src/vehicle.xpr

Nach dem Öffnen sollte auf *“Open Block Design“* geklickt werden. Hier sind einige Ausgaben zu betrachten.

- Das Block Design an sich sollte ungefähr wie in Abbildung 55 aussehen. Insbesondere sollten die drei IP-Blöcke *finn_dma*, *finn* und *finn_odma* wie in Abbildung 57 vorhanden sein. Um die geht es nämlich in den nächsten Schritten.
- Oben sollte eine Meldung erscheinen, dass neue Blöcke erkannt wurden.

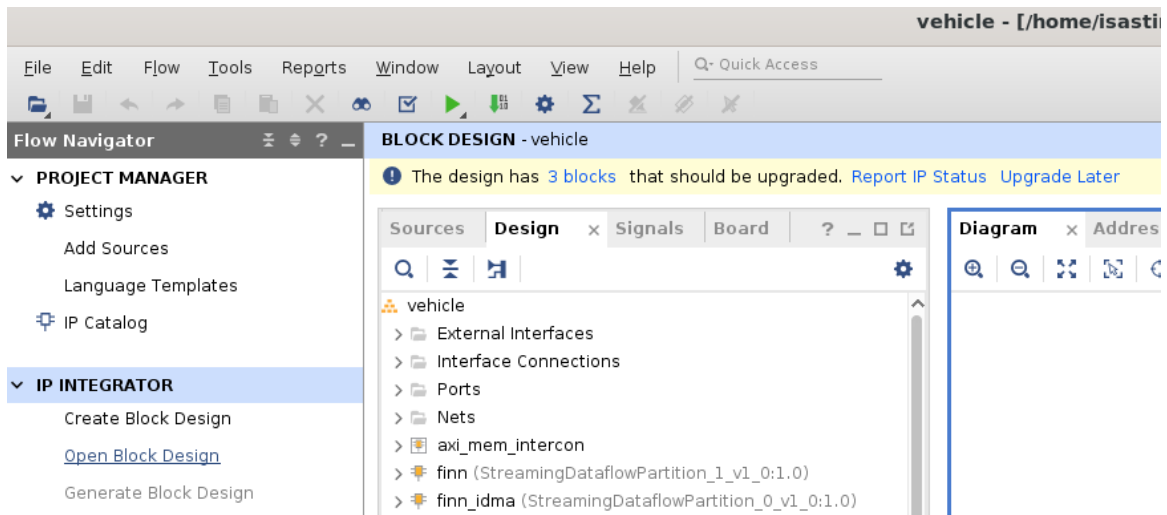


Abbildung 54: Block Design öffnen

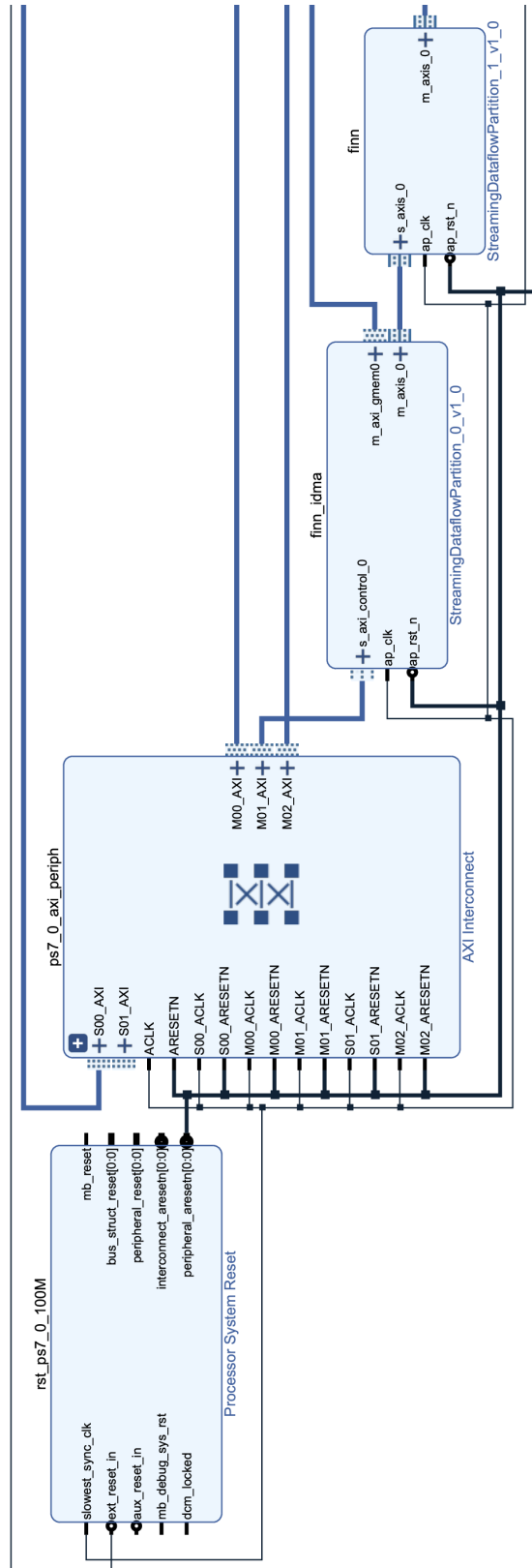


Abbildung 55: Gesamtansicht des Block Design Teil 1

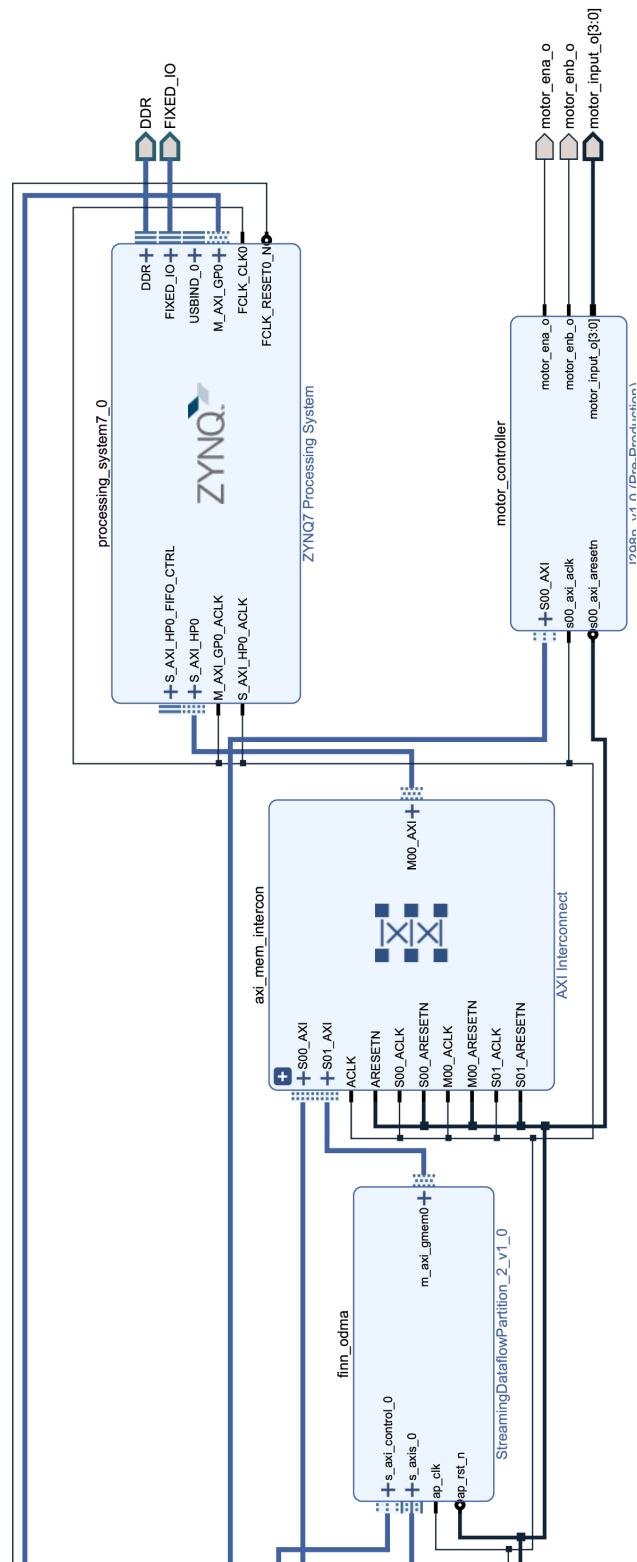


Abbildung 56: Gesamtansicht des Block Design Teil 2

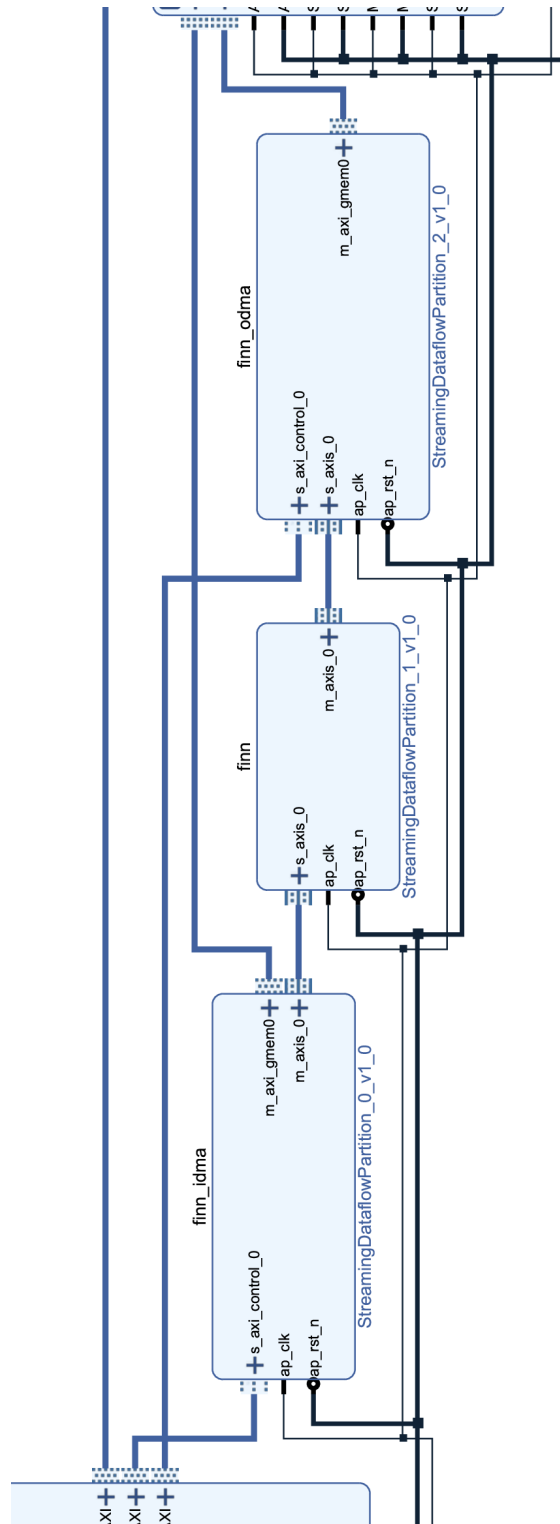


Abbildung 57: Relevante IP-Blöcke

Falls alles ungefähr wie in den Abbildungen erscheint, kann auf *“Report IP Status geklickt werden“*. Im unteren Panel sollten im *“IP Status“* Reiter nur die drei erwähnten IP-Blöcke aufgelistet sein (siehe Abbildung 58). Danach kann auf *“Upgrade Selected“* geklickt werden und die folgenden Meldungen

bejaht werden bis gefragt wird, ob die Hardware generiert werden soll. Hier sollte vorerst auf "skip" geklickt werden.

Source File	IP Status	Recommendation	Change Log	IP Name	Current V
vehicle (3)					
/finn	Incompatible IP data detected	Upgrade IP		StreamingDataflowPartition_1_v1_0	1.0 (Rev.
/finn_odma	Incompatible IP data detected	Upgrade IP		StreamingDataflowPartition_2_v1_0	1.0 (Rev.
/finn_idma	Incompatible IP data detected	Upgrade IP		StreamingDataflowPartition_0_v1_0	1.0 (Rev.

Abbildung 58: IP-Status

Es fehlen nämlich noch die Master Base Adressen für die *finn_idma* und *finn_odma* Blöcke. Hierzu muss der Address Editor geöffnet werden (siehe Abbildung 59). In der darauf folgenden Ansicht sollten die beiden Einträge mit fehlenden Adressen befindlich sein (siehe Abbildung 60).

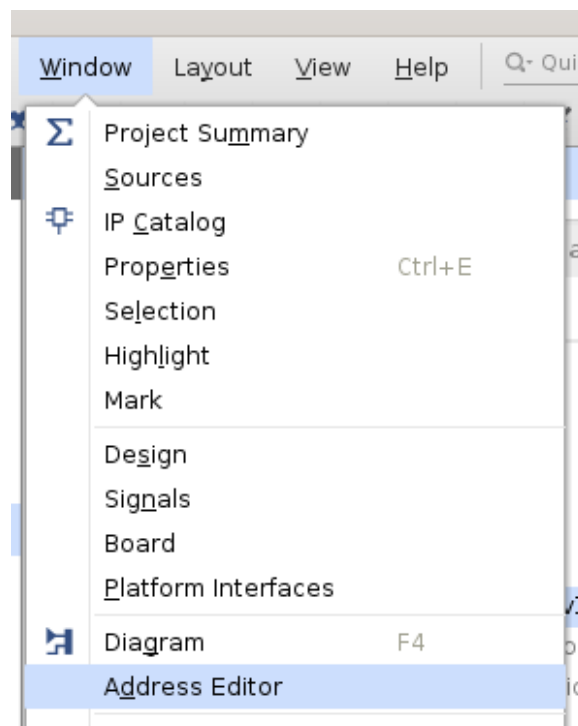


Abbildung 59: Address Editor öffnen

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/finn_idma					
/finn_idma/m_axi_gmem0 (64 address bits : 16E)					
/processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000_0000_0000 ⁰	512M	0x0000_0000_1FFF_FFFF
/finn_odma					
/finn_odma/m_axi_gmem0 (64 address bits : 16E)					
/processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000_0000_0000 ⁰	512M	0x0000_0000_1FFF_FFFF
Network 1					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/motor_controller	S00_AXI	S00_AXI_reg	0x43C2_0000	64K	0x43C2_FFFF
Unassigned (2)					
/finn_idma	s_axi_control_0	Reg			
/finn_odma	s_axi_control_0	Reg			

Abbildung 60: Unzugewiesene Adressen

Mittels eines Rechtsklicks können den Blöcken einzeln Adressen zugewiesen werden (siehe Abbildung 61). Womöglich werden automatisch die richtigen Adressen für die beiden Blöcke erkannt, jedoch sollte sicherheitshalber geprüft werden, dass diese auch wie in Abbildung 62 korrekt sind.

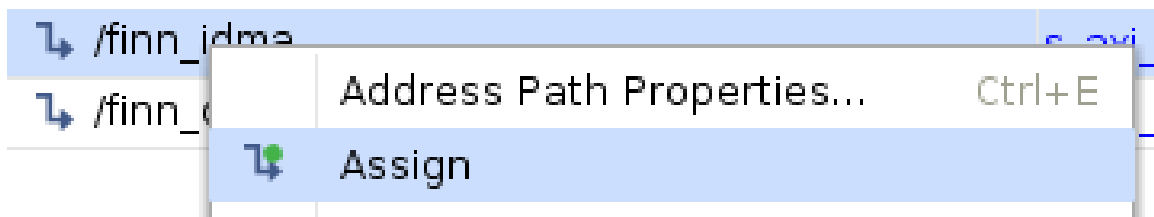


Abbildung 61: Adresse zuweisen

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/finn_idma					
/finn_idma/m_axi_gmem0 (64 address bits : 16E)					
/processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000_0000_0000 ⁰	512M	0x0000_0000_1FFF_FFFF
/finn_odma					
/finn_odma/m_axi_gmem0 (64 address bits : 16E)					
/processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000_0000_0000 ⁰	512M	0x0000_0000_1FFF_FFFF
Network 1					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/finn_idma	s_axi_control_0	Reg	0x4000_0000	64K	0x4000_FFFF
/finn_odma	s_axi_control_0	Reg	0x4001_0000	64K	0x4001_FFFF
/motor_controller	S00_AXI	S00_AXI_reg	0x43C2_0000	64K	0x43C2_FFFF

Abbildung 62: Korrekt zugewiesene Adressen

Nun kann links auf "Generate Bitstream" geklickt werden. Es können alle Meldungen bejaht werden. Der Prozess kann länger dauern. Oben rechts im Vivado Fenster sollte der Status stehen. Nach diesem Schritt sollte die Übersetzung abgeschlossen sein. Von den generierten Dateien sind nur folgende zwei relevant:

- `vehiclepl/src/vehicle.runs/impl_1/vehicle_wrapper.bit` und
- `vehiclepl/src/vehicle.srscs/sources_1/bd/vehicle/hw_handoff/vehicle.hwh`

K Training in Microsoft Azure

Geschrieben von Stanislav Voytas

1. Einleitung

Das Training auf einer herkömmlichen CPU kann bei einer komplexen Netztopologie sehr viel Zeit in Anspruch nehmen. Mit einer GPU kann dieser Vorgang wesentlich beschleunigt werden. Hierbei stößt man in einer Gruppenarbeit zwangsweise auf das Problem, das nicht jeder Studierende über eine leistungsstarke GPU verfügt.

Die aktuelle gemeinsame Entwicklungsumgebung lässt sich außerdem nur auf Ubuntu einrichten. Da die meisten Teilnehmer dieses Betriebssystem nicht vorinstalliert haben, entsteht dadurch automatisch die Anforderung dieses entweder auf dem eigenen Rechner zu installieren oder in einer VM. In einer VM kann die GPU jedoch nicht verwendet werden, da diese in den meisten Fällen nicht an die VM zur Nutzung freigegeben werden kann.

Um jedem Teilnehmer zu ermöglichen eine GPU in einer einheitlichen Entwicklungsumgebung zu verwenden ohne selbst eine zu besitzen wurde beschlossen das Training in die Cloud zu verschieben, wo Hardware Ressourcen kurzzeitig genutzt werden können.

Als Cloud Plattform wurde nach einer Recherche Microsoft Azure gewählt, als Student erhält man dort 100€ Startguthaben welches für die Ziele des Projekts frei genutzt werden kann ohne weitere Kosten zu verursachen.

In dieser Anleitung werden alle nötigen Schritte beschrieben um mit dem Training in der Cloud zu beginnen.

2. Einen Azure Account erstellen

Um die Cloudressourcen von Azure zu verwenden, wird zunächst ein Microsoft Azure Account für Studenten benötigt. Diesen kann man hier mit einem Klick auf "Kostenlos starten" anlegen: <https://azure.microsoft.com/de-de/free/students/>

Man wird aufgefordert sich mit einem Microsoft Konto einzuloggen oder ein neues anzulegen.

***Wichtig:** An dieser Stelle noch nicht die HS-Bremen E-Mail verwenden. Jeder studierende hat zwar bereits automatisch ein Microsoft Konto und kann sich mit seiner HS-Bremen E-Mail einloggen, jedoch wird dieser Account-Typ für Azure nicht unterstützt.*

Es folgt eine Identitätsprüfung per Telefon. Hier muss man seine Handynummer eintragen um eine Bestätigungs-SMS oder Anruf zu erhalten.

Schüler-/Studentenverifizierung ^

Verifizierung der Bildungseinrichtung erforderlich

Das Konto, bei dem Sie angemeldet sind, wurde noch nicht für den Zugriff auf die Angebotsvorteile verifiziert. Verwenden Sie zur Verifizierung das folgende Formular.

Überprüfungsmethode

Schul- oder Uni-E-Mail-Adresse v

Geben Sie Ihre Schul- oder Uni-E-Mail-Adresse ein. Wenn Ihre Bildungseinrichtung in unserer Datenbank enthalten ist, senden wir Ihnen per E-Mail einen Link zur Verifizierung.

Ihr Schul- oder Uni-E-Mail-Adresse wird nur zur Verifizierung verwendet. Verwenden Sie für alle anderen Zwecke die E-Mail-Adresse Ihres Microsoft-Kontos.

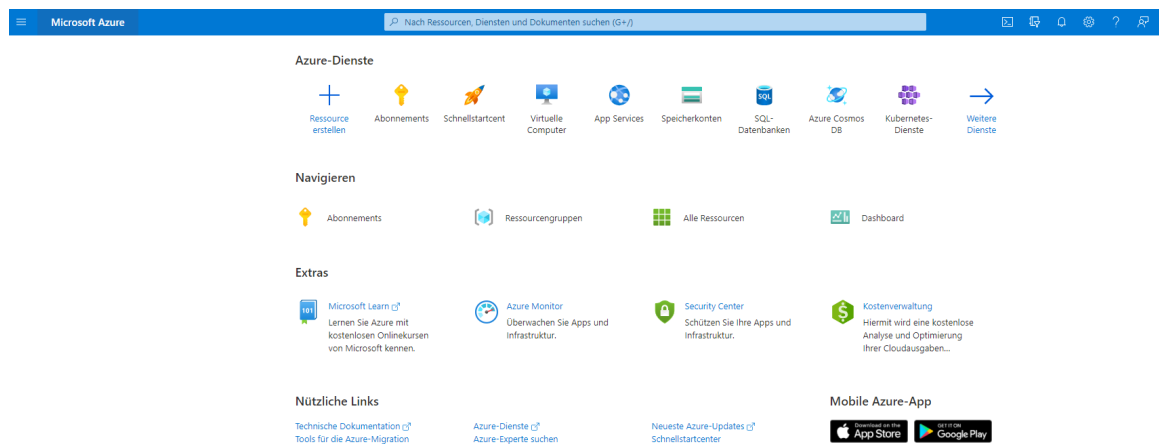
Schul- oder Uni-E-Mail-Adresse

Schul- oder Uni-E-Mail-Adresse erneut eingeben

Akademischen Status verifizieren

Nach der erfolgreichen Überprüfung folgt eine Studentenverifizierung, bei der nun die HS-Bremen E-Mail angegeben werden muss.

Man bekommt dann einen Bestätigungslink über den die Registrierung abgeschlossen werden kann. An dieser Stelle müssen nochmal die Daten wie Name, E-Mail und Telefonnummer angegeben werden. Nach dem Absenden der Daten landet man im **Azure Portal**:



3. Machine Learning Ressource erzeugen

Die Erstellung und Bedienung aller Cloud Ressourcen kann auf drei Wegen stattfinden:

1. Azure CLI
2. PowerShell
3. Azure Portal

In dieser Anleitung wird zur Veranschaulichung die Web GUI Azure Portal verwendet. Diese ist unter <http://portal.azure.com/> zu erreichen.

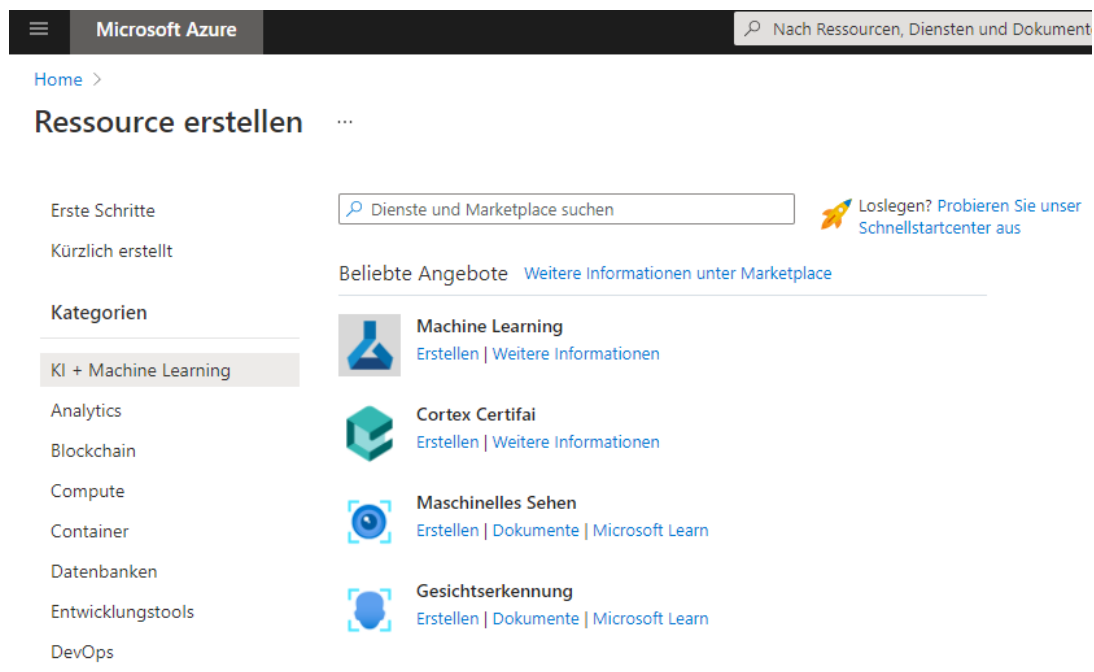
Die anderen beiden Methoden können aber bei Interesse selbstverständlich ebenfalls verwendet werden. Mehr dazu wird in den Lern-Modulen von Microsoft beschrieben: <https://docs.microsoft.com/en-us/learn/modules/management-fundamentals/>

3.1 Arbeitsbereich erstellen

Nachdem man sich im Azure Portal eingeloggt hat klickt man auf "Ressource erstellen":



Es öffnet sich eine neue Seite auf der man links im Menü die Kategorie "KI + Machine Learning" auswählt. Dann klickt man unter "Machine Learning" auf "Erstellen".



Nun kann die Ressource konfiguriert werden.

Abonnement sollte per Default auf "Azure für Bildungseinrichtungen" eingestellt sein. Dieses Abonnement wurde in Kapitel 2 durch die studentische Verifizierung automatisch angelegt und enthält 100€ Startguthaben.

Bei dem Punkt **Ressourcen Gruppe** klickt man auf den Link "Neues Element erstellen" und gibt der Ressourcengruppe einen sinnvollen Namen. Eine Ressourcengruppe ist als Projektordner zu verstehen, der sämtliche inhaltlich zusammengehörenden Cloud-Ressourcen (VMs, Datenbanken, Appsm ML-Workspaces) gruppiert. Ein guter Name wäre z.B. SoC-NN.

Unter **Arbeitsbereichsname** vergibt man einen sinnvollen Namen für unseren neuen Arbeitsbereich

[Home](#) > [Ressource erstellen](#) >

Machine Learning

ML-Arbeitsbereich erstellen

Grundlagen Netzwerk Erweitert Tags Überprüfen + erstellen

Projektdetails

Wählen Sie das Abonnement aus, um bereitgestellte Ressourcen und Kosten zu verwalten. Verwenden Sie Ressourcengruppen wie z. B. Ordner zum Organisieren und Verwalten all Ihrer Ressourcen.

Abonnement * ⓘ	<input type="text" value="Azure für Bildungseinrichtungen"/>
Ressourcengruppe * ⓘ	<input type="text" value="SoC-NN (neu)"/> Neues Element erstellen

Details zum Arbeitsbereich

Geben Sie den Namen und die Region für den Arbeitsbereich an.

Arbeitsbereichsname * ⓘ	<input type="text" value="SoC-NN-ML"/>
Region * ⓘ	<input type="text" value="Nordeuropa"/>
Speicherkonto * ⓘ	<input type="text" value="(neu) socnml0337729697"/> Neu erstellen
Schlüsseltresor * ⓘ	<input type="text" value="(neu) socnml0233145527"/> Neu erstellen
Application Insights * ⓘ	<input type="text" value="(neu) socnml5166459917"/> Neu erstellen
Containerregistrierung * ⓘ	<input type="text" value="Keine"/> Neu erstellen

[Überprüfen + erstellen](#)

[< Zurück](#)

[Weiter: Netzwerk](#)

Der Arbeitsbereich ist zu verstehen als eine Sammlung von allen für Machine Learning nötigen Elemente an einem Ort: Code, Jupyter Notebooks, Computing Ressourcen, Datasets, Modelle etc. Ein sinnvoller Name könnte also sein: "SoC-NN-ML".

Nachdem ein Arbeitsbereichsname eingetragen wurde bekommen die Punkte **Speicherkonto**, **Schlüsseltresor**, **Application Insights** einen Default Wert. Dieser kann so belassen werden.

Als Region sollte **Nordeuropa** gewählt werden. Hier wird spezifiziert in welcher Geografischen Zone die Server liegen sollen auf denen unsere Ressourcen angelegt werden.

Anschließend kann man alle anderen Werte auf den Default-Werten belassen und auf "Überprüfen + erstellen" klicken. Es wird dann eine Validierung der eingegebenen Werte vorgenommen. Diese sollte im Normalfall erfolgreich durchlaufen woraufhin die Ressource erstellt werden kann.

[Home](#) > [Ressource erstellen](#) >

Machine Learning

ML-Arbeitsbereich erstellen

✔ Validierung erfolgreich

Grundlagen Netzwerk Erweitert Tags Überprüfen + erstellen

Grundlegende Einstellungen

Abonnement	Azure für Bildungseinrichtungen
Ressourcengruppe	(Neu) SoC-NN
Region	Nordeuropa
Arbeitsbereichsname	SoC-NN-ML
Speicherkonto	(neu) socnml0337729697
Schlüsselresor	(neu) socnml0233145527
Application Insights	(neu) socnml5166459917
Containerregistrierung	Keine

Netzwerk

Konnektivitätsmethode	Öffentlicher Endpunkt (alle Netzwerke)
-----------------------	--

Erweitert

Verschlüsselungstyp	Von Microsoft verwaltete Schlüssel
HBI-Flag aktivieren	Deaktiviert

[Erstellen](#) [< Zurück](#) [Weiter >](#) [Vorlage zur A](#)

Es dauert dann 1-2 Minuten bis die Ressourcen zur Verfügung gestellt werden. Die Seite aktualisiert sich dann automatisch.

3.2 Arbeitsbereich konfigurieren

Zunächst klickt man im Azure Portal links oben auf das Hamburger-Menü und wählt den Punkt Ressourcengruppen. Hier sehen wir die eben erstellte Ressourcengruppe "SoC-NN". Wenn wir diese anklicken sehen wir alle Ressourcen die eben angelegt wurden:

SoC-NN
Ressourcengruppe

Suchen (STRG+ /)

Erstellen Spalten bearbeiten Ressourcengruppe löschen Aktualisieren In CSV-Datei exportieren Abfrage öffnen Feedback In Mobilversion öffnen

Übersicht

Abonnement (Ändern): Azure für Bildungseinrichtungen
Abonnement-ID: a36c2817-e649-4a1b-91ec-e5151c47552f
Tags (Ändern): Klicken Sie hier, um Tags hinzuzufügen

Bereitstellungen: 1 Erfolgreich
Standort: Nordeuropa

Nach einem beliebigen ... Typ == alle Standort == alle Filter hinzufügen

Es werden 1 bis 4 von 4 Datensätzen angezeigt. Ausgeblendete Typen anzeigen Keine Gruppierung

Name	Typ	Standort
SoC-NN-ML	Machine Learning	Nordeuropa
socnml0233145527	Schlüsseltresor	Nordeuropa
socnml0337729697	Speicherkonto	Nordeuropa
socnml5166459917	Application Insights	Nordeuropa

SoC-NN-ML ist unser Machine Learning Arbeitsbereich.

Das Speicherkonto, der Schlüsseltresor und Application Insights sind Default Ressourcen die automatisch angelegt werden, wenn man eine ML Ressource erzeugt.

Wenn man auf die Machine Learning Ressource (in diesem Beispiel "SoC-NN-ML") klickt sieht man folgende Seite:

SoC-NN-ML
Machine Learning

Suchen (STRG+ /)

"config.json" herunterladen Löschen

Übersicht

Ressourcengruppe: SoC-NN
Standort: Nordeuropa
Abonnement: Azure für Bildungseinrichtungen
Abonnement-ID: a36c2817-e649-4a1b-91ec-e5151c47552f

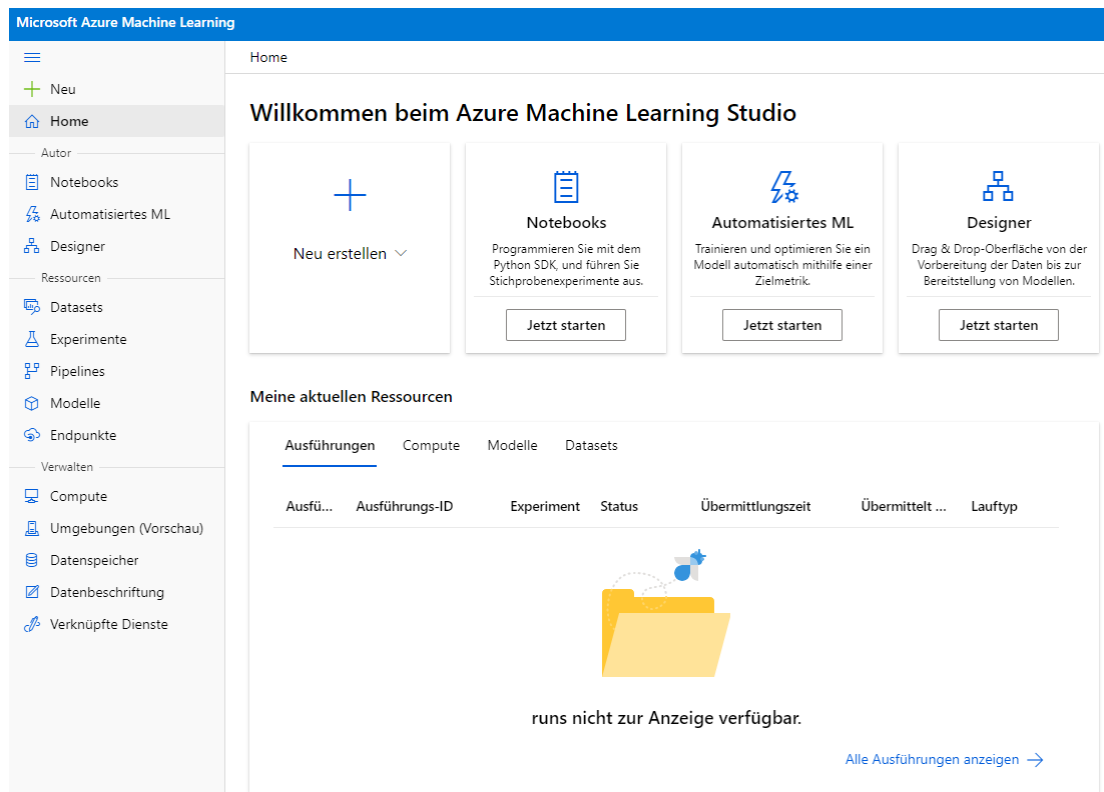
Studio-Web-URL: <https://ml.azure.cc>
Speicher: socnml03377296
Registrierung: ...
Schlüsseltresor: socnml02331455
Application Insights: socnml51664599

Machine Learning-Lebenszyklus verwalten

Verwenden Sie Azure Machine Learning Studio, um Machine Learning-Modelle zu erstellen, zu trainieren, auszuwerten und bereitzustellen. [Weitere Informationen](#)

Studio starten

Mit einem Klick auf "Studio starten" landet man im **Machine Learning Studio**:



Das ist der Bereich in dem wir unser Model trainieren werden und wir sehen hier im linken Menü bereits 2 der wichtigsten Punkte für unser Projekt:

- **Notebooks**
- **Compute**

Diese müssen wir nun mit unseren Daten befüllen.

3.2.1 Compute Ressource erzeugen

Zunächst erzeugen wir eine VM mit einer GPU auf der wir unser Model später trainieren können.

Hierzu klickt man im Machine Learning Studio links im Menü unter "Verwalten" auf "Compute". Nun kann eine neue Compute-Instanz erstellt werden. Hier werden die meisten möglichen Konfigurationen ausgegraut sein, was mit Beschränkungen bei der Nutzung eines kostenlosen Studenten-Accounts zusammenhängt.

Als VM-Typ wählt man GPU und anschließend die einzige mögliche Option "Standard_NC6". Dann vergibt man der VM einen Namen und klickt auf Erstellen.

Erforderliche Einstellungen

Erweiterte Einstellungen

Erforderliche Einstellungen konfigurieren

Wählen Sie den Namen und die VM-Größe zur Verwendung für Ihre Compute-Instanz aus. Hinweis: Das Freigeben einer Compute-Instanz ist nicht möglich, sie kann nur von einem einzigen zugewiesenen Benutzer verwendet werden. Standardmäßig wird eine Compute-Instanz dem Ersteller zugewiesen. Sie können dies im Abschnitt "Erweiterte Einstellungen" ändern und einen anderen Benutzer festlegen.

Computename *

Standort

VM-Typ CPU GPU

VM-Größe Aus empfohlenen Optionen auswählen Aus allen Optionen auswählen

Insgesamt verfügbares Kontingent: 24 Kerne

Name ↑	GPU-Gerät	Workloadtypen	Verfügbares K...	Kos...
<input checked="" type="checkbox"/> Standard_NC5 6 Kerne, 26 GB RAM, 380 GB Speicher	1 x NVIDIA Tesla K80	Testen oder Trainieren von Deep Learning-Modellen mit einem einzelnen Knoten	6 Kerne	0,97 U...

[Vorlage zur Automatisierung herunterladen](#)

Daraufhin wird die VM bereitgestellt, was einige Minuten dauern kann. Den aktuellen Status kann man immer in der Spalte Status in der Übersicht der VM-Instanzen sehen.

Sobald der Status auf "Wird ausgeführt" springt, ist die VM einsatzbereit.

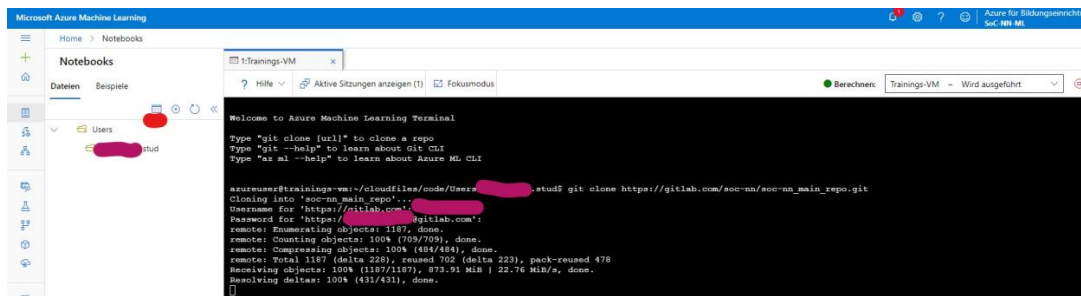
Wichtig: Eine VM in Ausführung kostet den vollen Preis (0,97 USD/h). Da wir hier nur mit einem Studenten-Abonnement arbeiten, wird im Worst-Case das Guthaben von 100€ aufgebraucht und der Account danach nicht nutzbar. Wenn dann in einem bestimmten Zeitraum keine weiteren Schritte (Upgrade zu einer Pay-To-Go Lizenz) vorgenommen werden, werden alle Ressourcen auf diesem Account gelöscht. Es entstehen also auf keinen Fall unkontrollierbare zusätzliche Kosten. Um den unnötigen Guthaben-Verbrauch jedoch vorzubeugen sollte die VM nach der Benutzung immer ausgeschaltet werden. Dies kann in der Übersicht der Compute Instanzen gemacht werden, indem man die VM auswählt und dann auf Beenden klickt.

4. Einrichtung der Projekt-Daten in der Arbeitsumgebung

4.1 Notebook-Daten clonen

Um unsere Dateien aus GitLab in die Arbeitsumgebung zu importieren, verwenden wir Git.

Hierfür klickt man im Machine Learning Studio im linken Menü auf Notebooks und öffnet dann das Terminal der in Kapitel 3.2.1 erstellten Computing-Ressource:



Hier kann das Git Repository geclont werden:

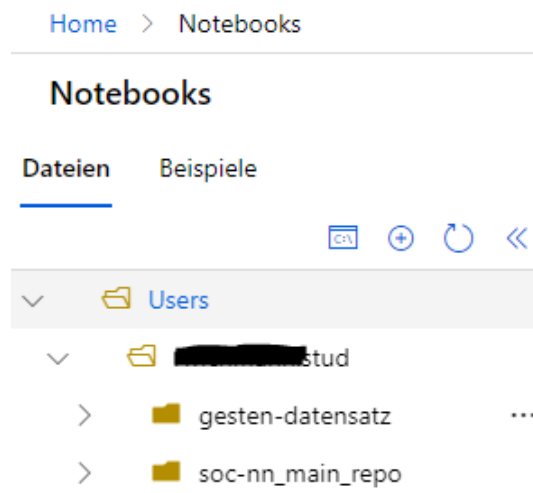
```
git clone https://gitlab.com/soc-nn/soc-nn\_main\_repo.git
```

4.2 Dataset clonen

Ebenfalls hier kann das Dataset Repository geclont werden:

```
git clone https://gitlab.com/soc-nn/gesten-datensatz.git
```

Nach dem clonen beider Repositories sollte die Dateistruktur so aussehen:



4.3 Notebook anpassen

Das Notebook für das Training befindet sich unter "soc-nn_main_repo/src/training/Train.ipynb"

Da das Notebook nun nicht mehr in der vorgesehenen Ubuntu Umgebung läuft müssen ein paar Kleinigkeiten angepasst werden:

1. Pfade im Code anpassen
2. Einzelne Packages nachinstallieren

Der Pfad in der ersten Codezelle muss in der Variable `build_dir` angepasst werden von:
`/home/azureuser/cloudfiles/code/Users/USERNAME/`

zu: `/home/azureuser/cloudfiles/code/Users/USERNAME/gesten-datensatz/`

Wobei `USERNAME` mit dem jeweiligen Usernamen zu ersetzen ist.

Bei der Variablen `data_dir` muss der String `'data'` ersetzt werden zu `'gesture'`.

In der zweiten Codezelle muss bei der Variablen `dataset` muss der String `'Gesture'` ersetzt werden zu `'source'`.

Folgende Pakete müssen bei der ersten Ausführung auf einer neu aufgesetzten VM ausgeführt werden:

- `pip install -e git+https://github.com/Xilinx/brevitas.git@aff49758ec445d77c75721c7de3091a2a1797ca8#egg=Brevitas`
- `pip install torch==1.1.0`
- `pip install netron==4.9.6`
- `pip install torchvision==0.2.2`

Danach muss die Compute-Ressource neugestartet werden.

5. Training

Nachdem die Arbeitsumgebung vollständig eingerichtet wurde, kann das Jupyter Notebook direkt in Machine Learning Studio ausgeführt werden. Hierfür wählt man im linken Menü den Punkt "Notebooks" und öffnet das Notebook unter `"soc-nn_main_repo/src/training/Train.ipynb"`.

Dieses kann nun vollständig ausgeführt werden und nach dem erfolgreichen Training die ONNX Datei im angegebenen build Verzeichnis abgelegt.

Training

```
[9] 1 # from models.cnv import train
    2 from models.cnv import CNV
    3 from utils.notebook_helpers import train
    4
    5 model = CNV(3, 1, 1, 8, 3)
    6 train(model, model_path, dataset.train_loader, device, 1000, 9)
    7 # train(model_path, dataset.train_dataloader, device)
    ✓
```

Started Training:
Finished Training in 3896.734s.

Measure accuracy

```
[10] 1 # from models.cnv import validate
    2 from utils.notebook_helpers import validate
    3
    4
    5 # validate(model_path, dataset.train_dataloader, device)
    6 model = CNV(3, 1, 1, 8, 3)
    7 validate(model, model_path, dataset.test_loader, device)
    ✓
```

21 images. 7 were predicted wrong. Accuracy: 66.66666666666667%

6. Verbliebenes Guthaben überprüfen

Da man als Student vorerst auf 100€ Guthaben pro Jahr begrenzt ist, ist es sinnvoll das aktuelle Guthaben immer im Auge zu behalten. Das Guthaben kann jederzeit unter diesem Link eingesehen werden: <https://www.microsoftazuresponsorships.com/Balance>

7. Weitere Möglichkeiten/Ausblick

7.1 Mehr GPU Power

Wie bereits beschrieben, sind im kostenlosen Studenten-Abonnement die meisten VMs mit leistungsstärkeren GPUs nicht verfügbar. Um diese zu nutzen, muss die ANzahl der so genannten Quotas (maximal nutzbare CPU-Kerne) jeweils für den Account, die Subscription, die Geografische Region und die Ressource selbst erhöht werden. Dies erfolgt durch ein Ticket bei dem Azure Support, in dem die gewünschte Quota-Erhöhung begründet wird. Ein solches Ticket kann jedoch nicht aus der kostenlosen Subscription erstellt werden, bzw. Besteht in dieser kein Anspruch darauf.

Daher muss, wenn eine stärkere GPU benötigt wird, ein neues Pay-As-You-Go Abonnement erstellt werden, für das eine Kreditkarte hinterlegt werden muss, die monatlich mit dem verbrauchten Betrag belastet wird. Mehr dazu kann in der Dokumentation nachgelesen werden:

<https://docs.microsoft.com/en-us/azure/machine-learning/how-to-manage-quotas>

<https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/azure-subscription-service-limits>

7.2 Automatisiertes Aufsetzen der Azure Ressourcen

Um die Schritte der Erstellung und Einrichtung der Ressourcen (Kapitel 3 und 4) zu sparen, kann ein Template dieser Ressourcen erstellt werden. Es würde dann genügen dieses einfach zu importieren um automatisch alle Ressourcen in der richtigen Konfiguration zu erstellen. Näheres dazu kann in der Dokumentation nachgelesen werden: <https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/export-template-portal>

7.3 Eigenes Dataset in der Cloud erstellen

Zwar können wir das Dataset wie bisher einfach in einem Ordner im Dateisystem ablegen und verwenden, Machine Learning Studio bietet aber die Möglichkeit das Dataset komplett neu zu erstellen und separat als Entität zu "lagern" und auch verschiedene Versionen zu verwalten und nach Belieben zu verwenden. Dadurch muss es nicht in einem Git-Repository verwaltet werden, was bei Bild-Daten generell eine fragliche Methode ist.

Man lädt hierfür einfach alle Bilder gemeinsam in Machine Learning Studio hoch und labelt diese anschließend, wobei ein Azure ML Algorithmus beim Label-Vorgang "live" lernt und unterstützung leistet. Mehr Details hierzu sind in der Dokumentation zu finden: <https://docs.microsoft.com/en-us/azure/machine-learning/how-to-create-register-datasets>

8. Quellen

- <https://docs.microsoft.com/en-us/azure/?product=featured>
- <https://docs.microsoft.com/en-us/azure/machine-learning/>

Alle Quellen und Links zuletzt aufgerufen am 08.07.2021 20:17

L Anleitung: Installation der Basisstation

Geschrieben von Tobias Nießen

Diese Anleitung beschreibt die Schritte, welche nötig sind, um die Basisstation in Linux zu kompilieren und auszuführen. Zur Installation der Basisstation in Linux muss zuerst das Repository geklont werden:

```
git clone https://gitlab.com/soc-nn/vehiclebasestation
```

Anschließend muss dieses für das System kompiliert werden:

```
cd ~/vehiclebasestation/  
make
```

Nach Kompilierung kann die Basisstation ausgeführt werden:

```
sudo make run
```

Es öffnet sich ein Fenster, welches auf die Verbindung mit dem Fahrzeug wartet:



Nach Verbindung mit dem Fahrzeug sind folgende Tasten belegt:

Taste	Belegung
M	Manuelles/Automatisches Fahren
Leer	Not-Halt
WASD	Manuell Fahren
R	Video-Auflösung ändern (Runter-Skalierung)

Bekannter Fehler Bei Ausführung der Basisstation kommt es ohne Verbindung mit dem Fahrzeug zu einem Fehler, dass unbekannte Pakete erkannt werden. Dieser Fehler beeinträchtigt die Funktionalität jedoch nicht.

```
Unknown packet type: 02  
Unknown packet type: 02  
Unknown packet type: 02  
Unknown packet type: 02  
Unknown packet type: 02  
Unknown packet type: 02  
█
```