HSB
Hochschule Bremen
City University of Applied Sciences

Neuronale Netze auf einem FPGA zur Sprachkommandoerkennung

Projektbericht zur Lehrveranstaltung Embedded Systems

Jarno Burggräf
Nils Jahns
Henning Kröger
Christian Last
Felix Luther
Christoph Senft
Malte Voigt
Jan Bredereke

8. März 2023

Zusammenfassung

Wir untersuchen am Beispiel einer Sprachkommandoerkennung, wie man ein neuronales Netz in einer stark leistungsbeschränkten Umgebung am besten nutzen kann, indem man ein feldprogrammierbares Gate-Array (FPGA) einsetzt. Das hier beschriebene Projekt implementiert aus Zeitgründen bisher erst Teile dafür. Die Systemarchitektur wurde, aufbauend auf einem Vorgängerprojekt, detailliert ausgearbeitet. Ein Teil der Audiodatenverarbeitung wurde auf dem FPGA realisiert. Die Qualität von Trainingsdaten, die aus dem Vorgängerprojekt übernommen wurden, wurde verbessert, und es wurde ein systematisches Problem beim Aufnehmen von Trainingsdaten identifiziert und behoben. Es wurde die Möglichkeit geschaffen, einfacher verschiedene Strukturen für das neuronale Netz zu erstellen, um damit zu experimentieren. Es wurden bereits viele Schritte für die Implementierung des neuronalen Netzes auf dem FPGA ausgearbeitet; die Integration mit der Audiodatenverarbeitung fehlt allerdings noch. Erste Zeitmessungen zeigen, dass der Einsatz eines quantisierten neuronalen Netzes die Verwendung von Fließkommazahlen bei der Inferenz überflüssig macht, welche ohne eine leistungshungrige Hardware nicht verfügbar sind. Zur Erkennungsrate für Sprachkommandos auf dem FPGA liegen noch keine aussagekräftigen Daten vor.

Inhaltsverzeichnis

1 Einleitung

geschrieben von Jan Bredereke

Dies ist der Projektbericht zum Mini-Projekt in der Lehrveranstaltung „Embedded Systems“ an der Hochschule Bremen im Wintersemester 2022/23. Dieses Kapitel führt zunächst in den Kontext ein und präsentiert dann die Forschungsfrage. Anschließend stellt es die Beispielanwendung und einschlägige vorhergehende Arbeiten an der Hochschule Bremen vor. Das Kapitel schließt mit dem konkreten Projektziel.

1.1 Kontext

Die 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 7 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.

Insbesondere möchte man gerne neuronale Netze nutzen, die üblicherweise in Rechenzentren mit leistungstarker und stromhungriger Spezialhardware eingesetzt werden. Sie können allerdings nicht einfach am Rande der Cloud („Edge“), d.h. nahe bei den Sensoren und Aktoren, oder gar autonom von Daten- und Stromversorgungsverbindungen, auf einem Mikrocontroller eingesetzt werden, denn 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.

1.2 Forschungsfrage

Unsere Forschungsfrage ist, wie man neuronale Netze auf FPGAs am besten nutzen kann, die so leistungsbeschränkt wie strahlungsfeste FPGAs sind. Etwas allgemeiner stellt sich diese Frage entsprechend für neuronale Netze auf FPGAs für Edge-Computing.

1.3 Die Beispielanwendung

Ein autonomes Modellauto als konkretes Vehikel (siehe Abb. 1) dient uns als Stellvertreter für ein autonomes Raumfahrzeug. 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 SoC ist auf einem PYNQ-Z1 Board verbaut, in Abb. 1 an der altrosa („pinken“) Farbe zu erkennen.

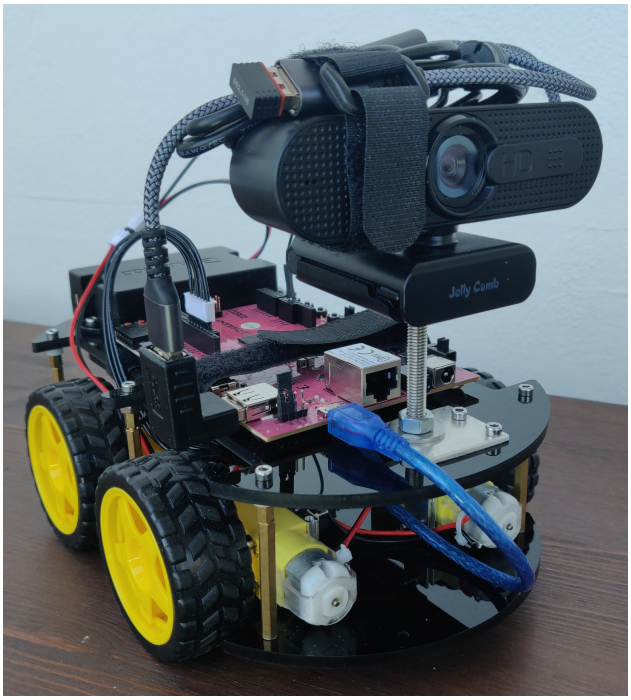

Abbildung 1: Das Fahrzeug aus vorhergehenden Lehrveranstaltungen mit optischer Objekterkennung auf einem FPGA. Foto: Felix Müller

1.4 Vorhergehende Arbeiten an der Hochschule Bremen

Die vorhergehende Lehrveranstaltung „Projekt: SoC-NN – FPGAs für neuronale Netze: Edge Computing auf autonomen Vehikeln“ im Wintersemester 2021/22 sowie einige Lehrveranstaltungen davor [Alt+22; Alt+21; Mül21; Mül+21; Hut+20] realisierten eine Bilderkennung: Das Fahrzeug soll autonom und in Echtzeit eine Person in ihrem Blickfeld erkennen, dieser Person folgen, wenn sie sich bewegt, und Fahrkommandos durch einfache Gesten gehorchen. Der Konferenzbeitrag [Bre22] fasst die Ergebnisse dieser Arbeiten zusammen.

Die vorhergehende Lehrveranstaltung „Embedded Systems“ im Wintersemester 2021/22 [Hag+22] begann im Rahmen des zugehörigen Mini-Projekts, auf dem selben Vehikel eine Fahrsteuerung durch Sprachkommandos statt durch Gesten zu realisieren, ebenfalls autonom und in Echtzeit. Damit stiegen wir neu in den Bereich der akustischen Signale ein. In diesem Gebiet gibt es weniger fertige Lösungen für neuronale Netze als für die Bilderkennung. Dieses Mini-Projekt erstellte ein Konzept, und es produzierte Trainingsdaten, es kam aber auf Grund

der beschränkten Zeit noch nicht weiter. Abbildung 2 auf der nächsten Seite zeigt die bisher entworfene Systemarchitektur aus dem Projektbericht [Hag+22].

Inhaltlich verwandt ist das Projekt „ANN-KNX – Akustische Ansteuerung von KNX-Systemen mit neuronalen Netzen“ im Sommersemester 2022 [Wes+22] und fortgesetzt im aktuellen Wintersemester 2022/23. Es findet auch an der Hochschule Bremen, aber unter der Leitung von Manfred Mevenkamp statt. Dort geht es ebenfalls um Sprachkommandoerkennung mit neuronalen Netzen bei lokaler Verarbeitung. Allerdings werden dort keine leistungsbegrenzten Chips und keine Hardwarebeschleunigung mit einem FPGA sowie keine quantisierten neuronalen Netze (s.u.) eingesetzt.

1.5 Projektziel

Während es im Projekt SoC-NN darum ging, dass das Fahrzeug einfache Arm-Gesten eines Menschen erkennen soll und damit gesteuert werden soll, sollte es nun Ziel sein, dass das Fahrzeug einfache Sprachkommandos erkennt und damit gesteuert wird.

Volle Spracherkennung ist eine komplexe Aufgabe. Sie reicht von der Analog-Digital-Wandlung über die zeitliche Fensterung über die Zerlegung in das Frequenzspektrum per schneller Fouriertransformation (FFT) über eine Filterung des resultierenden Frequenzspektrums mit nachfolgender Meta-Frequenzanalyse (Cepstrum) bis zu einer Mustererkennung mit Hilfe eines akustischen Modells, eines Aussprache-Wörterbuchs und eines Sprachmodells, um schließlich den Text in Schriftform zu erhalten. Dies setzt viel Wissen über den Aufbau von Sprache und auch viel Rechenleistung voraus.

Wir beschränkten uns daher auf das vordere Ende dieser Verarbeitungskette und begnügten uns als Ziel mit mit dem Erkennen von wenigen, einfachen Sprachkommandos für unser Fahrzeug, wie z.B. „links“, „rechts“, „gerade“ und „stopp“. Wesentliche selbst zu entwickelnde Verarbeitungsstufen sind dabei die zeitliche Fensterung, die schnelle Fouriertransformation (FFT) und die Klassifikation von Mustern in deren Ausgabe per neuronalem Netz.

Das Projektziel soll in mehreren Zwischenschritten erreicht werden, von denen auch nur einige erste im aktuellen Semester umsetzbar sein werden. Zunächst werden wir Miniprojekt-eigene PYNQ-Z1 Boards ohne ein Fahrzeug darumherum verwenden. Wir werden die frühen Audio-Verarbeitungsschritte im FPGA implementieren. Wir werden die praktische Erkennungsrate des neuronalen Netzes auf brauchbare Werte verbessern. Das neuronale Netz werden wir zunächst auf der CPU des SoCs realisieren. Das verschiebt die Einarbeitung in das doch recht komplexe Implementieren eines neuronalen Netzes auf einem FPGA auf später. Sobald die Verarbeitungsschritte verstanden und umgesetzt sind, optimieren wir die Verarbeitungsgeschwindigkeit, indem wir z.B. das neuronale Netz auch auf das FPGA bringen. Ebenso integrieren dann wir die Sprachkommandoerkennung in das reale Fahrzeug. Es wird nicht erwartet, dass schon im aktuellen Semester alle diese Schritte vollständig erreicht werden.

Weitere Schritte in Richtung auf ein fertiges System werden später nötig sein. Das Fahrzeug wird zunächst beim Fahren Lausch-Pausen einlegen müssen, um Sprachkommandos verstehen zu können. Möglicherweise müssen wir ein besseres Mikrofon als das auf dem Board eingebaute ergänzen, oder sogar mehrere Mikrofone für eine Richtcharakteristik, um ausreichend Reichweite zu erzielen und um mit Störgeräuschen umgehen zu können.

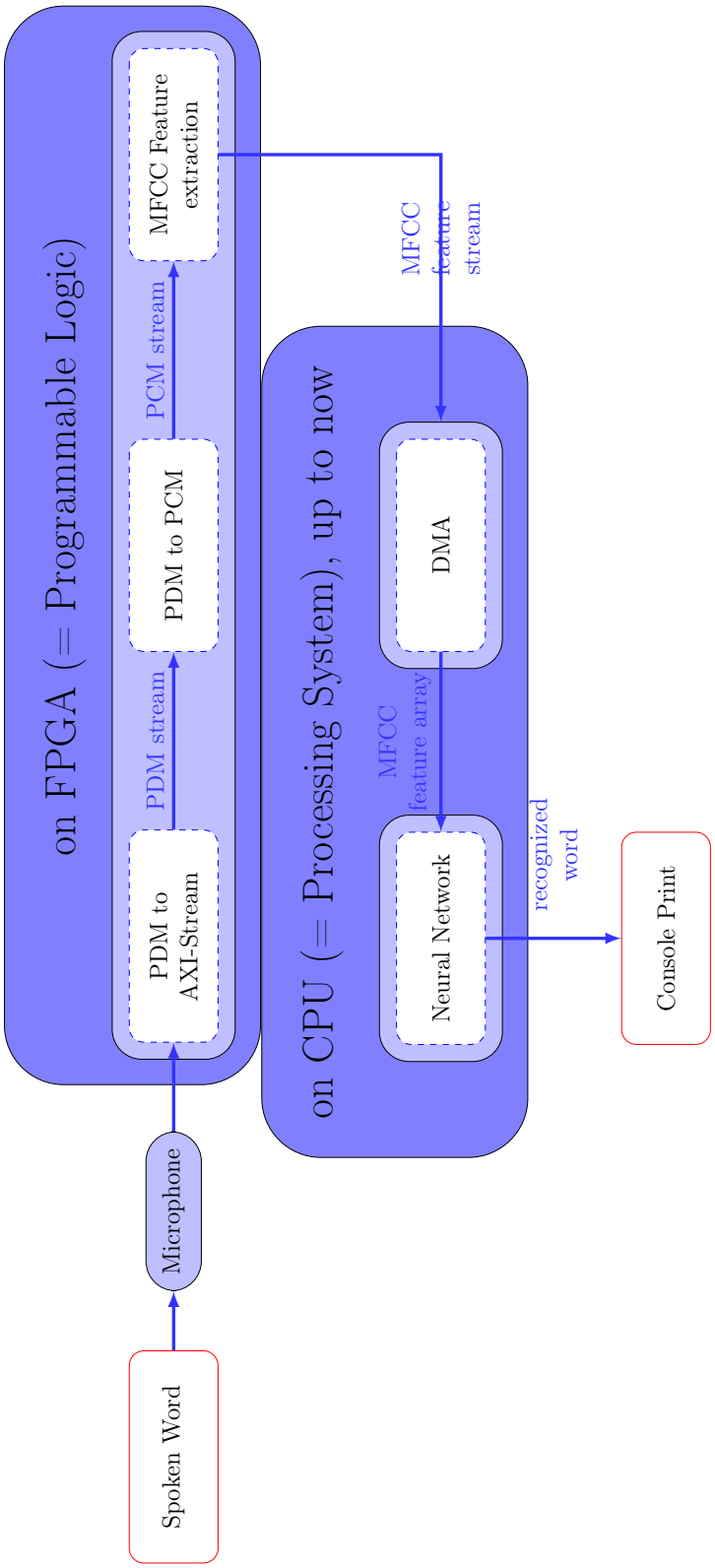

Abbildung 2: Bisher entworfene Systemarchitektur aus [Hag+22] (etwas angepasst)

Übergeordnetes Projektziel soll sein, die Leistung der Signalverarbeitung 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

2.1 Neuronale Netze

geschrieben von Christoph Senft

Neuronale Netze sind eine Art der Daten-Verarbeitung und -Bewertung, in Anlehnung an die Neuronen aus der Biologie.
Sie finden in den verschiedensten Einsatzgebieten Anwendung. Beispiele lassen sich in der Objekterkennung von Bild- und Videodaten, in der Audio Datenverarbeitung oder der Prognose und Optimierung von komplexen System, wie z.B. in der Produktion, finden. Bevor ein Neuronales Netz jedoch in seinem Einsatzgebiet funktionieren kann, muss es erst einmal trainiert werden. Dafür werden Trainings- und Validierungsdaten benötigt, anhand derer eine gewisse sehr spezifische Intelligenz erarbeitet bzw. trainiert und anschließend getestet wird. Somit ist eine Verwendung des Trainierten Netzes nur in seinem sehr spezifischen Bereich und auch nur in seiner spezifischen Aufgabe sinnvoll.
Für diese unterschiedlichen Aufgaben gibt es sehr viele unterschiedliche Netzwerk Architekturen.
In unserem Projekt wurden hauptsächlich 3 neuronale Netztopologien in Erwägung gezogen und im weiteren Verlauf evaluiert. Dabei wurde ein besonderes Augenmerk auf den Nutzen in unserem Projekt gelegt. Die drei Topologien sind das Feat Forward Neural Network (FF), das Recurrent Neural Network (RNN) und das Convolutional Neural Network (CNN). Diese drei sind auch die bekanntesten Netzarten und haben jeweils ihre spezifischen Vor- und Nachteile. 5.1.1

- **Feat Forward Neural Network (FF):** Dieses neuronale Netz ist die einfachste Variante von Netzen. Hier sind alle Knoten aus Ebene N mit allen Knoten aus Ebene N+1 miteinander verbunden und es gibt keinerlei Möglichkeiten im Netz zurückzugehen.
- **Recurrent Neural Network (RNN):** Vom Prinzip her funktioniert diese Variante genauso wie die Erste, nur dass es die Möglichkeit von Rücksprüngen gibt. Notwendig werden diese, wenn es um Daten geht, wo ein Kontext wichtig ist.
- **Convolution Neural Network (CNN):** Diese Netzart funktioniert etwas anders. Sie verwendet Faltungsoperatoren, um Merkmale aus Bildern zu extrahieren und sie für die Kategorisierung zu verwenden.

Aus diesen Funktionsweisen der einzelnen Netze kann später abgeleitet werden, für welchen Einsatzzweck sie am besten geeignet sind und welche Netzart in unserem Projekt am sinnvollsten ist.

2.2 FINN Projekt

geschrieben von Henning Kröger

Das FINN-Projekt ist ein von einer Forschungsgruppe der Firma Xilinx entwickeltes Framework zur Erforschung von quantisierten neuronalen Netzen auf FPGAs. Es umfasst die Python-Library Brevitas, den FINN-Compiler und das Deployment des Netzes mit PYNQ. [Xil23e]

Brevitas dient zum Training quantisierter neuronaler Netze. Es bietet dazu Layer, die in PyTorch im entwurf des Neuronalen Netzes verwendet werden können. Die Dokumentation ist derzeit noch unvollständig (siehe [Xil23a]). Im Code des GitHub-Repositories kann jedoch unter src/brevitas/nn eingesehen werden welche Layer zur Verfügung stehen (siehe [Xil23b]). Unterstützt werden derzeit Feed-Forward Netze, sowie RNNs und LSTMs, wobei die LSTMs eine Python-Datei mit den RNNs teilen. Die Nutzung von Pytorch und Brevitas im Projekt wird in Abschintt 5 auf Seite 21 beschrieben.

Der FINN-Compiler erzeugt unter Verwendung von Vivado-HLS die Beschreibung der Hardware eines Quantisierten Neuronalen Netzes auf einem FPGA, die dann auf einem PYNQ-Board deployt werden kann. Derzeit werden ausschließlich Feed-Forward-Netze unterstützt. Dies ist nicht explizit erwähnt, da auch hier die Dokumentation unvollständig ist, kann jedoch aus den verschidenen Beispielen, die ausschließlich auf Feed-Forward-Netze beschränkt sind geschlossen werden (siehe: [Xil23d]). Genaueres zum FINN-Compiler und dessen Verwendung folgt in Abschnitt 6 auf Seite 30.

2.3 Audiodatenverarbeitung

geschrieben von Felix Luther

2.3.1 PCM

Die Pulse-Code-Modulation ist ein Verfahren zur Umwandlung eines anologen Signals in ein diskretes digitales Signal. Hierbei wird das Signal quantifiziert und kann eine Vielzahl an Werten annehmen, abhängig von der gewählten Datenbreite. Es findet große Anwendung zum digitalen Abspeichern von Audiodaten, auf einer Vielzahl von Speichermedien.

2.3.2 PDM

Pulsedichtemodulation oder auch Deltamodulation stellt ein Verfahren dar, in welchen ein analoges Signal in ein digitales Signal, hier unser Audiosignal nur einen von zwei Pegeln annehmen kann. Das Signal wird somit als Dichte von Pulsen, abhängig von der Amplitude des anlogen Signals dargestellt.

2.3.3 Dezimierung

Dezimierung bezeichnet einen Prozess, mit welchem ein PDM Signal in ein PCM Signal umgewandelt werden kann. Hierbei handelt es sich um eine Variante des Abtastratenkonvertierung, bei welcher die Eingabedaten in eine niedrigere Abtastrate umgewandelt werden, welches die Qualität komprimiert.

2.3.4 WAV

Das WAV(E) Dateiformat wird verwendet um Audiodaten zu Speichern. Es handelt sich hierbei um eine Variante des RIFF (Resource Interchange File Format) Formats. Dieses Format kann rohe PCM Daten enthalten und ist in diesem Fall entsprechend unkomprimiert. Die format-spezifischen Informationen liegen vor den Rohdaten und umfassen im Falle von enthaltenen PCM Rohdaten nur 28 Byte. Eine Spezifikation für das WAVE-Format ist hier zu finden: `https://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html`.

2.3.5 FFT

Die FFT (Fast-Fourier-Transform) stellt eine optimierte Variante der DFT dar. Diese hat im Vergleich zu unoptimierten Varianten der DFT eine Laufzeitkomplexität von $O(n * log(n))$, statt $O(n^2)$. Somit ist sie vor allem bei großen Datenmengen geeigneter, da diese besser skaliert. Bei der Fourier Transformation wird eine Folge an Werten in eine mathematische Funktion umgewandelt, welche den Werteverlauf der initialen Eingangsdaten approximiert.

2.4 Hardware-Architektur

geschrieben von Jarno Burggräf

Wie bereits im Abschnitt 1.3 beschrieben wurde, wird die Signalverarbeitung nicht auf einem klassischen FPGA, sondern einem SoC der Zynq 7000 Familie durchgeführt. Da es Unterschiede in der Anzahl und Taktrate der Prozessorkerne und der Größe des FPGAs gibt, wird im folgenden nur auf den konkret vorhandenen XC7Z020-1CLG400C eingegangen.

2.4.1 Zynq 7000 SoC

Der verwendete SoC vereint einen ARM Cortex-A9 Zweikernprozessor inklusive Peripheriegeräten für Schnittstellen u. a. zu Gigabit Ethernet, SD-Karten, UART und USB 2.0 mit einer programmierbaren Schaltung, die mit Artix-7 FPGAs vergleichbar ist. In den Dokumenten von Xilinx wird der Prozessor inklusive den Peripheriegeräten und Speicher als PS und die programmierbare Logik mit den FPGA-Elementen, zusätzlichem Speicher und seriellen sowie analogen Schnittstellen als PL bezeichnet.[Xil18]

Auf dem Pynq-Z1 wird der Prozessor mit 650MHz betrieben und hat zusätzlich zu den Caches und den im Prozessor verbauten RAM noch 512MB DDR3 externen RAM verbaut. Dies reicht aus, um neben der hardwarenahen Baremetal-Programmierung auch Betriebssysteme wie

Linux einzusetzen. Im von Digilent zur Verfügung gestellten Softwarepaket [1] ist beispielsweise ein unter Linux laufender Webserver enthalten, der genutzt werden kann, um in Jupyter Notebook Python-Skripte auszuführen. [Dig17]

PS und PL können unabhängig voneinander arbeiten, allerdings startet das PS zuerst und konfiguriert danach per Software die PL. Somit ist es nicht möglich, auf dieser Hardware das PS komplett abzuschalten, es wäre jedoch möglich, einen Bootloader zu verwenden, der nur den Bitstream in den FPGA lädt und keine weitere Software ausführt. [Xil18]

2.4.2 Die AXI-Protokollfamilie

Damit das PS die PL nicht nur konfigurieren kann, sondern auch eine Kommunikation stattfinden kann, gibt es mehrere Schnittstellen zwischen PS und PL. Der dabei verwendete Bus ist im ARM AMBA AXI Standard [2] definiert und das PS besitzt mehrere Master- und Subordinate-Schnittstellen. [3] Im PL können diese mit zu Blöcken zusammengefassten Schaltungen, die als intellectual property core (kurz IP-Core) bezeichnet werden, verbunden werden und so anwendungsspezifische Peripheriegeräte implementieren. [Xil18]

Je nach Einsatzzweck können verschiedene Varianten des Busses eingesetzt werden:

- **AXI4** ist der für die PS-PL-Kommunikation verwendete Bus, bei dem ein Teil des Prozessoradressraumes auf die über diesen Bus adressierten Peripheriegeräte abgebildet wird.
- **AXI4-Lite** ist eine vereinfachte Version von AXI4, die weniger schnell, aber leichter zu implementieren ist.
- **AXI4-Stream** verwendet keine Adressen, sondern sendet einen Datenstrom mit beliebiger Länge.

Da der Zynq-Hersteller Xilinx an der Erarbeitung der AXI-Spezifikation beteiligt war und sich das Protokoll für die FPGA-Entwicklung gut eignet, wird das Protokoll von den Entwicklungswerkzeugen gut unterstützt. [Cro+14]

2.4.3 Der Xilinx DMA Controller

Eine auf AXI aufbauende Schnittstelle zwischen PS und PL ist der Direct Memory Access Controller (kurz DMA-Controller). Dieser IP-Core überträgt Daten von AXI4 auf AXI4-Stream und umgekehrt. Diese Übertragung kann in und aus dem RAM des PS erfolgen, ohne dass der Prozessor dauerhaft mit dem Kopieren beschäftigt ist. Stattdess wird eine Übertragung beispielsweise durch den Prozessor konfiguriert und gestartet, vom DMA eigenständig bearbeitet und mit einem Signal vom DMA an den Prozessor beendet. Wenn dabei der DMA-Controller per AXI mit dem Speicher im PS und per AXI-Stream mit einer Schaltung im PL verbunden ist, dient dieser IP-Core auch als Schnittstelle zwischen PS und PL. [Xil22]

[1] Das GitHub-Repository zum PYNQ-Projekt befindet sich unter https://github.com/xilinx/pynq

[2] Die Standards können hier eingesehen werden: https://www.arm.com/architecture/system-architectures/amba/amba-specifications

[3] Master und Subordinate sind die neueren Bezeichnungen für Master und Slave.

3 Systemarchitektur

Dieses Kapitel gibt einen Überblick über die Systemarchitektur, im Hinblick auf den Datenfluss, sowie die verwendeten Tools. Der Datenfluss wird einmal so dargestellt, wie er für später geplant ist, wenn alle Komponenten auf demselben System untergebracht sind, und einmal dem aktuellen Projektstand entsprechend.

3.1 Geplanter Datenfluss

geschrieben von Henning Kröger

Abbildung 3 zeigt das Konzept für den Datenfluss im fertiggestellten System. Dabei finden alle Schritte von der Audioaufnahme, über die Vorverarbeitung, bis zur Inferenz auf dem FPGA statt. Erst bei der Ausgabe der Predictions als numpy-Array-Datei und dem Umgang mit diesen wird die CPU verwendet.

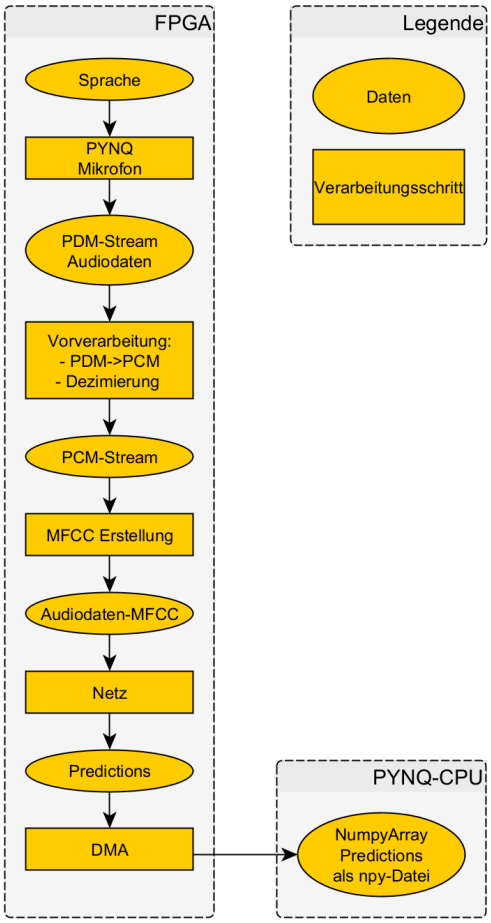

Abbildung 3: Geplanter Datenfluss.

3.2 Aktueller Datenfluss

geschrieben von Henning Kröger

Abbildung 5 auf der nächsten Seite zeigt den Datenfluss des aktuellen Projektstandes. Dabei werden zuerst mit einem PYNQ-Board Audiodaten unter Verwendung des OnBoard-Mikrofons aufgenommen und auf dem FPGA digital vorverarbeitet. Diese werden in der CPU unter Verwendung von Python als .wav-Datei gespeichert, von wo sie manuell kopiert werden, um das Netz an einem externen PC zu trainieren. Dort können auch Daten ins Netz gegeben werden, um Erkennungen durchzuführen um das Training des Netzes zu beurteilen – dies zeigt der Pfad rechts außen. Andernfalls werden die Daten weiterverarbeitet und dann wieder manuell auf das PYNQ mit dem trainierten Netz übertragen. Das Netz läuft dabei noch nicht gleichzeitig auf demselben Board, wie die Audiodatenaufnahme, dieser Schritt steht noch aus. Auf dem zweiten Board können dann Audiodaten mittels Python an das Netz im FPGA übergeben werden. Die Predictions kommen als numpy-Array-Datei zurück.

Abbildung 4 zeigt eine Übersicht über die verwendeten Tools bis zum Deployment beider Boards. Die nötigen Schritte bis zur Aufnahme von Audiodaten erfordern hauptsächlich die Verwendung von Vivado zur erstellung des Overlays, das dann durch ein Jupyter-Notebook mit Python geladen werden kann, wodurch es auf dem FPGA ausgeführt wird. Das Neuronale Netz wird mit den aufgenommenen Trainigsdaten in Pytorch trainiert und dort im ONNX-Format gespeichert. Dann wird es an den FINN-Compiler übergeben, der das Overlay für das PYNQ-Board erzeugt. Dieses wird auch hier mit Python in einem Jupyter-Notebook ins FPGA geladen.

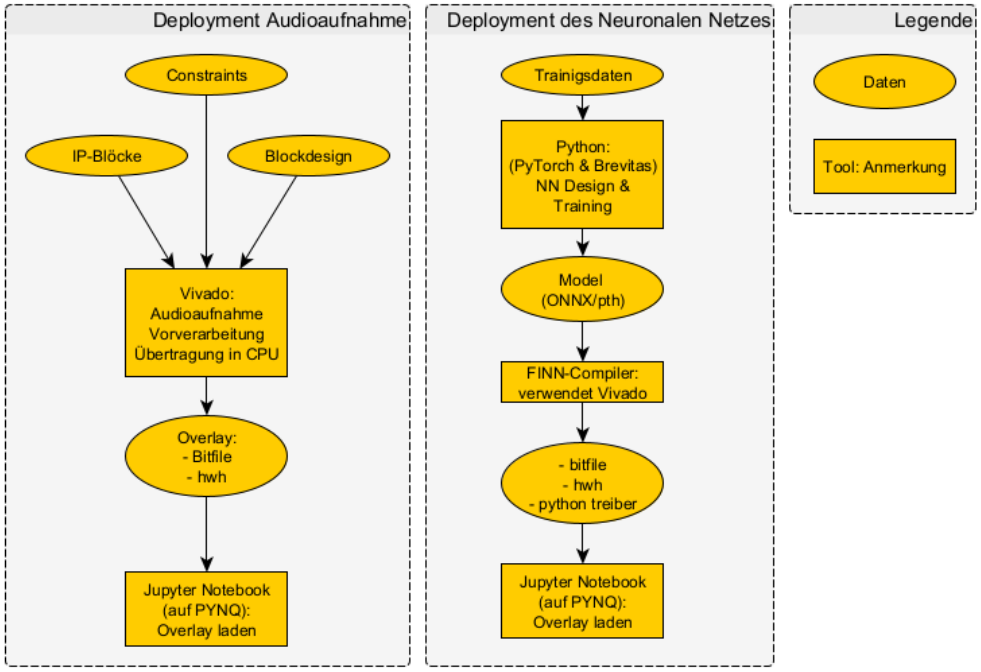

Abbildung 4: Übersicht über verwendete Tools.

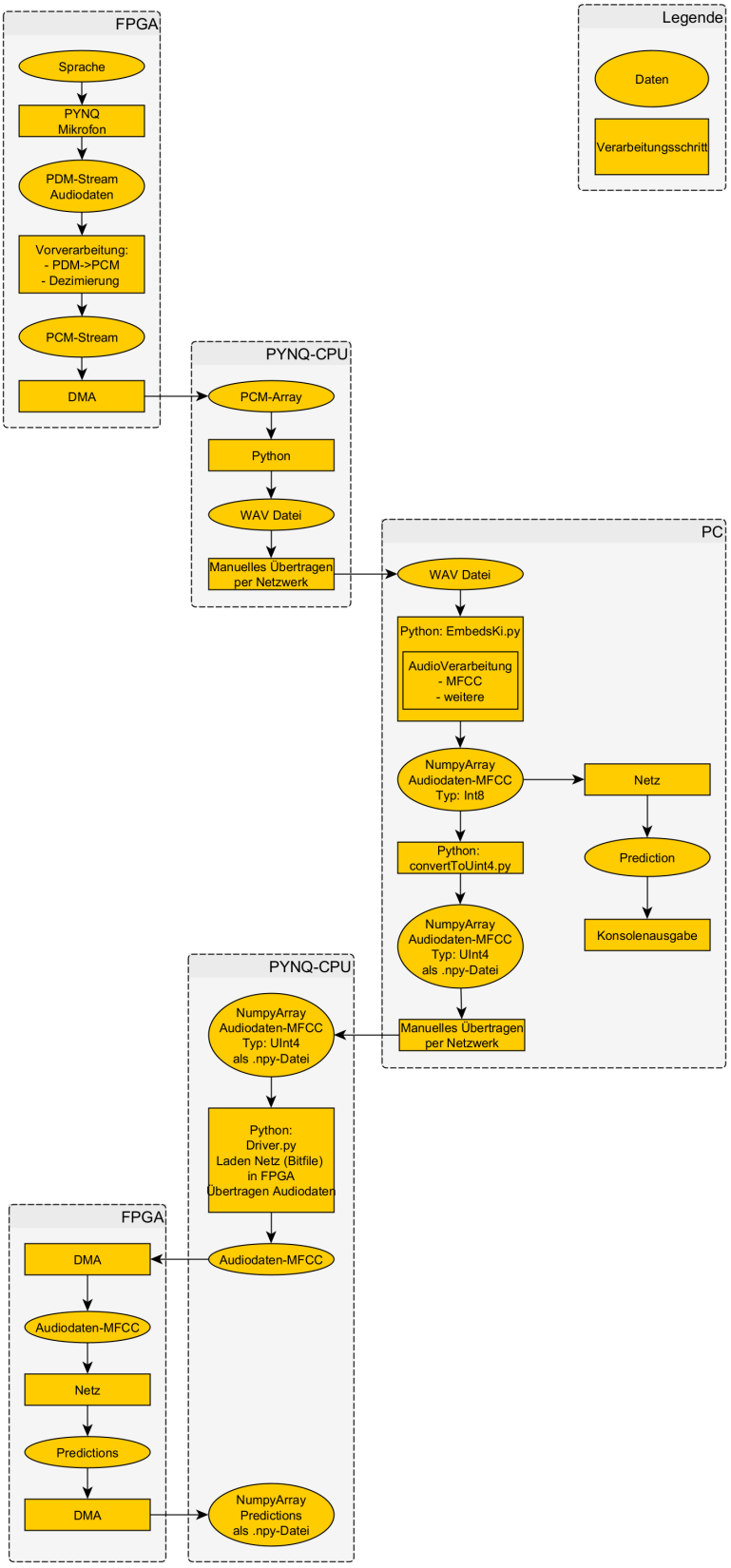

Abbildung 5: Der Datenfluss zum aktuellen Projektstand.

3.3 Ausgewählte Tools

3.3.1 Hardware

geschrieben von Christian Last

Digilent PYNQ-Z1 Board

Als Basis für das Testen bzw. Ausführen des FPGA-beschleunigten Netzes dient ein Digilent PYNQ-Z1 Board (siehe: [Dig23]). Dieses Board besitzt neben einem Xilinx ZYNQ XC7Z020 FPGA unter anderem ein Elektret-Mikrofon für das Erfassen von Audio-Daten. Darüber hinaus ist es mit einem Cortex-A9 Prozessor ausgestattet, auf dem in einer Linux Umgebung ein Jupyter-Notebook Webserver gehostet wird. Letzterer vereinfacht das Programmieren des FPGAs bzw. das Ausführen von Programmen auf diesem. Die PNYQ Linux-Umgebung (Version 3.0.1, siehe: [Xil23f]) wurde auf einer 32GB großen microSD-Karte installiert, von der das Board gestartet wird.

Weitere Hardware

Das Generieren der hardwarebeschleunigten Netze mittels FINN Compiler ist ein sehr rechenintensiver Prozess. Da die Leistung des PYNQ-Z1 Boards zwar für das Ausführen der beschleunigten neuronalen Netze, nicht aber für das Erzugen dieser ausreichend ist, wird darüber hinaus ein entsprechender Computer benötigt (Desktop-PC, Laptop, o.ä.).

Ferner wird für die Kommunikation mit dem PYNQ-Z1 Board via Ethernet ein DHCP fähiger Router benötigt. In unserem Fall kam eine AVM Fritz!Box 7362 SL zum Einsatz, jedoch kann hier jeder generische Router verwendet werden, sofern dieser über einen DHCP Server verfügt.

3.3.2 Software

geschrieben von Nils Jahns

Im Folgenden werden die verwendeten Softwarekomponenten von diesem Projekt aufgelistet:

Python

Python ist eine interpretierte Programmiersprache, die auch häufig Anwendung im Zusammenhang mit Neuronale Netze findet. Python bietet eine Vielzahl von Bibliotheken und Frameworks (TensorFlow, Keras und PyTorch), die das Erstellen und Trainieren von Modellen von Neuronale Netzen stark erleichtern.

Python findet in der Version 3.8.10 in diesem Projekt Anwendung.

PyTorch

PyTorch ist ein quelloffenes Deep Learning Framework, welches von Facebook entwickelt wurde und heute Teil der Linux Foundation ist. Es basiert auf Python und wird für das Erstellen und Trainieren von Neuronalen Netzen verwendet. PyTorch bietet eine Programmierschnittstelle, die es Entwicklern erleichtert, Modelle zu erstellen und zu trainieren.

PyTorch verfügt auch über die Möglichkeit GPU-Beschleunigung zu nutzen, die es ermöglicht, die Modelle auf Grafikkarten zu trainieren, was die Trainingsgeschwindigkeit deutlich erhöht.

Python findet in der Version 1.13.1 in diesem Projekt Anwendung.

Brevitas

Brevitas ist eine Bibliothek von Xilinx, die auf PyTorch aufbaut. Sie wurde speziell für die Erstellung von quantisierten Neuronalen Netzen (QNN) entwickelt. Dazu gehört auch die Unterstützung von QNNs auf FPGAs.

Brevitas findet in der Version 0.8.0 in diesem Projekt Anwendung.

TensorFlow (TensorBoard)

TensorFlow ist ein quelloffenes Deep Learning Framework, das von Google entwickelt wurde. Für diesen Projekt wird allerdings nur ein kleiner Teil - nämlich TensorBoard - von diesem Framework verwendet.

TensorBoard ist ein webbasiertes Werkzeug, das für die Visualisierung von Neuronalen Netzen verwendet werden kann. Es bietet eine Schnittstelle, um Modelle und Trainingsdaten zu visualisieren und zu analysieren und erleichtert damit das Verbesserung und das Verstehen von Modellen.

TensorBoard findet in der Version 2.11.0 in diesem Projekt Anwendung.

Xilinx Vivado

Vivado ist eine Software-Suite, die von Xilinx entwickelt wurde und zur Entwicklung und Implementierung von FPGAs und SoC Designs verwendet wird. Sie bietet eine Reihe von Tools und Funktionen, die den gesamten Entwicklungsprozess vom Design bis zur Implementierung und Inbetriebnahme unterstützt.

Vivado findet in der Version v2022.1 in diesem Projekt Anwendung.

FINN
geschrieben von Christian Last

Der Xilinx FINN Compiler ist ein Werkzeug, das quantisierte neuronale Netzwerke in synthetisierbaren Code übersetzt, um diesen auf FPGAs einsetzen zu können. Er arbeitet mit trainierten Netzwerkmodellen und generiert automatisch FPGA-beschleunigten RTL-Code (Register-Transfer-Level), der zur Programmierung von Xilinx-FPGAs verwendet werden kann. Der FINN Compiler vereinfacht den Einsatz von neuronalen Netzen im Edge-Bereich, wo geringer Stromverbrauch und hohe Leistung kritische Anforderungen sind, und ermöglicht es, hardwarebeschleunigte neuronale Netze zu erstellen, die auf Geschwindigkeit, Energieeffizienz und eine geringe Latenz optimiert sind.

In diesem Projekt kam der FINN-Compiler dev-Branch mit Stand vom 15.01.2023 zum Einsatz (siehe: [Xil23i]).

Weitere Voraussetzungen für die Nutzung der Compiler-Toolchain sind eine Linux-Installation, die Xilinx Vivado Design Suite und Docker. Die folgenden Versionsstände wurden genutzt:

- Ubuntu Linux 22.04 LTS
- Xilinx Vivado Design Suite 2022.1
- Docker 20.10.22

3.3.3 Infrastruktur-Tools

geschrieben von Felix Luther

Git

Git ist eine Software zur Versionskontrolle und erlaubt eine effiziente Arbeit im Team, sie tut dieses ohne Arbeitsobjekte zu blockieren, sollten mehrere Personen an dem gleichen Arbeiten wollen. Es tut dieses durch die Verwendung von Branches (dt. Zweigen) und Pull Requests, in welchen diese zusammengeführt werden.

GitLab

GitLab ist die für das Projekt gewählte Plattform, welche die im Rahmen des Projektes aufgesetzten Repositories hostet. GitLab stellt zudem eine Vielzahl an Funktionen seinen Benutzern zur Verfügung, welche auch in diesem Projekt Anwendung fanden. Hierzu gehören z.B. die Zugriffskontrolle, sodass nur Beteiligte am Projekt Zugriff auf das Projekt haben. Zudem wurde großer Nutzen von der CICD Funktionalität gemacht, welche in diesem Projekt zur Überprüfung der LaTeX Dateien und der folgenden Generierung des Berichts im PDF-Format genutzt wurde. Gruppen in GitLab können verwendet werden um Repositories und die enthaltenen Dateien nur einem bestimmten Personenkreis zur Verfügung zu stellen.

LaTeX

LaTeX ist eine Sprache zur Definition von Dokumenten, welche vor allem in wissenschaftlichen Kreisen oft Anwendung findet. Sie wurde für die Berichterstellung ausgewählte und folgend verwendet. LaTeX stellt den Nutzern viel Funktionalität zur Verfügung und erlaubt es Dokumente zur erstellen, welche den genauen Vorstellungen des Nutzers entsprechen. LaTex ist mithilfe einer Vielzahl an existierenden Bibliotheken stark erweiterbar. Anders als wie bei z.B. Word-Dokumenten ist hier der hinter einem Dokument stehende Code menschlich lesbar und gut verwaltbar. Dieses ermöglicht die Arbeit in der Gruppe mit einem Versionskontrollsystem wie Git.

TeXworks

TeXworks ist die Umgebung, welche in der Gruppe als Standard zur Arbeit am Abschlussbericht gewählt wurde. Diese erlaubt eine effiziente Arbeit mit LaTex und bietet Funktionalitäten wie die Syntax-Hervorhebung, die Einbindung einer Vielzahl an Compilers und auch dem Anzeigen generierter PDF-Dateien. Eine Installationsanleitung ist hier zu finden: A.6.

Docker

Docker ist eine beliebte Software zur Auslieferung von Software. In sogenannten Images kann die Umgebung für das auszuliefernde Software-Produkt definiert werden. Beim Starten eines solchen Images werden die benötigten Abhängigkeiten automatisch beschafft und separat vom Betriebssystem installiert und ausgeführt. Eine hiermit ausgelieferte Software ist somit auf alles Platformen ausführbar, welche Docker unterstützen. Docker findet hier im Projekt bei dem Aufsetzen eines manuellen Runners Anwendung.

TeX Live

TeX Live ist ein Software Paket mit einer Vielzahl an Programmen, welche zur Generierung von Dokumenten aus LaTeX Dateien dienen. TeX Live ist für eine Vielzahl an Betriebssystemen verfügbar und wird als Docker Image verwendet, um die in GitLab definierten Schritten für die CICD Jobs auszuführen.

GlitLab Runner

GitLab Runner ist die Software, welche auf eigenen Maschinen installiert werden kann um aus GitLab heraus ausgelößte CICD Jobs zu verarbeiten. Anders als bei den von GitLab zur Verfügung gestellten Runnern ist für diese keine Zahlung an GitLab notwendig um CICD Jobs auszuführen. Eine Anleitung zum Aufsetzen eines eigenen Runners mithilfe eines Docker Images ist hier zu finden: A.5.

4 Audiodatenverarbeitung

geschrieben von Jarno Burggräf

Das Ziel der Audiodatenverarbeitung ist es, aus der gesprochenen Sprache die Schlagworte zu erkennen. In diesem Projekt bestand das Ziel darin, aufbauend auf den Vorarbeiten Audiodateien mit einer Frequenz von 44,1kHz auf dem FPGA aufnehmen zu können, die dann auf einem PC weiter ausgewertet und zum Training des Neuronalen Netzes verwendet werden können.

4.1 Auslesen des PDM-Mikrofons

geschrieben von Jarno Burggräf

Eine Schaltung zur Ansteuerung des auf dem Pynq-Z1 verbauten Mikrofons vom Typ Knowles SPK0833LM4H-B wurde bereits von der Vorgängergruppe implementiert und beschrieben. [Hag+22] Sie wurde als IP-Core in diesem Projekt weiterverwendet, wobei die folgenden Einstellungen getroffen wurden: Das Mikrofon wird mit einer Frequenz von 3,078MHz ausgelesen und jedes 1-Bit-Sample wird per AXI-Stream ausgegeben. Nach 1543500 Samples, was einer halben Sekunde an Daten entspricht, wird der Frame abgeschlossen und dies mit dem TLAST-Signal dem folgenden Block signalisiert. Für die Aufnahme bedeutet dies außerdem, dass immer ganze Blöcke von 1543500 Samples aufgenommen werden, auch wenn während der Übetragung das `start_recording`-Signal zurückgesetzt wird.

4.2 Umwandlung von PDM zu PCM

geschrieben von Jarno Burggräf

Für diese Umwandlung der Audiosamples von PDM zu PCM wird ein CIC-Filter eingesetzt, der sich effizient im FPGA implementieren lässt und als Low-Pass-Filer sowie als Dezimator dient. Da die Filtereigenschaften des CIC-Filters nicht optimal sind, können diese durch einen zusätzlichen FIR-Filter kompensiert werden.[Liu+20] Letzteres wurde in diesem Projekt jedoch nicht implementiert, da die Qualität der Ausgabe auch mit nur zwei Filterstufen und ohne Kompensierung ausreichte.

Konkret verwendet wurde der CIC Compiler IP-Core von Xilinx in der Version 4.0. Dieser dezimiert die Mikrofonsamples mit dem Faktor 70, sodass die PCM-Samples mit einer Frequenz von 44,1kHz und einer Auflösung von 15 Bits ausgegeben werden. Obwohl intern 15 Bits große Samples generiert werden, wird die Ausgabe immer auf ganze Bytes erweitert, sodass der ausgegeben AXI-Stream eine Auflösung von 16 Bits hat. Analog dazu ist der Eingabevektor 8 Bit groß, obwohl nur die beiden niederwertigsten Bits gelesen werden. Ein `bit_to_unsigned` genannter IP-Core bildet die Ausgabe des Mikrofons auf die richtige Eingabe für den CIC-Compiler ab.

$$\texttt{bit\_to\_unsigned}(x) = \begin{cases} 11_2 = -1_{10} & \text{für } x = 0_2 \\ 01_2 = 1_{10} & \text{für } x = 1_2 \end{cases} \tag{1}$$

4.3 Übertragung der Daten in den Prozessor

geschrieben von Jarno Burggräf

Bei der Übertragung der Daten in das PS wäre es möglich, dass der DMA diese kontinuierlich überträgt. Eine lange Übertragung hätte allerdings zwei Nachteile: Einerseits ist nicht klar, wie weit der Speicher bereits beschrieben wurde und es muss aufgepasst werden, dass keine noch nicht beschriebenen Speicherbereiche ausgewertet werden. Andererseits kann der bereits ausgelesene Speicherbereich nicht stückweise wieder freigegeben werden. Dies würde dazu führen, dass der Speicherplatz schnell komplett belegt wäre und die Aufnahme abbrechen müsste. Damit der DMA die Übertragung zwischendurch unterbrechen und neu starten kann, ohne dass Daten verloren gehen, werden diese zuerst in einem AXI4-Stream Data FIFO zwischengespeichert. Dieser IP-Core kann bis zu 32768 Samples aufnehmen und über ein `prog_full` genanntes Signal dem PS signalisieren, dass eine vorher definierte Anzahl an Samples, in dieser Implementierung 24000, im Zwischenspeicher vorhanden ist. Wenn eine Übertragung über den DMA gestartet wird, sorgt ein weiterer IP-Core `tlast_gen` dafür, dass die Übertragung nach dem Erreichen von hier eingestellten 24000 Samples beendet wird. Somit ist sichergestellt, dass jede DMA-Übertragung ohne Wartezeiten und mit einer festgelegten Anzahl an Samples beendet werden kann, wenn vor dem Starten der Übertragung kein Fehler vorlag und das `prog_full`-Signal gesetzt ist.

4.4 Weitere Verarbeitung in Python und Abspeicherung

geschrieben von Jarno Burggräf, Felix Luther

Das Python-Skript läuft auf dem Pynq-Z1 in Jupyter Notebook. Es lädt den Bitstream, führt das nötige Setup für die GPIO Pins aus, startet die Aufnahme, liest die Audiodaten in Vielfachen von 24000 Samples ein, führt eine kurze Bereinigung der Daten durch und speichert das Resultat als `wav`-Datei ab. Zudem beinhaltet das Notebook mehrere Code-Blocks, welche mithilfe der Python Bibliothek `matplotlib` eine Visualisierung der eingelesenen Audio Daten ermöglicht, ein Abspielen der Audio FIle im Browser ist zudem auch möglich. Diese ist beim Abspielen sehr leise, weswegen auch eine normalisierte Version angelegt wird, was in der finalen Anwendung jedoch nicht verwendet werden sollte, da die Lautstärke ein wichtiges Detail für die weitere Verarbeitung darstellt. [4]

[4]Eine weitere Möglichkeit zur Erhöhung der Lautstärke könnte sein, die Anzahl der CIC-Filterstufen zu erhöhen und so PCM-Werte mit einem größeren Betrag zu erhalten. Dabei muss jedoch beachtet werden, dass bei der im CIC-Compiler bei "Quantization" als Modus nicht "Truncation" ausgewählt wird, da ansonsten die gewonnene Amplitude und Präzision ggf. wieder verloren geht.

Die Ausführung im Browser bietet zusätlich die Möglichkeit, die Aufnahme direkt im Browser abzuspielen und die Daten grafisch darzustellen, was das Nachvollziehen der Daten erleichtert.

4.5 Generieren der MFCCs

geschrieben von Nils Jahns

Mel Frequency Cepstral Coefficients (MFCCs) dienen der Analyse von Sprach- und Audiosignalen. MFCCs liefern eine kompakte (kompakter als beim Mel-Spektrogramm) Darstellung des Frequenzspektrums. Hierfür sind die Koeffizienten verantwortlich. Die theoretischen Hintergründe sind schon in [Hag+22] erläutert.

In Python gibt es mehrere Bibliotheken und Tools, die zur Erstellung von MFCCs verwendet werden können und einen Großteil der Komplexität bei der Erstellung verbergen. In diesem Projekt wird die Bibliothek python_speech_features, wie es auch in den Beispielen [Mra23a] von Xilinx ist, verwendet. Die verwendeten Parameter für die Erstellung entsprechen ebenfalls denen aus dem Beispiel von Xilinx. Die Daten der erstellten MFCCs dienen als Input für das Neuronale Netz beim Training und bei der Erkennung der Kommandos. Zusätzlich wurde noch eine Möglichkeit implementiert sich die Mel-Spektrogramm und die MFCCs zu visualisieren (siehe Abb. 6).

Um die MFCCs zu erstellen, wird zuerst das Audiosignal in numerische Daten konvertiert. Dies geschieht in Python mit der Bibliothek Librosa. Danach können die MFCCs extrahiert werden. [Mra23a]

Noch geschieht die Konvertierung auf einem Desktop-PC. Die Umsetzung auf dem Zynq 7000 SoC sollte ebenfalls problemlos möglich sein, da der Pythoncode sich dort so übernehmen ließe. Für eine Erstellung der MFCCs auf einem FPGA sind allerdings noch weitere Schritte nötig, wie es auch schon in [Hag+22] erwähnt wurde.

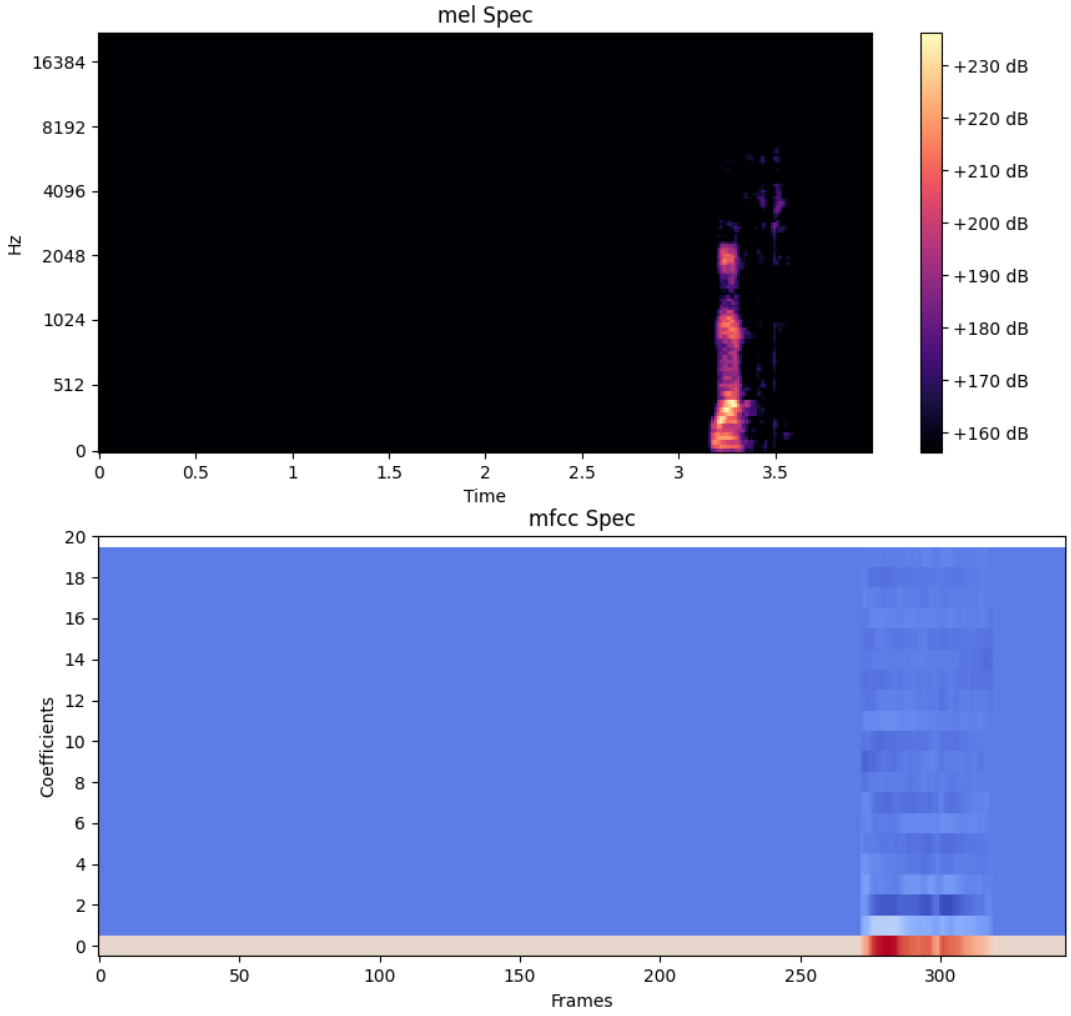

Abbildung 6: Mel-Spektrogramm und MFCC von einem Kommando (*left*)

5 Neuronales Netz

5.1 Wahl eines Neuronalen Netzes

5.1.1 Allgemeines

geschrieben von Chistoph Senft

Es ist nicht gerade einfach neuronale Netze miteinander zu vergleichen, weil es auch immer wichtig ist, wie gut die Trainingsdaten im einzelnen auf die Anwendung zugeschnitten sind. Daher muss versucht werden, aus dem Aufbau des Netzes abzuleiten, für welchen Anwendungszweck das Netz besser geeignet ist als andere. Man könnte natürlich auch mit exakt dem gleichen Datensatz unterschiedliche Netze trainieren, um eine Vergleichbarkeit zu erzielen. Diese Methode war in der Zeit dieses Projektes jedoch nur in einem kleinen Rahmen umsetzbar.
Eine kurze Erklärung wie welches Netz aufgebaut ist, ist in Abschnitt 2.1 zu finden.

	Vorteile	Nachteile
FF	Sehr einfaches Netz	Keine Rückschritte möglich und daher nicht gut für Daten mit einem Zusammenhang verwendbar
RNN	Rückschritte möglich daher gut für Daten mit Zusammenhang	Mäßig komplexes Netz
CNN	Gut für Bildverarbeitung, weil der Filter der Über das Bild läuft 5.1.2 Lokale Merkmale auf einem Bild erkennen kann und diese Bilder beliebig groß sein können. Durch die Pooling-Layer5.1.2 ist es nicht relevant an welcher Stelle das Bildes ein bestimmtes Merkmal zu finden ist.	Komplexes Netz

Aus den oben beschriebenen Vorteilen sticht für unseren Anwendungsfall das RNN als "bestes"Model heraus, da bei Audiodaten immer ein Kontext notwendig ist. Somit würde auch eine längere Aufnahme und Verarbeitung von 1s Ausschnitten ausfallen, was mehr der natürlichen Verarbeitung von Audio entspricht und den Prozessor entlastet. Außerdem schafft das RNN mit einer mittelmäßigen Komplexität eine solche Funktionalität zu bieten.
Als unsere Vergleiche zu einem Ergebnis gekommen sind, welches Netz theoretisch das Beste für unser Projekt sein müsste, ist aufgefallen, dass FINN gar nicht alle Netztypen unterstützt und wir uns somit an die Vorgaben von FINN anpassen mussten, um das Netz überhaupt auf der Hardware laufen zu lassen.

5.1.2 Auswahl zusätzlicher Layer im Neuronalen Netz

geschrieben von Christoph Senft

mit geschrieben von Malte Voigt

Das bereits erstellte Neuronale Netz besteht aus insgesamt 6 Layern von 3 verschiedenen Typen: 3 LinearLayern, 2 ReLu-Layern und einem Batchnormalization Layer als Ausgang. Im Training hat dies schon ganz gute Ergebnisse geliefert, schaut man sich aber den holprigen Trainingsverlauf an, gibt es durchaus noch Luft nach oben. Eine Verbesserung des Trainingsergebnisses erhoffen wir uns also durch die folgenden Layer:

- **Dropout-Layer:** Diese Layer werden von torch bereitgestellt und bewirken eine Abschaltung von einigen Neuronen. Wie viele, gibt man bei der Initialisierung des Layers an. Dies hat eine bessere Generalisierung zu Folge und verringert das Overfitting, da mehr verschiedene Neuronen Zusammenarbeiten müssen, um ein richtiges Ergebnis zu liefern und nicht immer die Selben. [Hin+12]

- **Pooling-Layer:** Da Audio-Erkennung mit Neuronalen Netzen ähnlich gehandhabt wird wie Bilderkennung, indem das Bild eines Spektrogramms dem Netz präsentiert wird, können auch hier die Pooling-Layer verwendet werden. Diese filtern überflüssige Informationen heraus, indem sie sich einen definierten Bereich anschauen und darüber je nach verwendeter Methode das Maximum (MaxPool) oder den Mittelwert (AvgPool) der Werte ermitteln und für die weitere Berechnung des Ergebnis nutzen. Dadurch kann der Output kleiner als der Input sein, aber die wesentlichen Merkmale bleiben erhalten. Sie werden oft zusammen mit Convolution-Layern benutzt.

- **Convolution-Layer:** In einem Convolution-Layer wird ein Filter verwendet, der über das Bild des Spektrogramms bewegt wird. Das Bewegen des Filters wird immer Stück für Stück gemacht, damit eine Überlappung mit dem vorherigen Bildausschnitt gewährleistet ist. Der Filter ist eine Matrix, die mit dem zu bearbeitenden Bild, welches auch in Form einer Matrix vorliegt, multipliziert wird, um zum nächsten Layer zu gelangen. Diese Operation nennt man auch Faltung, so kommt auch der Name des Layers zustande.

- **Batch-Normalization:** Ein Batchnormalization-Layer wird dafür verwendet, das Training des neuronalen Netzes einfacher und stabiler zu machen. Dabei wird eine Methode zum re-centering und re-scaling verwendet (Genaueres unter [Wes+23] Abschnitt 4.2)

- **ReLU-Layer:** Ein ReLU-Layer(Rectified Linear Unit) ist eine ziemlich einfache Art der Neuronen. Sie hat eine nicht lineare Aktivierungsfunktion.

- **Linear-Layern:** Ein Linear-Layer ist die einfachste Art der Neuronen. Sie hat eine lineare Aktivierungsfunktion und ist mit allen Neuronen des nächsten Layers verbunden.

5.1.3 Vergleich DefaultModel mit FINN-Beispiel

geschrieben von Christoph Senft

Das bereits erstellte Neuronale Netz, das hier als DefaultModel angesehen und mit dem Finn Beispiel [Mra23b] verglichen, wird besteht aus insgesamt 6 Layern von 3 verschiedenen Typen: 3 LinearLayern, 2 ReLu-Layern und einem Batchnormalization Layer als Ausgang.

Das Finn Beispiel besteht aus 13 Layern von denen es 4 unterschiedliche Layertypen gibt. Aufgebaut ist das Netz aus einer Wiederholenden Abfolge von Layern: LinearLayer, Batchnormalization, Dropout-Layer, ReLu-Layer. Dieses Muster wiederholt sich 3 mal bis am Ende noch einmal ein LinearLayer als Output angefügt wird.

Was genau die Layer als Hauptaufgabe haben wurde für die spezielleren Beispiele in Abschnitt 5.1.2 schon genauer beleuchtet.

5.1.4 Neue Modelklasse

geschrieben von Malte Voigt

mit geschrieben von Christoph Senft

Zum Erstellen und Trainieren eines Neuronalen Netzes hat unsere Vorgängergruppe eine Klasse geschrieben, in der ein Model mit seinen verschiedenen Layern definiert wurde. Da die Vorverarbeitung der Audio-Daten beim Training allerdings sehr lange dauert und erfahrungsgemäß andere Layer als die Verwendeten eine gute Ergänzung für das Model wäre, bzw. die Möglichkeit verschiedene Models miteinander zu vergleichen eine sinnvolle Erweiterung ist, wurde eine weitere Klasse eingefügt, die die verschiedenen Models beherbergt.
Ein Model wird nun wie folgt definiert:
In der Modelklasse existieren 2 Superklassen, die eine ist `SuperModel`, die andere `SuperModelSeq`. Sinn dieser beiden Superklassen ist es, zwischen Models, deren Layer direkt auf dem Objekt der Klasse sind und denen, deren Layer in einem `Sequential` untergebracht sind, zu unterscheiden. Je nachdem, welche Definition man bevorzugt, kann man diese wählen.
Ein weiterer Vorteil der Superklassen ist, dass man eine `foward` Methode (die das Netz mit Input Daten umgehen soll) für alle Subklassen definieren kann. Diese können von den Subklassen natürlich überschrieben werden, sollte dies nötig sein. Außerdem kann man über die Superklasse alle Subklassen einer Klasse finden und über diese iterieren, bis man die gewünschte Klasse gefunden hat. Dies wurde bereits für die FileLoader Klasse implementiert und jetzt für die Models übernommen.
Um ein neues Model zu erstellen, muss nun nur eine neue Klasse erstellt werden, die von einer der beiden Superklassen erbt und die verschiedenen Layer definiert werden, indem man für jeden Layer eine eigenes Attribut auf der Klasse erzeugt (SuperModel) oder das Attribut `model` mit einem `torch.nn.Sequential` füllt (SuperModelSeq). Zur Auswahl stehen die Layer von `brevitas.nn` und ein paar von `torch.nn`.
Die bereits existierende Klasse `NeuralNetwork` wurde also dermaßen erweitert, dass die Layer nicht mehr direkt auf dem Objekt der Klasse liegen, sondern im Attribut `model` gespeichert

werden. Wird ein neues Neuronales Netz über den Konstruktor erstellt, kann ein Name für das Model mit übergeben werden, welcher der Klassenname des Models in der Modelklasse ist. Wird er das nicht, wird als Standard-Argument `DefaultModel` verwendet und das Model der Vorgängergruppe erstellt. Über den Namen wird dann aus allen Subklassen von SuperModel und SuperModelSeq das Model ausgesucht, dessen Namen dem übergebenen entspricht. Passt keiner, wird ebenfalls das DefaultModel verwendet.
Um verschiedene Versionen eines Models zu erzeugen, etwa um kleine Parameterabweichungen zu testen oder eben verschiedene Versionen eines Models zu haben und sie miteinander zu vergleichen, da die Initialisierung der Gewichtungen zufällig erfolgt und somit kein Model einem anderen gleicht, kann ein Suffix mit einem Unterstrich an den Model-Namen angehängt werden, wie etwa `_V2`.
Beim Training kann nun über den Parameter `-modelName` ein oder mehrere kommaseparierte Model-Namen angegeben werden, um sie alle hintereinander zu trainieren und die Vorverarbeitung nicht für jedes Model einzeln zu machen. Entsteht bei dem Training von einem der Models ein Fehler, wird mit dem nächsten Model fortgefahren, um so wenig Zeitverlust wie möglich zu haben.

5.2 Training des Netzes

geschrieben von Nils Jahns, Malte Voigt

Beim Training des Neuronales Netze konnte auf der Ergebnisse von [Hag+22] aufgebaut werden. Im folgenden werden Änderungen und Erkenntnisse dazu beschrieben.

5.2.1 CPU/GPU Training

geschrieben von Nils Jahns

Grundsätzlich kann man ein Neuronales Netz auf zwei Wege, die sich in ihrer Performanz unterscheiden, trainieren. Das Training erfordert viel Parallelität und basiert auf der Berechnung von Fließkommazahlen [Li+16]. Daher ist das Training auf einer CPU deutlich langsamer als auf einer GPU. Die GPU ist für die parallele Berechnung von Fließkommazahlen bestens geeignet.

Das Training mit einer oder mehrerer CPUs ist bei PyTorch die Standardeinstellung. Um die Beschleunigung beim Training in diesem Projekt nutzen zu können, wurde der Trainingsprozess um die Möglichkeit erweitert, ihn mit GPU-Unterstützung auszuführen. Die dafür nötigen Schritte und Voraussetzungen sind im Repository (A.4) beschrieben.

Zum Vergleich die Trainingszeiten eines NN mit und ohne GPU-Unterstützung:

	CPU	GPU
Hardware:	AMD Ryzen 5 3600 6-Core Processor 3.59 GHz	NVIDIA GeForce RTX 3060
Epochen:	100	100
Dauer:	ca. 1600 Sekunden	ca. 100 Sekunden

Es ist also erstrebenswert die Unterstützung einer GPU zu nutzen, da die Trainingsdurchläufe deutlich verkürzt werden.

5.2.2 Confusion Matrix

geschrieben von Nils Jahns

Um die Trainingsergebnisse besser beurteilen zu können, wurde eine Confusion Matrix als weiteres Instrument zu TensorBoard (siehe 3.3.2) hinzugefügt.

Die Confusion Matrix ist eine statistische Methode zur Bewertung der Prognose eines Neuronalen Netzes. So können die Anzahl der korrekt und der falsch klassifizierten Datenpunkte in jeder Klasse (also für jedes Kommando) visualisiert werden.

Es wurde so umgesetzt, dass alle zehn Epochen eine Confusion Matrix erstellt wird. Abbildung 7 zeigt exemplarisch die Confusion Matrix bevor das Training beginnt. Abbildung 8 zeigt die Confusion Matrix nach 150 Epochen Training.

Links von jeder Zeile steht das jeweilige Kommando, welches dem Netz präsentiert wurde. Die Summe einer Zeile ist die Anzahl an Testdaten für dieses Kommando. Die Beschriftung unter jeder Spalte gibt an, welches Kommando erkannt wurde. So ergibt es sich, dass sich im Laufe des Trainings eine Diagonale bilden sollte, wie es auch in Abbildung 8 zu sehen ist.

Hier zeigt sich dann auch die Stärke dieser Darstellung, da Anomalien schnell erkannt werden können. So wurde bei diesem Trainingsdurchlauf mit fast 10% das Kommando *left* statt *back* erkannt. Diese Auffälligkeit lohnt sich zu untersuchen, was auch getan wurde. Es stellte sich raus, das einige korrupte Audiodateien dafür verantwortlich waren.

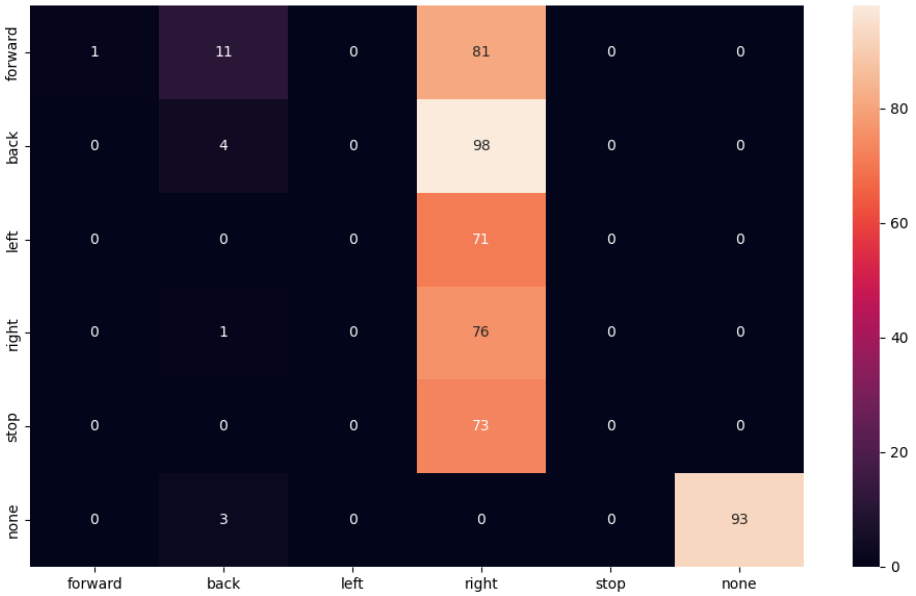

Abbildung 7: Confusion Matrix bei 0 Epochen

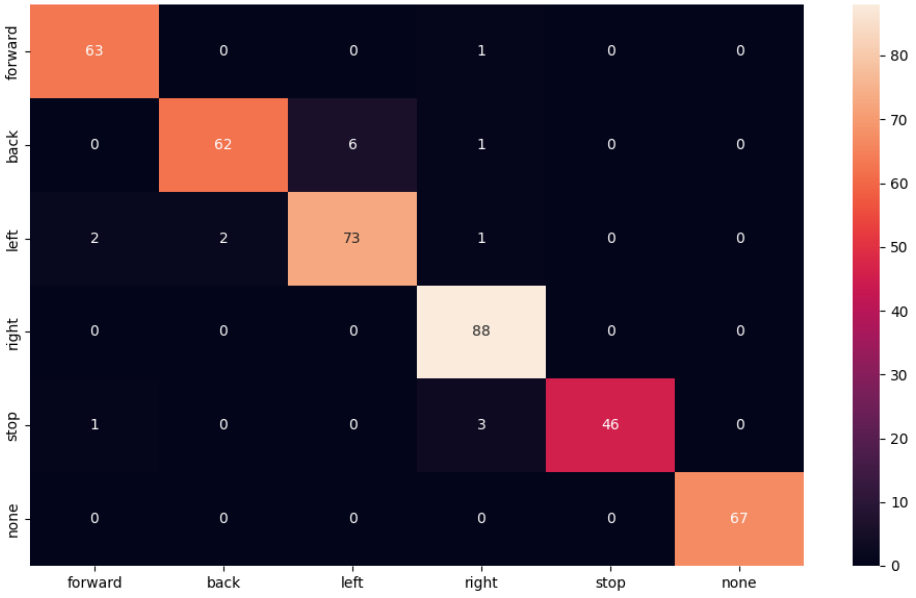

Abbildung 8: Confusion Matrix bei 150 Epochen

5.2.3 Datensatz

geschrieben von Nils Jahns

Der bestehende Datensatz zum Training des Neuronalen Netzes, der zum Großteil vom Vorgängerprojekts übernommen wurde, enthält nun 477 Audiodateien (mp3), die in verschiedene Kategorien eingeteilt sind.

Dazu gehören die fünf gültigen Kommandos, die erkannt werden sollen:

- back (84 Dateien)
- forward (74 Dateien)
- left (69 Dateien)
- right (92 Dateien)
- stop (75 Dateien)

Sowie eine Kategorie für ungültige Kommandos (74 Dateien). Zudem gibt es noch Fahrgeräusche, um die Daten mit Data Augmentation [Ko+15] zu vervielfältigen.

Es hat sich gezeigt, dass einige Audiodateien aus dem Datensatz korrupt waren. Wie im Abbildung 8 zu sehen ist, wird häufig das Kommando *left* erkannt, obwohl das Kommando *back* dem Netz präsentiert wurde. Tabelle 1 zeigt die schlechte Erkennung vom Kommando *back* mit einem zusätzlich aufgenommen Datensatz, der eine Fehlerrate von 50% aufweist.

Audiodatei	Erkennung
BACK - 108.mp3	BACK (100%)
BACK - 109.mp3	BACK (100%)
BACK - 110.mp3	LEFT (91%)
BACK - 111.mp3	BACK (100%)
BACK - 112.mp3	LEFT (100%)
BACK - 113.mp3	LEFT (100%)
BACK - 114.mp3	LEFT (97%)
BACK - 115.mp3	LEFT (100%)
BACK - 116.mp3	BACK (100%)
BACK - 117.mp3	BACK (100%)

Tabelle 1: Übersicht Erkennung vom Kommando *back*

Nachdem die korrupten Audiodateien, die hauptsächlich unverständlich waren, gelöscht und das Netz neu trainiert wurde, zeigte sich, dass die Erkennungsrate (siehe Tabelle 2) sich nun deutlich verbessert hat. Zu den gelöschten Audiodateien gehören:

- BACK - 65.mp3
- BACK - 69.mp3
- BACK - 70.mp3
- LEFT - 19.mp3
- LEFT - 67.mp3
- NEGATIVES - 34.mp3
- STOP - 22.mp3
- STOP - 37.mp3

Audiodatei	Erkennung
BACK - 108.mp3	BACK (100%)
BACK - 109.mp3	BACK (100%)
BACK - 110.mp3	BACK (100%)
BACK - 111.mp3	BACK (100%)
BACK - 112.mp3	BACK (100%)
BACK - 113.mp3	BACK (100%)
BACK - 114.mp3	BACK (100%)
BACK - 115.mp3	BACK (100%)
BACK - 116.mp3	BACK (100%)
BACK - 117.mp3	BACK (100%)

Tabelle 2: Übersicht Erkennung vom Kommando *back* nach der Änderung

5.2.4 Neue Vorverarbeitung

geschrieben von Malte Voigt

Vor dem Training müssen die Daten zuerst dem Netz nutzbar gemacht werden. Dafür hat unsere Vorgängergruppe eine Pipeline geschrieben, die aus den Daten die mfcc-features extrahiert und diese anschließend quantisiert. Vor dem extrahieren der mfcc-features wird das Audiosignal bis zu einer bestimmten Größe mit Nullen aufgefüllt. Diese wurden immer an das Ende der Datei angehängt, somit war der „wichtige“ Teil immer am Anfang des Inputs und dies hat sich das Neuronale Netz gemerkt. Auch ein DropOut-Layer kann dies nicht vollständig verhindern. Somit wurde die Vorverarbeitung derart verändert, dass das Signal nicht nur am Ende aufgefüllt wird, sondern auf beiden Seiten.

Dies wurde wie folgt realisiert:
Nehmen wir an, dass n die Länge ist, die aufgefüllt werden soll. Mit der Python Library `random` wird nun eine Zufällige ganze Zahl zwischen 0 und n ermittelt, um die Länge zu erhalten, die auf der rechten Seite aufgefüllt werden soll. Der Rest wird dann auf der linken Seite eingefügt. Somit ist das aufgenommene Audio-Signal beim Training nicht immer an der gleichen Stelle und das Netz kommt besser mit unbekannten Daten zurecht. Dies wurde im Labor experimentell herausgefunden, als wir Aufnahmen, die wir über das FPGA-Mikrofon aufgenommen hatten, dem Netz zum Testen übergeben haben und diese nicht erkannt wurden. Nach dem Umstellen der Vorverarbeitung wurden diese Dateien nun alle korrekt erkannt.
Trotz dieses Erfolges lässt die Erkennung von direkt eingespeisten Mikrofon Signalen noch sehr zu wünschen übrig. Wenn die eine Aufnahme als wav oder mp3 Datei gespeichert und das Netz damit getestet wird, liegt die Erkennungsrate bei ca. 90%. Bei den Mikrofon Daten bei ca. 5%. Der Grund dafür ist weiter unbekannt.

5.3 Umwandlung des Netzes in eine ONNX-Datei

geschrieben von Malte Voigt

Um das Netz über den FINN-Compiler auf ein FPGA-Board zu bekommen, muss das Neuronale Netz in ein bestimmtes Format gebracht werden. Hierfür schlägt FINN das Dateiformat **O**pen **N**eural **N**etwork e**X**change (onnx [ONN23]) vor. Dies ist ein standardisiertes Format zum Übertragen von Neuronalen Netzen und wird auch von PyTorch und Brevitas unterstützt. Für den Finn-Compiler kommen aber nicht alle von onnx unterstützen Graphen in Frage und anders herum [FIN23]. Finn benutzt viele eigene Operationen, die so nicht von onnx unterstützt werden. Brevitas besitzt aber die Funktion `export_finn_onnx` um ein Neuronales Netz in eine für den Finn-Compiler lesbare onnx-Datei zu konvertieren. Diese erwartet das Model, einen Pfad zum exportieren und die Shape des InputLayers, in unserem Fall y = 1, x = 1990.
Bei dem erstmaligen Versuch das Netz umzuwandeln, gab es die Fehlermeldung `TypeError: export() got an unexpected keyword argument 'enable_onnx_checker'`. Diese wurde verursacht, da die installierte Version von PyTorch geprüft wird und anschließend ein Attribut in die Argumente der nächsten Methode eingefügt wird, die damit nicht umgehen kann. Diese „fehlerhafte“ Methode muss dann einmal auskommentiert werden. Sie befindet sich in Zeile 77 des `brevitas/export/onnx/`
`manager.py` von Brevitas und trägt den Namen `solve_enable_onnx_checker`.
Zum konvertieren eines Netzes muss beim starten des `EmbedsKi.py` Skriptes der Parameter `-convert` mit einem oder mehreren kommaseparierten ModelNamen bspw. `-modelName=MyModel` angegeben werden.

6 FINN-Compiler und FPGA-Deployment

geschrieben von Christian Last, Nils Jahns

6.1 Installation bzw. Einrichtung

geschrieben von Christian Last

6.1.1 Betriebssystem

Linux Distribution

Für die Nutzung des FINN-Compilers ist eine Linux-Installation sowie grundlegende Kenntnisse im Umgang mit dieser Voraussetzung. Grundsätzlich ist die Wahl der Distribution dem Benutzer überlassen, Voraussetzung ist allerdings, dass sich sowohl die Xilinx Vivado Design Suite als auch die Docker Umgebung installieren lassen müssen, da beides für die Benutzung des FINN Compilers benötigt wird.

Für dieses Projekt wurde der FINN-Compiler in einer nativen Linux Distribution ausgeführt. Das Ausführen des Compilers in einer virtuellen Maschine mit Linux Installation wird in diesem Dokument nicht behandelt.

Hardware

Bezüglich Hardware gelten verschiedene Voraussetzungen. Zum einen müssen die generellen Anforderungen der zum Einsatz kommenden Linux-Distribution beachtet werden. Ferner gelten die Anforderungen der Xilinx Vivado Design Suite (siehe: [Xil23h]) und des FINN-Compilers (siehe: [Xil23g]).

Benutzerrechte

Auch wenn es im Hinblick auf Sicherheitsaspekte das Ziel sein sollte, möglichst alle Aktionen innerhalb eines Betriebssystems auf Benutzerebene auszuführen, sind für die Installation des FINN-Compilers stellenweise Root-Rechte erforderlich, d.h. dass bestimmte Befehle als Root-Benutzer mit erhöhten Zugriffsrechten auf das System ausgeführt werden müssen. Daher ist es für das Befolgen dieser Anleitung notwendig, dass das Passwort für den Root-Benutzer bekannt ist.

6.1.2 Xilinx Vivado Design Suite

Der FINN-Compiler benötigt zum Erzeugen der FPGA-optimierten neuronalen Netze die Xilinx Vivado Design Suite bzw. Tools aus dieser Suite. Da der Compiler erfolgreich mit Version 2022.1 der Vivado Design Suite getestet wurde, wird im Folgenden die Installation dieser Version beschrieben.

Voraussetzungen

Für die Installation und ggf. für das spätere Einrichten einer Lizenz wird ein Konto bei Xilinx (resp. AMD) vorausgesetzt. Dieses Konto kann kostenlos eingerichtet werden (siehe: [AMD23b]).

Ferner muss für eine erfolgreiche Installation sichergestellt werden, dass ein Paket namens „libtinfo5“ innerhalb der Linux-Umgebung installiert ist. Das Paket lässt sich mit folgendem Konsolenbefehl installieren:

```
$ sudo apt-get -y install libtinfo5
```

Web-Installer

Im ersten Schritt der Installation muss der Web Installer von der Xilinx Webseite (siehe: [AMD23a]) heruntergeladen werden. Es ist darauf zu achten, dass auf der linken Seite die Version 2022.1 ausgewählt wurde. Etwas weiter unten auf der Seite befindet sich der Download-Link für den gut 265MB großen Installer („Xilinx Unified Installer 2022.1: Linux Self Extracting Web Installer“). Gegebenenfalls ist es erforderlich, sich vor dem Download in seinen Xilinx-Account einzuloggen.

Nach dem Download muss das Installationspaket unter Umständen zunächst ausführbar gemacht werden. Dies kann über folgenden Befehl in einer Konsole erfolgen:

```
$ chmod +x Xilinx_Unified_2022.1_0420_0327_Lin64.bin
```

Danach lässt sich der Web Installer mit folgendem Konsolenbefehl ausführen:

```
$ ./ Xilinx_Unified_2022.1_0420_0327_Lin64.bin
```

Nach dem Start des Installationsprogramms müssen zunächst die Login-Daten des Xilinx Accounts eingegeben werden. Ferner bietet der Installer nun zwei Optionen: zum einen „Download and Install Now“ und zum anderen „Download Image (Install Separately)“.

Es wird dringend empfohlen, die zweite Option auszuwählen und somit ein Abbild der Installation auf den eigenen Rechner zu laden. Die Installation der Vivado Design Suite ist aufgrund der Download-Größe von etwa 70GB sehr langwierig und falls es bei der direkten Installation zu Problemen kommt, muss der gesamte Prozess erneut durchlaufen werden, wohingegen ein einmal heruntergeladenes Installations-Abbild immer wieder ausgeführt werden kann.

Für den Download muss ein Ort gewählt werden, auf den das Installationsprogramm Zugriff hat. Für das Beispiel (siehe Abb. 9) wurde das Download-Verzeichnis des aktuellen Benutzers gewählt. Die Auswahl für das Betriebssystem muss „Linux“ sein und es sollte das „Full Image“ heruntergeladen werden (siehe Abb. 9).

Wichtig: Das Installations-Abbild der Vivado Design Suite darf nicht in einem Verzeichnis liegen, das Leerzeichen enthält, da sich ansonsten der Offline-Installer nicht starten lässt.

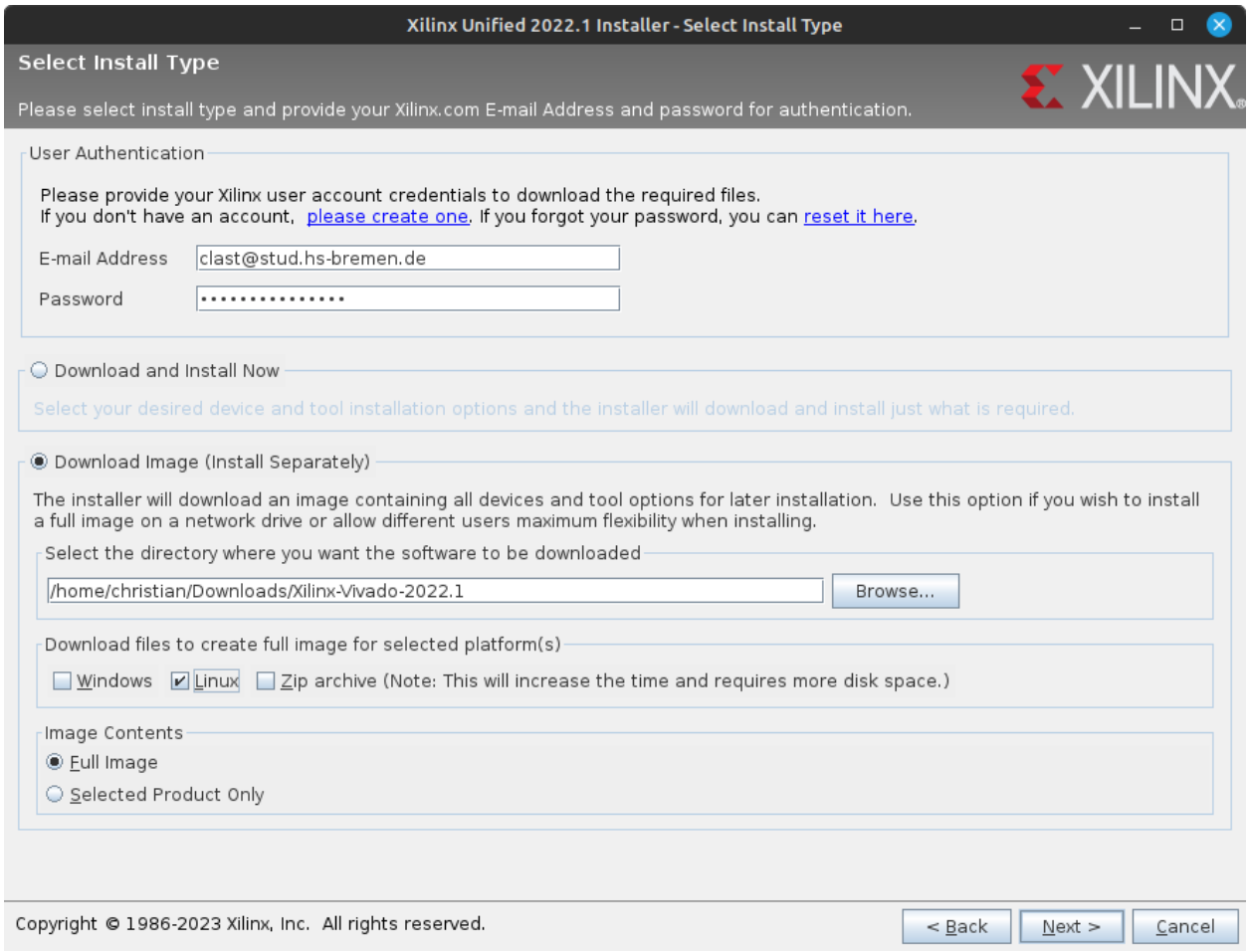

Abbildung 9: Xilinx Unified Web Installer

Installation

Im zweiten Schritt kann nach dem Download des Installationsabbildes die Xilinx Vivado Design Suite installiert werden. Hierzu wird der Offline-Installer über den folgenden Konsolenbefehl aufgerufen:

```
$ ./xsetup
```

Im nun gestarteten Installationsprogramm wählt man nun zunächst „Vivado“ aus. Ob „Vivado ML Standard“ oder „Vivado ML Enterprise“ installiert werden soll, spielt keine Rolle. In diesem Beispiel wird die Enterprise-Version installiert.

Nach der Wahl der Programmversion folgt die Auswahl der Komponenten. Die bereits ausgewählten Einstellungen können übernommen werden, lediglich die Checkbox für „Acquire or Manage a License Key“ sollte deaktiviert werden (siehe Abb. 10). Im Anschluss müssen noch vier Lizenzvereinbarungen akzeptiert werden.

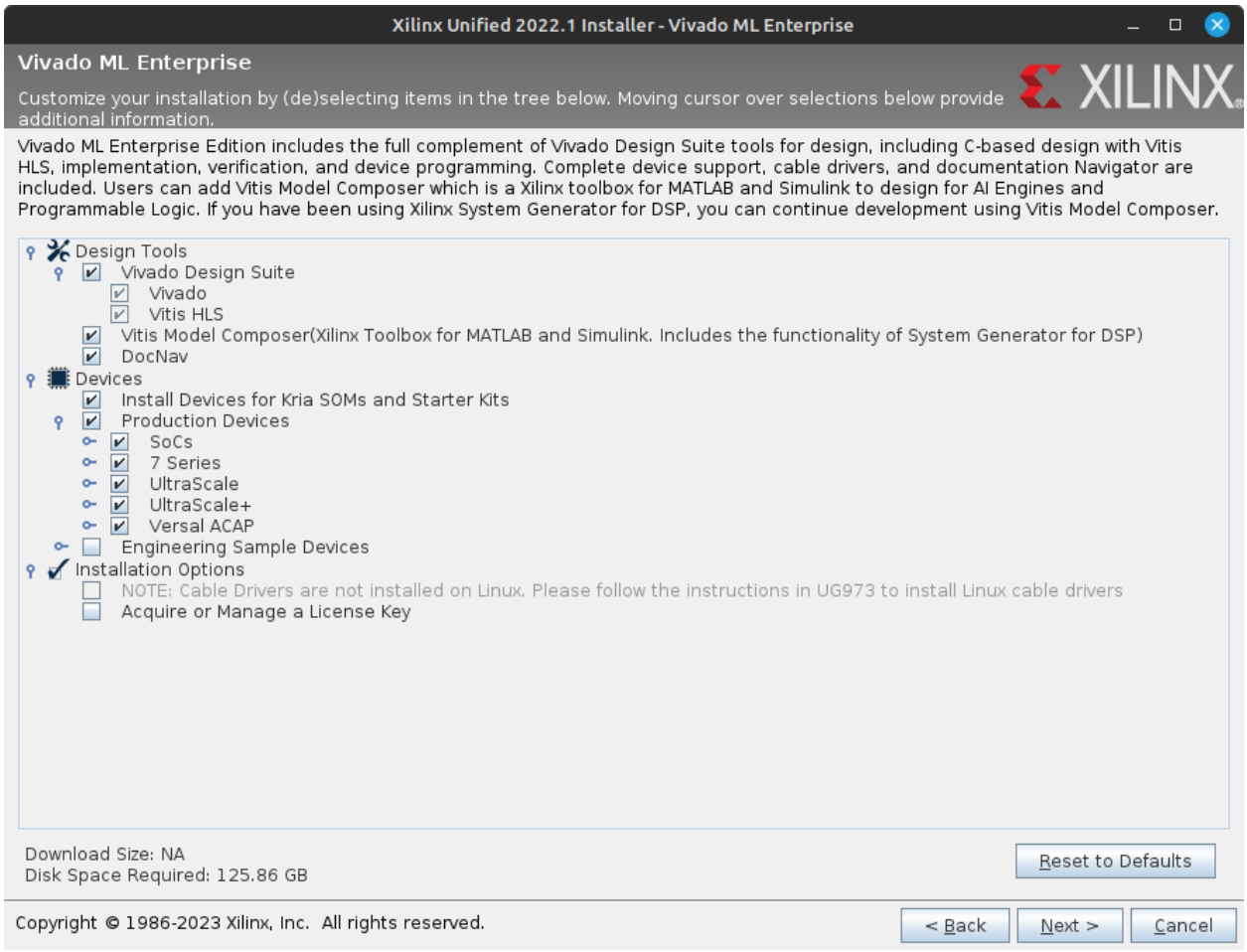

Abbildung 10: Xilinx Vivado Suite Offline Installer: Einstellungen

Abschließend wird das Zielverzeichnis der Installation festgelegt. Das im Voraus eingetragene Verzeichnis lässt sich nicht beschreiben, da hierfür Root-Rechte benötigt werden und die Installation ohne diese Rechte ausgeführt wurde. In diesem Beispiel wird daher ein Verzeichnis innerhalb des Home-Verzeichnisses des aktuellen Benutzers gewählt (siehe Abb. 11). Optional können durch das Installationsprogramm Einträge im Menü der Desktopumgebung bzw. Shortcuts auf dem Desktop angelegt werden.

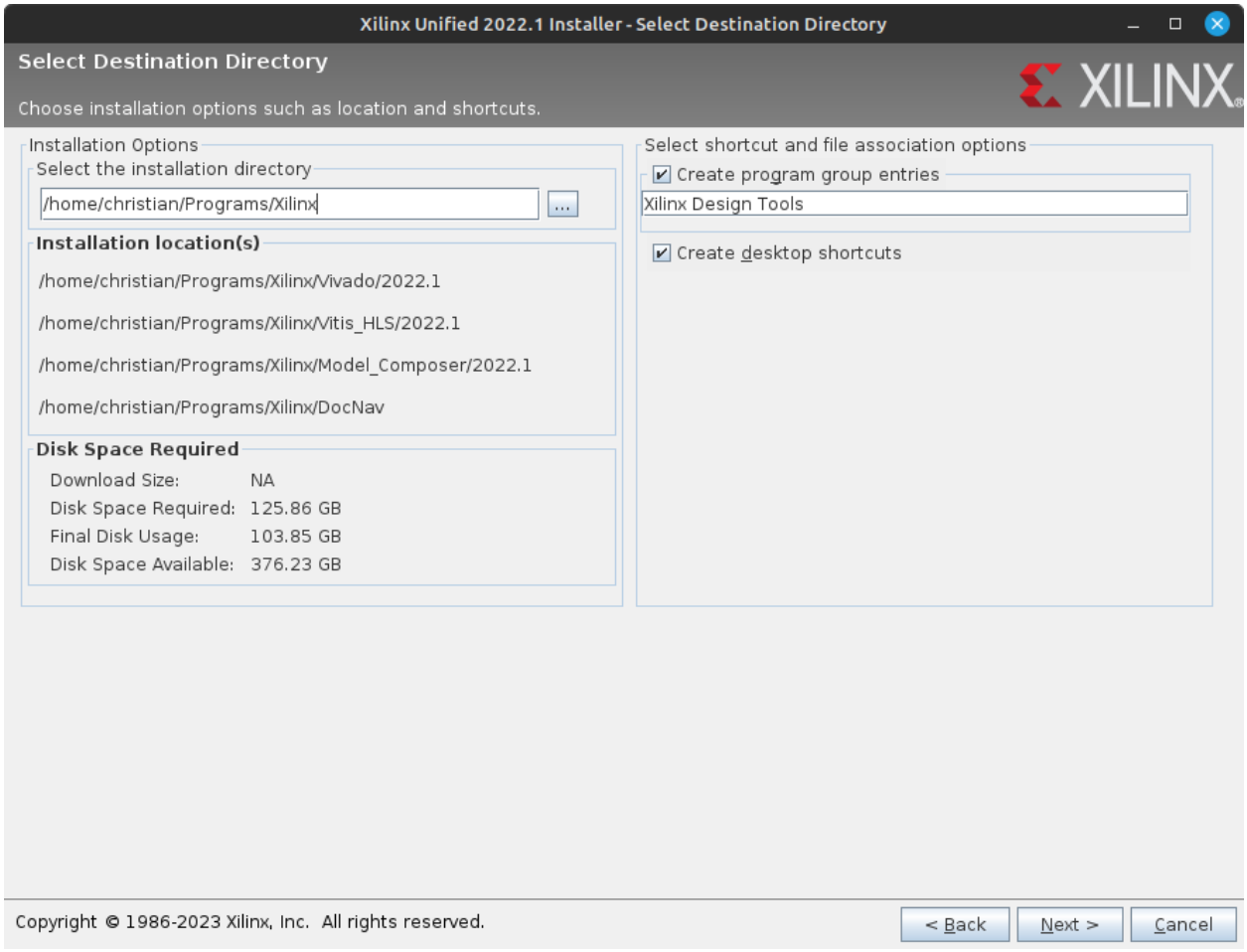

Abbildung 11: Xilinx Vivado Suite Offline Installer: Installationsverzeichnis

Automatisches Laden der Konfiguration

Die Xilinx Vivado Design Suite ist nun installiert und kann z.B. über die Desktopumgebung aufgerufen werden. Um eine korrekte Funktion innerhalb des gesamten Systems gewährleisten zu können, sollte die dafür nötige Konfiguration automatisch geladen werden. Dies kann auf verschiedene Wegen passieren; in diesem Beispiel wird die für den aktuellen Nutzer gültige Profil-Datei angepasst.

Zunächst muss die Datei mit dem Konfigurationsprofil des aktuellen Nutzers mit einem beliebigen Texteditor geöffnet werden. Wichtig hierbei ist, dass Root-Rechte benötigt werden. Unter Ubuntu Linux kann die Datei z.B. über die Konsole mit dem Editor „Gedit“ editiert werden:

```
$ sudo gedit /etc/profiles
```

Am Ende der Datei müssen nun zwei Befehle eingefügt werden, mit denen die Vivado-Konfiguration systemweit geladen wird:

```
source /Installationspfad/Vivado/2022.1/settings64.sh
source /Installationspfad/Vitis_HLS/2022.1/settings64.sh
```

Wichtig: Es muss der korrekte Pfad zur Installation der Design Suite angegeben werden; dieser ist hier mit „*Installationspfad*“ nur exemplarisch dargestellt. Für dieses Beispiel gilt das zuvor bei der Installation gewählte Verzeichnis (siehe Abb. 12).

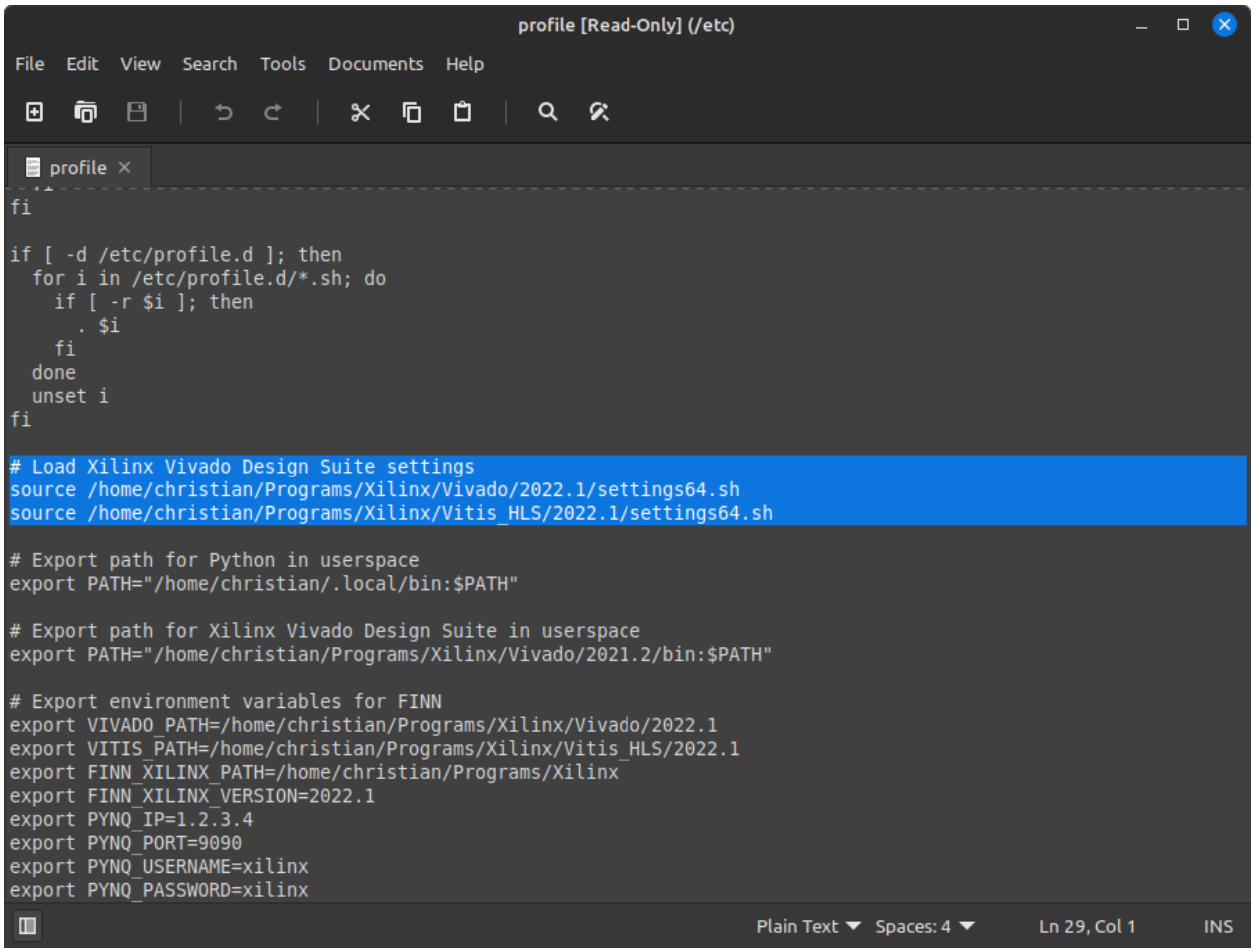

Abbildung 12: `/etc/profile` für ein automatisches Laden der Konfiguration anpassen

6.1.3 Docker

Der FINN-Compiler wird mit Hilfe des Plattform-Tools „Docker“ bereitgestellt. Docker bietet die Möglichkeit, eine Anwendung bzw. eine Sammlung von Anwendungen in einer lose isolierten Umgebung, einem sogenannten „Container“, zu verpacken und auszuführen.

Die Installation von Docker muss über die Konsole erfolgen und erfordert, dass zunächst ein paar Pakete innerhalb der Linux-Umgebung installiert werden. Dies erfolgt über folgenden Konsolenbefehl:

```
$ sudo apt-get update && sudo apt-get -y install ca-certificates curl \
  gnupg lsb-release
```

Nun muss der GPG Schlüssel des Docker Repositoriums hinzugefügt werden:

```
$ sudo mkdir -p /etc/apt/keyrings && curl -fsSL \
  https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o \
  /etc/apt/keyrings/docker.gpg
```

Jetzt kann das Repositorium als offizielle Quelle für den Paketmanager hinzugefügt werden:

```
$ echo "deb [arch=$(dpkg --print-architecture) \
  signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" \
  | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Abschließend kann Docker über den Paketmanager installiert werden:

```
$ sudo apt-get update && sudo apt-get -y install docker-ce docker-ce-cli \
  containerd.io docker-compose-plugin
```

Die Ausführung des Containers mit dem FINN-Compiler sollte als Nicht-Root-User erfolgen. Dafür müssen dem aktuellen Benutzer die Rechte zum Ausführen von Docker zugewiesen werden:

```
$ sudo groupadd docker && sudo usermod -aG docker $USER && newgrp docker
```

Ob Docker nun korrekt mit Benutzerrechten ausgeführt werden kann, kann über folgenden Konsolenbefehl getestet werden:

```
$ docker run hello-world
```

6.1.4 FINN Compiler

Der FINN-Compiler an sich ist kein Compiler im klassischen Sinne, sondern ist eher als „Compiler-Architektur“ zu verstehen, die aus diversen Komponenten besteht. Da es sich hier um ein komplexes System mit vielen Abhängigkeiten handelt, wird der FINN-Compiler von Xilinx in einem Docker Container bereitgestellt.

Um den Container des Compilers ausführen zu können, muss dieser zunächst erzeugt werden. Hierzu wird im ersten Schritt das Git-Repositorium des FINN-Compilers geklont.

Wichtig: Es wird dringend empfohlen, nicht den „`main`“-Zweig des Repositoriums zu benutzen, da dieser deutlich seltener aktualisiert wird und es ggf. zu Problemen mit Abhängigkeiten von Software-Modulen kommen kann. Stattdessen sollte der „`dev`“-Zweig des Git-Repositoriums verwendet werden.

Der `dev`-Zweig des Repositoriums kann mit folgendem Konsolenbefehl in ein lokales Verzeichnis (mit Schreibrechten) geklont werden:

```
$ git clone -b dev https://github.com/Xilinx/finn/
```

Bevor der Docker-Container nun erzeugt werden kann, sollten im System einige Umgebungsvariablen eingerichtet werden. Diese dienen unter anderem dazu, dass der FINN-Compiler die Installation der Xilinx Vivado Design Suite finden und nutzen kann. Das kann, wie bereits bei der Installation des Design Suite, über mehrere Einträge in der Konfiguration des Benutzerprofils (`/etc/profile`) geschehen. Folgende Einträge müssen am Ende der Datei vorgenommen werden:

```
export VIVADO_PATH=/Installationspfad/Vivado/2022.1
export VITIS_PATH=/ Installationspfad/Vitis_HLS/2022.1
export FINN_XILINX_PATH=/Installationspfad
export FINN_XILINX_VERSION=2022.1
```

Wichtig: Hier ist der Pfad zur Installation lediglich symbolisch mit „*Installationspfad*“ dargestellt. Für dieses Beispiel gilt das zuvor bei der Installation gewählte Verzeichnis (siehe Abb. 13).

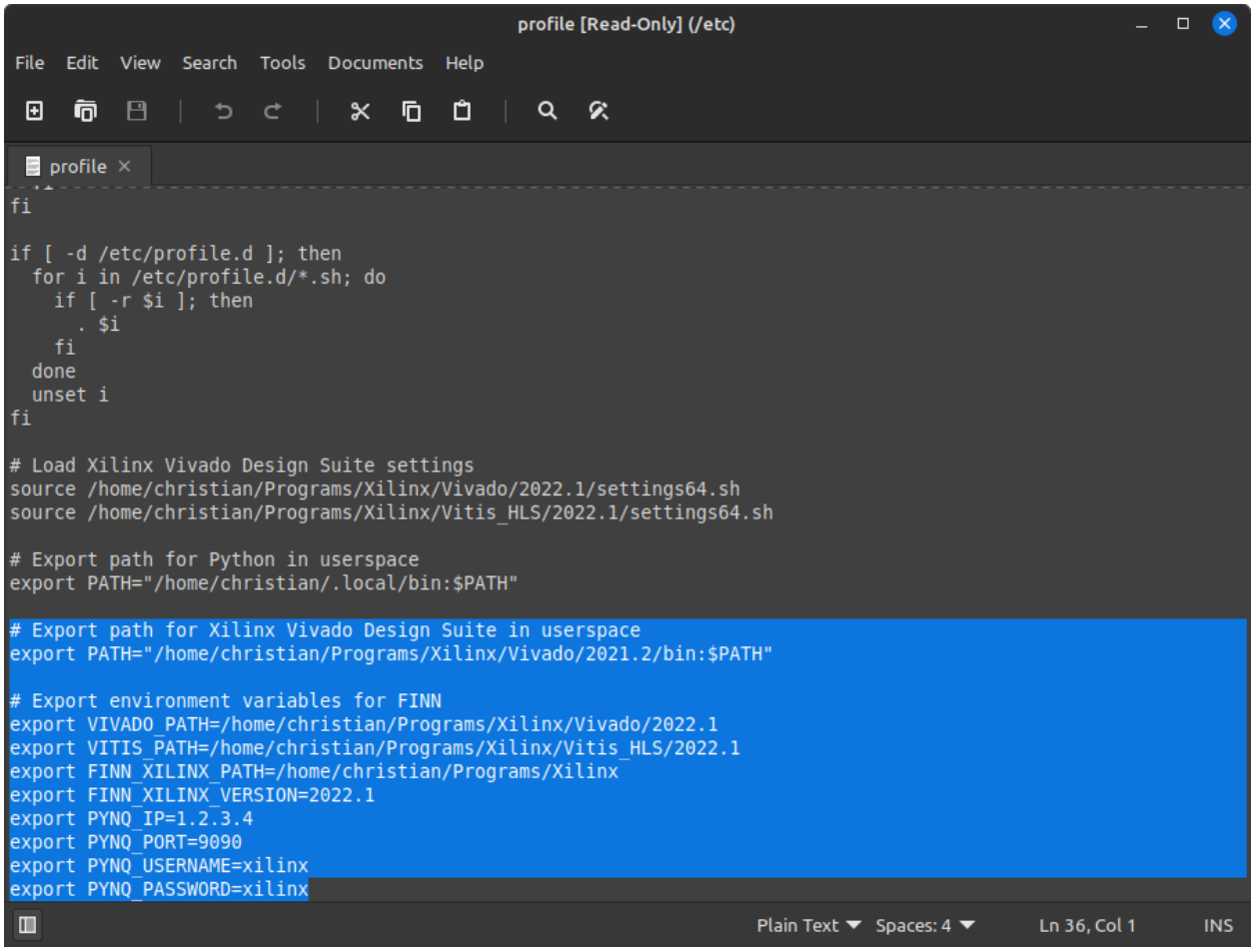

Abbildung 13: Ergänzungen in `/etc/profile`

Mit folgendem Konsolenbefehl kann die geänderte Konfiguration nun neu geladen bzw. eingelesen werden:

```
$ source /etc/profile
```

Sobald die Konfiguration neu geladen wurde, kann der Docker-Container des FINN-Compilers aus seinem Verzeichnis heraus mit folgendem Konsolenbefehl aufgerufen werden:

```
$ ./run-docker.sh
```

Wichtig: Beim ersten Ausführen des Containers werden alle nötigen Programme und deren Abhängigkeiten heruntergeladen bzw. neu kompiliert. Dies kann unter Umständen eine gewisse Zeit dauern. Sobald der Container einmal erzeugt wurde, wird das Ausführen desselben über den o.g. Befehl deutlich schneller geschehen.

Die einfachste Möglichkeit, den FINN-Compiler zu verwenden, wird durch einen Webserver für Jupyter Notebooks realisiert. Hierfür muss der Docker-Container mit folgendem Konsolenbefehl gestartet werden:

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

Der Container wird dann einen Webserver für Jupyter Notebooks ausführen und die dazugehörige Adresse nach dem Start des Servers in der Konsole ausgeben.

6.2 Erzeugen eines FPGA-beschleunigten neuronalen Netzes

geschrieben von Christian Last

Für das Erzeugen eines FPGA-beschleunigten neuronalen Netzes muss zunächst ein neuronales Netz vorliegen, welches quantisiert und via Brevitas im ONNX Format exportiert wurde. Die eigentliche Arbeit des Compilers besteht dann aus zwei Teilen:

1. Kompilierung: Der FINN-Compiler erzeugt aus dem Netz eine Hardware-Implementierung, die auf einem FPGA eingesetzt werden kann. Dies umfasst mehrere Optimierungsschritte, wie z. B. Folding und Datenlayout-Optimierung, um eine effiziente Implementierung zu erzeugen, die den Hardware-Einschränkungen des Ziel-FPGAs entspricht.
2. Bitstream-Erzeugung: Der zweite Schritt ist die Generierung eines Bitstreams, der auf das FPGA geladen werden kann, um das neuronale Netz auf dem FPGA auszuführen und Inferenzen auf neue Eingabedaten anzuwenden.

Kernstück des Compilers ist das sogenannte `build_Dataflow`-Tool. Diesem Tool können alle relevanten Informationen zur Erzeugung des FPGA-beschleunigten Netzes übergeben werden, woraufhin die benötigten Schritte automatisch abgearbeitet werden. Das Tool wird entweder über Kommandozeile des FINN Docker Containers ausgeführt, oder innerhalb des Jupyter-Notebooks als Python-Bibliothek verwendet.

Für die Verwendung des FINN-Compilers empfiehlt sich ein Blick in die durch Xilinx mit dem Docker Container bereitgestellten „End-to-End Flow“ Beispiele. Hier ist inbesondere das Jupyter Notebook `3-build-accelerator-with-finn.ipynb` des „`cybersecurity`“ Ordners hervorzuheben, auf dem die in diesem Projekt hervorgebrachten Ergebnisse basieren.

Build Config

Für die Erzeugung eines FPGA-beschleunigten Netzes wird eine Instanz von `finn.builder.build_dataflow_config.DataflowBuildConfig` definiert, welche als Build-Konfiguration dient. Es gibt viele Optionen zur Konfiguration, um verschiedene Aspekte des Builds anzupassen. Details zu allen Konfigurationsoptionen sind der FINN-API-Dokumentation zu entnehmen (siehe [Xil23c]).

Estimate Reports

Bevor ein Netz kompiliert wird, kann durch den Compiler ein sogenannter „*Estimate Report*“ erzeugt werden. Hier wird der zu erwartende Ressourcenverbrauch und die zu erwartende Leistung des Netzes erfasst, ohne jedoch ein komplette Synthese zu durchlaufen.

Folgender Quellcode wurde verwendet, um die Estimate Reports für das in diesem Projekt verwendete Netz zu erzeugen:

```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
import shutil

model_file = "DefaultModel.onnx"

estimates_output_dir = "output_estimates_only"

#Delete previous run results if exist
if os.path.exists(estimates_output_dir):
    shutil.rmtree(estimates_output_dir)
    print("Previous run results deleted!")


cfg_estimates = build.DataflowBuildConfig(
    output_dir          = estimates_output_dir,
    mvau_wwidth_max     = 80,
    target_fps          = 1000000,
    synth_clk_period_ns = 10.0,
    fpga_part           = "xc7z020clg400-1",
    steps               = build_cfg.estimate_only_dataflow_steps,
    generate_outputs=[
        build_cfg.DataflowOutputType.ESTIMATE_REPORTS,
    ]
)
build.build_dataflow_cfg(model_file, cfg_estimates)
```

Als wichtige Parameter der Konfiguration sind hier `fpga_part = "xc7z020clg400-1"` (die exakte Bezeichnung des auf dem PYNQ-Z1 Board verbauten FPGAs) und `generate_outputs = [build_cfg.DataflowOutputType.ESTIMATE_REPORTS]` (die Anweisung, einen Report zu erstellen) hervorzuheben.

Wird dieser Code nun ausgeführt, werden diverse Berichte im Verzeichnis `output_estimates_only` abgelegt, unter anderem die Datei `estimate_ network_ performance.json`, welche Aufschluss über den geschätzten Durchsatz oder die geschätzte Latenz des Netzes gibt:

```
{
  "critical_path_cycles": 140,
  "max_cycles": 120,
  "max_cycles_node_name": "MatrixVectorActivation_0",
  "estimated_throughput_fps": 833333.3333333334,
  "estimated_latency_ns": 1400.0
}
```

Eine weitere Datei namens `estimate_layer_resources.json` zeigt die benötigte Menge an Lookup-Tabellen und Blockspeicher:

```
{
  "Thresholding_Batch_0": {
    "BRAM_18K": 0,
    "BRAM_efficiency": 1,
    "LUT": 1592,
    "URAM": 0,
    "URAM_efficiency": 1,
    "DSP": 0
  },
  "MatrixVectorActivation_0": {
    "BRAM_18K": 17,
    "BRAM_efficiency": 0.22863051470588236,
    "LUT": 7971,
    "URAM": 0,
    "URAM_efficiency": 1,
    "DSP": 0
  },
  "total": {
    "BRAM_18K": 17.0,
    "LUT": 9563.0,
    "URAM": 0.0,
    "DSP": 0.0
  }
}
```

Bitfile und Treiber

Folgender Quellcode wurde verwendet, um die Code-Synthese des in diesem Projekt verwendeten Netzes anzustoßen und ein auf dem PYNQ-Z1 Board ausführbares Paket zu erhalten:

```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
import shutil

model_file = "DefaultModel.onnx"

final_output_dir = "output_final"

#Delete previous run results if exist
if os.path.exists(final_output_dir):
    shutil.rmtree(final_output_dir)
    print("Previous run results deleted!")

cfg = build.DataflowBuildConfig(
    output_dir          = final_output_dir,
    mvau_wwidth_max     = 80,
    target_fps          = 1000000,
    synth_clk_period_ns = 10.0,
    board               = "Pynq-Z1",
    shell_flow_type     = build_cfg.ShellFlowType.VIVADO_ZYNQ,
    generate_outputs=[
        build_cfg.DataflowOutputType.BITFILE,
        build_cfg.DataflowOutputType.PYNQ_DRIVER,
        build_cfg.DataflowOutputType.DEPLOYMENT_PACKAGE,
    ]
)
#build.build_dataflow_cfg(model_file, cfg)
```

Im Vergleich zum Quellcode für die Estimate Reports ist hier hervorzuheben, dass nicht mehr die Bezeichnung des FPGAs angegeben wird, sondern mit `board = "Pynq-Z1"` die Bezeichnung des verwendeten Boards. Ferner werden im Array `generate_outputs` die für das Deployment benötigten Ausgaben spezifiziert (`BITFILE`, `PYNQ_DRIVER` und `DEPLOYMENT_PACKAGE`)

6.3 Deployment und Ausführung auf dem PYNQ-Z1 Board

geschrieben von Nils Jahns

Für das Deployment muss nur der Ordner *deploy*, der vom FINN-Compiler generiert wurde, in das Dateisystem vom PYNQ-Z1 kopiert werden.

In diesem Ordner befinden sich u.a. das generierte Bitfile (*bitfile/finn-accel.bit*), eine HWH (hardware handoff)-Datei(*bitfile/finn-accel.hwh* und eine Python-Skript (*driver/driver.py*). Diese Dateien können später als Grundlage für das Zielsystem genutzt werden.

Zum ersten Testen des Netzes kann das Python-Skript *driver.py* ausgeführt werden. Dazu muss zum einen das Bitfile und zum anderen ein Inputfile übergeben werden.

Als Inputfile muss eine NPY-Datei übergeben werden, welche mit Python und der NumPy-Bibliothek erstellt werden kann. Im Prinzip handelt es sich um ein, in einer Datei gespeichertes, NumPy-Array. Als Datentyp für das Array wird ein vorzeichenloses 4-Bit Integer erwartet.

Für die Erstellung des Arrays wurde im Repository (A.4) die Möglichkeit geschaffen die MFCCs als NPY-Datei abzuspeichern. Diese haben aber noch den falschen Datentypen für die Ausführung auf dem FPGA, sodass erst noch das Hilfsskript *convertToint4.py*, welches extra für diesen Test geschrieben wurde, ausgeführt werden muss, um die Konvertierung durchzuführen. Die generierte NPY-Datei kann nun auch auf das PYNQ-Z1 kopiert werden.

Mit folgendem Kommando wird nun das Neuronale Netz in das PFGA geladen und die Erkennung durchgeführt:

```
$ python driver.py --bitfile ../bitfile/finn-accel.bit --inputfile input.npy
```

Als Ergebnis wird ein NPY-Outputfile generiert, das die Vorhersagen vom Netz beinhaltet. Dabei handelt es sich um ein Array vom Typ Int16, mit der Größe entsprechend der Anzahl der Kommandos, das in jedem Element einen Wert für die jeweiligen Kommandos enthält. Der höchste Wert gibt die höchste Erkennungswahrscheinlichkeit an.

7 Evaluation

7.1 Audiodatenverarbeitung

geschrieben von Jarno Burggräf

Die aufgenommenen und normalisierten Audioaufnahmen sind subjektiv klar verständlich und in ihrer Dauer nicht durch die Schaltung im FPGA beschränkt, sodass eine kontinuierliche Aufnahme möglich ist. Eine Verbesserung der Qualität durch eine bessere Parametrisierung des CIC-Filters und ggf. ein Hinzufügen eines FIR-Filters wäre möglich, wurde bisher aber nicht als notwendig eingestuft. Es bleibt außerdem noch zu untersuchen, wie sich die Anwendung im Gesamtsystem mit den Motoren als Störgeräuschquellen und einem größeren Abstand zwischen der die Befehle sprechenden Person und dem Mikrofon verhält.

7.2 Neuronales Netz

geschrieben von Malte Voigt, Nils Jahns

7.2.1 Modell

geschrieben von Malte Voigt

Es hat sich herausgestellt, dass es noch einige Verbesserungsmöglichkeiten, im Hinblick auf das verwendete Model und sein Training, gibt. Es wurde z.B. nur ein Bruchteil der zu Verfügung stehenden Layer benutzt, um ein Neuronales Netz zu bauen. Dafür wurde nun die Möglichkeit gegeben, einfacher verschiedenste Models zu erstellen und diese zu trainieren.
Weiterhin ist aufgefallen, dass die Vorverarbeitung der Trainingsdaten nicht ganz optimal verlaufen ist. Obwohl das Model mit 4s Aufnahmen trainiert wurde, konnte es nur Aufnahmen mit der Länge von $<= 1s$ einigermaßen erkennen. Dies lag daran, dass sich das Eingesprochene immer am Anfang der Daten befand. War das nun in einer Aufnahme nicht der Fall, wurde diese Datei überwiegend als `None` erkannt. Durch die Änderung an der Vorverarbeitung werden nun bereits aufgenommene wav und mp3 Dateien sehr zuverlässig erkannt, direkt eingespeiste Mikrofon-Daten hingegen noch immer nicht. Der Grund dafür ist noch nicht bekannt.
Beim Umwandeln eines Netzes in das ONNX Format gibt es aktuell noch das Problem, dass es anschließend häufig, aufgrund einer Output-Shape von (0,0), nicht zu gebrauchen ist. Hier müsste untersucht werden, woran das liegt und welche Layer noch Probleme bereiten.
Eine weitere Sache, die noch immer nicht ganz funktioniert, ist das Testen des Models mit dem Mikrofon. Wurde ein Model trainiert, kann es auf 2 verschiedene Arten getestet werden. Einmal über das Angeben einer Datei die ausgelesen, verarbeitet und an das Model gegeben wird, oder das Aufnehmen über ein Mikrofon, das dann wie die Datei verarbeitet und weitergegeben wird. Über die bereits aufgenommenen Dateien wird das Gesagte mit hoher Wahrscheinlichkeit erkannt, wird allerdings unmittelbar vorher aufgenommen, jedoch kaum. Dabei ist das verwendete Mikrofon nicht von Bedeutung. Es wurde mit min. 3 verschiedenen

Mikrofonen versucht, das Ergebnis war immer das Selbe. Eine Theorie ist, dass die Daten vom Mikrofon sich um einen Faktor x von den Daten aus den Aufgenommenen Dateien unterscheiden. Diese kommt aus der Erfahrung eines parallel laufenden Moduls, jedoch konnte noch kein solcher Faktor gefunden werden.

7.2.2 Training und Erkennung

geschrieben von Nils Jahns

Wie in 5.2.1 gezeigt, konnte das Training mit GPU-Unterstützung um den Faktor 16 beschleunigt werden. Der genaue Faktor hängt natürlich von den jeweiligen Hardwarespezifikationen ab, aber grundsätzlich besteht ein Vorteil gegenüber einem Training auf der CPU.

Auch konnte gezeigt werden, dass durch Löschen korrupter und unpassender Audiodateien die Erkennungsrate erhöht wurde. Der Hinweis, dass es korrupte Audiodateien gibt hat die neu implementierte Confusion Matrix (5.2.2) geliefert. So wurde beim Kommando *back* die Erkennungsrate von 50% auf 100% für den gegebenen Testdatensatz erhöht. Die Gesamtgenauigkeit konnte durch das Löschen der Audiodateien von 90,4% auf 94,9% für das betreffende Modell (vor der Bearbeitung mit dem FINN-Compiler) gesteigert werden.

Zur Ausführungszeit in Millisekunden des Neuronalen Netzes konnten folgende Werte ermittelt werden:

	CPU (Desktop PC)	GPU (Desktop PC)	FPGA
Erste Erkennungsdauer	699	709	649
Durchschnittliche Erkennungsdauer	5,1	7,5	620

Tabelle 3: Ausführungszeit in Millisekunde des NN

Zur Ermittlung der Werte wurde die Systemzeit nach der Audiodatenverarbeitung zwischen der Übergabe der Daten an das Neuronale Netz und dem Zurückgeben des Ergebnisses gemessen. Dabei wurden 50 Erkennungen in Folge durchgeführt, um einen Mittelwert zu ermitteln. Die erste Erkennung wurde beim Mittelwert nicht berücksichtigt, da sie auf dem Desktop PC deutlich langsamer ist als die folgenden Erkennungen.

Die Tatsache, dass die Erkennung beim Desktop PC auf der CPU schneller berechnet wird als auf der GPU liegt daran, dass es sich bei dem Netz um ein quantisiertes Netz handelt, das nicht mehr mit Fließkommazahlen arbeitet.

Zur Genauigkeit des Netzes auf dem FPGA konnten folgende Werte ermittelt werden:

	forward	back	left	right	stop	none
FORWARD - 11.mp3	-4	-27	-28	-414	-558	430
FORWARD - 13.mp3	33	-104	-67	-325	-637	504
FORWARD - 15.mp3	-78	-52	-136	-320	-649	487
FORWARD - 17.mp3	130	-133	-142	-367	-650	544
FORWARD - 19.mp3	19	-19	-178	-461	-645	570

Tabelle 4: Erkennungwerte auf dem FPGA

Hier zeigt sich, dass in jedem Testfall die Kategorie *none* mit der deutlich höchsten Wahrscheinlichkeit ausgegeben wurde. Jedoch ist immerhin in vier von fünf Fällen das richtige Kommando mit dem zweithöchsten Wert angegeben. Die jeweiligen Kommandos wurden mit dem Neuronalen Netz - bevor es vom FINN-Compiler verarbeitet wurde - auf dem Desktop-PC durchgehend richtig erkannt. Hier liegt der Verdacht nahe, dass durch die Konvertierung durch den FINN-Compiler die Genauigkeit des Neuronalen Netzes stark leidet.

7.3 FINN-Compiler

geschrieben von Christian Last

Ein Teilaspekt des Projektes war es, das trainierte Netz auf dem FPGA ausführbar zu machen. Hierzu konnte mit dem FINN-Compiler eine FPGA beschleunigte Version des neuronalen Netzes erzeugt und auf dem PYNQ-Z1 Boards ausgeführt werden. Durch die Funktion der „Estimate Reports“ (siehe 6.2) konnte ebenfalls ein Überblick über die geschätzte Leistung und den Ressourcenverbrauch generiert werden. Ein geschätzter Durchsatz von gut 83000fps bzw. ein Latenz von 1.4µs scheinen hierbei zunächst ein durchaus brauchbares Ergebnis darzustellen. Da jedoch aus zeitlichen Gründen im Projekt noch keine Kombination von Audioaufnahme bzw. -verarbeitung und Auswertung der Audiodaten durch das beschleunigte Netz erfolgt ist, kann keine abschließende Bewertung vorgenommen werden, ob die Performanz tatsächlich ausreichend für die Erkennung von Schlüsselwörtern ist.

7.4 Konfigurationsmangement

geschrieben von Felix Luther

7.4.1 GitLab

GitLab hat sich als eine gute Plattform heraugestellt um die Arbeit im Team zu ermöglichen. Mit der Vielzahl an Features konnten zudem hilfreiche Aktionen wie die Überprüfung und Generierung von LaTeX Quelldateien und folgend der PDF-Datei. Dieses konnte nach einem einmaligen Aufsetzen dem gesamten Team die Arbeit an der Dokumentation reduzieren.

7.4.2 GitLab Gruppen

Offiziell sind nur Gruppen von bis zu fünf Mitgliedern von GitLab kostenfrei möglich. Dieses Limit wird zu diesem Zeitpunkt jedoch noch nicht umgesetzt, für zukünftige Projekte könnte dieses Limit jedoch Anwendung finden und es muss nach einer Alternative gesucht werden, wie z.B. die Verwendung einer selbst-gehosteten Instanz von GitLab oder der verwendung öffentlich-einsehbaren Repositories.

7.4.3 GitLab Runner

Die Verwendung von manuellen Runnern hat eine Vielzahl von Vorteilen mit sich gebracht. Zu einem ist das Projekt nicht mehr dem Risiko ausgesetzt, dass die gratis Verfügbaren CICD-Minuten auslaufen, wie es beim Projekt des Vorjahres der Fall war. Für diese monatlich frei verfügbaren Minuten wird zusätzlich die Angabe einer Kreditkarte benötigt. Selbst mit älterer Hardware, in diesem Fall einem Intel Core 2 Duo wurde eine signifikant höhere Geschwindkeit der CICD Jobs erzielt. Beim Vorjahresprojekt nahm ein Job knapp unter drei Minuten in Anspruch, während der eigene Runner mit anderen laufenden Services im Hintergrund nur um die 40 Sekunden brauchte. Statt der Verwendung des immer neusten Images wurde nun in der Datei `.gitlab-ci.yml` eine feste Version von TeXLive hinterlegt. Diese hat den Vorteil, dass das Projekt somit von evtl. problematischen Änderungen von TeXLive unbetroffen ist, als auch nur ein Mal beim Aufsetzen eines Runners das Image sich heruntergeladen werden muss. Zu Anfang des Projektes kam es aufgrund der wöchentlichen Releases von TeXLive zu einer Verzögerung beim ersten Job um etwa fünf Minuten, da sich das neue Image, welches mehrere Gigabyte umfasst zuerst heruntergeladen werden musste.

8 Ergebnisse und Ausblick

geschrieben von Nils Jahns

Im folgenden sollen die Ergebnisse des Projektes beschrieben und ein kleiner Ausblick gezeigt werden.

Abbildung 5 auf Seite 12 gibt einen guten Überblick über das Ergebnis der Datenverarbeitung. Es weicht allerdings noch vom Zielsystem ab (siehe Abbildung 3 auf Seite 10).

Ein Teil der Audiodatenverarbeitung konnte auf dem FPGA umgesetzt werden. Es können nun Audiodateien mit dem FPGA-Mikrofon generiert werden und stehen für den Trainingsprozess zur Verfügung. Die Weiterverarbeitung findet allerdings noch auf einem Deskop-PC statt. Die Audioqualität des FPGA-Mikrofon hat sich als zufriedenstellend herausgestellt.

In einem nächsten Schritt muss nun noch der Rest der Audiodatenverarbeitung bis hin zu einem MFCC auf dem FPGA umgesetzt werden.

Der Trainingsprozess eines Neuronalen Netzes ist durch die Implementierung zusätzlicher Werkzeuge nun etwas transparenter geworden und kann besser verstanden werden. Auch konnte so das trainierte Netz verbessert werden. Zum einen durch eine veränderte Audiovorverarbeitung und zum anderen durch ein Review der vorhandenen Audiodatenbasis.

Es sind aber noch weitere Schritte notwendig, um die Netzstruktur für ein Neuronales Netz, das auf einem FPGA läuft, zu verbessern. Die Basis hierfür ist nun geschaffen, da es jetzt möglich ist verschiedene Modell-Klassen im Trainingsprozess zu unterscheiden und damit zu experimentieren.

In einem weiteren Schritt sollten Aufnahmen von FPGA-Mikrofon in das Training integriert werden, um die Erkennungsgenauigkeit einen Neuronalen Netzes auf dem FPGA zu erhöhen.

Um das optimale Netz für die gegebene Aufgabe zu finden, ist es ratsam eine Hyperparametersuche in den Trainingsprozess zu integrieren. Dabei werden mehrere Trainingsdurchläufe mit verschiedenen Parametern durchgeführt und können so am Ende verglichen werden. Hier zeigt sich dann, wie sich bestimmte Parameter auf die Qualität des Netzes auswirken. Gerade durch die Möglichkeit ein Netz mit GPU-Unterstützung trainieren zu können, ist dieses eine attraktive Option.

Zusätzlich wurde die Möglichkeit geschaffen das trainierte Netz im ONNX-Format zu konvertieren, sodass es mithilfe des FINN-Compilers auf dem FPGA eingesetzt werden kann.

Auch hier sind noch weitere Untersuchungen notwendig, da nicht jedes trainierte Netz - im Abhängigkeit der Netzstruktur - automatisch mit dem FINN-Compiler verarbeitet werden kann.

Nach dem Deployment des Netzes, welches auf dem Desktop-PC noch eine gute Erkennungsgenauigkeit gezeigt hat, auf dem FPGA haben erste Tests gezeigt, dass das Ergebnis für die geplante Anwendung noch zu ungenau ist. Das übergebene Kommando wurde immer mit der höchsten Wahrscheinlichkeit in die Kategorie *none* eingeordnet und höchstens an zweiter Stelle, allerdings mit großen Abstand, wurde das richtige Kommando erkannt. Hier sind

weitere Untersuchungen nötig, um herauszufinden, warum dieses Ergebnis zustande kam. Das betrifft explizit die Erkennungsgenauigkeit nach der Bearbeitung durch den FINN-Compiler und der Ausführung auf dem FPGA. Das gleiche Netz, das auf einem Desktop-PC ausgeführt wurde, hat eine deutlich bessere Erkennungsgenauigkeit.

Für die Ausführungszeit auf dem FPGA konnte ein durchschnittlicher Wert von 620 Millisekunden ermittelt werden. Ob dieser Wert für das Zielsystem akzeptabel ist oder ob hier noch nach Beschleunigungsmöglichkeiten gesucht werden muss, bedarf weiterer Tests.

Ein weiterer Punkt, der noch untersucht werden muss, ist, ob das FPGA noch genügend Kapazität bietet, um die Audiodatenverarbeitung neben dem Neuronalen Netz auf dem FPGA auszuführen oder ob das Netz schon zu viel Platz einnimmt.

Was in diesem Projekt noch unberührt bliebt, war das autonome Fahrzeug, auf dem das Neuronale Netz zum Einsatz kommen soll. Sobald eine zuverlässige Erkennung möglich ist, sollte das Fahrzeug in das System integriert werden.

A Repository

geschrieben von Nils Jahns, Felix Luther

A.1 Struktur vom Repository

Für das Projekt wurde auf Gitlab eine Gruppe angelegt. `https://gitlab.com/embedsprojekt_wise22` Ähnlich wie beim Vorgängerprojekt wurden drei Repositories in dieser Gruppe erstellt:

- Documentation (`https://gitlab.com/embedsprojekt_wise22/documentation`)
- FFT (`https://gitlab.com/embedsprojekt_wise22/fft`)
- KI (`https://gitlab.com/embedsprojekt_wise22/ki`)

A.2 Documentation Repository

Dieses Repository enthält die Quelldateien zu diesem Dokument, welches mit LaTeX erstellt wurde. Zudem ist eine Kopie der Quelldateien des Vorjahres im Unterordner `old/`

A.3 FFT Repository

Dieses Repository wurde für die Aufnahme mit dem eingebauten Mikrofon und die Vorverarbeitung der Daten für das neuronale Netz auf dem PYNQ-Board erstellt. Es enthält die Vivado-Projekte zum Lesen und Verarbeiten der Daten. Eine Kopie der Ergebnisse des Vorjahres ist im Unterordner `old/` zu finden.

A.4 KI Repository

Dieses Repository enthält ein Python-Skript (textitki-main/EmbedsKi.py), mit dem Trainingsdaten erzeugt und das Neuronale Netz trainiert werden kann. Ebenfalls enthalten ist eine Markdown-Datei, die bei der Installation und Inbetriebnahme hilft. Die Arbeit fand im Unterordner `old/` auf Basis des Vorjahres statt.

A.5 Aufsetzen eines manuellen Group-Runners

geschrieben von Felix Luther

Für die im Folgenden genannten Befehle werden Administrationsrechte vorausgesetzt. Es wird zudem angenommen, dass die offizielle GitLab Instanz verwendet wird, falls später zu einer selbst gehosteten Instanz gewechselt wird, muss diese entsprechend als Instanz URL beim Registrierungsprozess eingetragen werden. Eine Installation von Docker wird auch vorausgesetzt. Da die Installation von Docker sich von System zu System stark unterscheidet und viele Informationen hierzu gut auffindbar sind (s. `https://docs.docker.com/`), wurde auf diese Schritte im Folgenden verzichtet. Diese Schritte wurden auf folgenden Plattformen erfolgreich durchgeführt: `Debian`, `Ubuntu` und `Windows 10`. Falls es doch zu Problemen im Registrierungsprozess kommt, kann die Seite `https://docs.gitlab.com/runner/register/index.html` für weitere Informationen zu spezifischen Zielsystemen betrachtet werden.

1. `https://docs.gitlab.com/runner/install/index.html` aufrufen
2. Unter `Binaries` die Seite für das Zielsystem auswählen
3. Runner entsprechend den gegebenen Schritten downloaden und installieren
4. Runner Service mit `gitlab-runner start` starten
5. Registrierungsprozess mit `gitlab-runner register` anstoßen
   (a) Als Instanz Url `https://gitlab.com/` eingeben
   (b) Auf der Weboberfläche zum Zielprojekt navigieren
   (c) Auf den Unterpunkt `CI/CD->Runners` navigieren
   (d) Auf `Register a group runner` klicken
   (e) Den Registrierungstoken anzeigen/kopieren
   (f) Den Token beim Registrierungsprozess eintragen
   (g) Eine Beschreibung, bzw. einen Namen für den Runner eintragen
   (h) Optional komma-getrennt Tags eintragen
   (i) Optional eine Wartungsnotiz eintragen
   (j) `docker` als Executorëintragen
   (k) Image aus der Datei `.gitlab-ci.yml` im `Documentation` Repository eintragen, zzt. `registry.gitlab.com/islandoftex/images/texlive:TL2022-2023-02-05-full`
6. Den Runner mit `gitlab-runner start` starten
7. Zurück zu der Group-Runner Website wechseln
8. Rechts auf den `Edit` Knopf beim neu aufgetauchten Runner klicken
9. Den Haken bei `Run untagged jobs` setzen
10. Der Runner ist nun bereit und führt neu ankommende Jobs aus

A.6 Installation von TeX Live und TeXworks

geschrieben von Felix Luther

Im Folgenden werden die Schritte beschrieben um die TeX Live inklusive dem TeXworks Editor auf einem Windows System zu installieren. Dieses stellte die Umgebung dar, in welcher das Team an diesem Bericht gearbeitet hat. Die Installation auf Systemen mit einem anderen Betriebssystem kann sich unterscheiden, die Website `https://www.tug.org/texlive/` enthält für diesen Fall mehr Informationen.

1. Zu `https://www.tug.org/texlive/` navigieren
2. `install on windows` klicken
3. `install-tl-windows.exe` klicken
4. Warten bis der Download abgeschlossen ist
5. Installer ausführen
6. Punkt `Install` auswählen, Next Drücken
7. `Install` Knopf drücken
8. Auf den Extraktionsprozess warten
9. Unter `Specific mirror...` den geographisch nähesten auswääöhlen
10. Download starten, dieses kann **mehrere Stunden** dauern
11. Nach dem Download öffnet sich ein neues Menü
12. Sicherstellen, dass der Box `Install TeXworks front end` ausgewählt ist
13. `Install` drücken, dieses kann mehrere Stunden dauern
14. Hiernach sollte die Umgebung zur Arbeit mit LaTex zur Verfügung stehen

Literatur

[Alt+21] Philipp Altnickel u. a. *Gesten- und Objekterkennung durch schwache FPGAs in autonomen Fahrzeugen mittels neuronaler Netze*. Techn. Ber. Hochschule Bremen, 1. Sep. 2021. 153 S. URL: https://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/gestenerkennung-fpga-neuronale-netze-projekt-21.html (besucht am 30. 03. 2022).

[Alt+22] Philipp Altnickel u. a. *Personenerkennung durch schwache FPGAs in autonomen Fahrzeugen mittels Neuronaler Netze*. Techn. Ber. Hochschule Bremen, 1. März 2022. 57 S. URL: https://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/personenerkennung-fpga-neuronale-netze-projekt-2122.html (besucht am 30. 03. 2022).

[AMD23a] AMD/Xilinx. *Download Page: Xilinx Unified Web Installer*. Englisch. 2023. URL: https://www.xilinx.com/support/download.html (besucht am 28. 02. 2023).

[AMD23b] AMD/Xilinx. *Support Article 34573: „How Do I Create a New Account on Xilinx.com“*. Englisch. 2023. URL: https://support.xilinx.com/s/article/34573?language=en_US (besucht am 28. 02. 2023).

[Bre22] Jan Bredereke. „Enabling Neural Network Edge Computing on a Small Robot Vehicle“. Englisch. In: *Intelligent Distributed Computing XVI* (Bremen, Germany, 14.–16. Sep. 2022). Hrsg. von Kai Jander, Lars Braubach und Costin Badica. Studies in Computational Intelligence. Springer, 2022. URL: https://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/bredereke-enabling-nn-edge-computing-on-robot-2022.html (besucht am 10. 10. 2022). Im Erscheinen.

[Cro+14] Louise H Crockett u. a. *The Zynq book: embedded processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 all programmable SoC*. Strathclyde Academic Media, 2014.

[Dig17] Digilent. *PYNQ-Z1 Board Reference Manual*. Englisch. 2017. URL: https://digilent.com/reference/_media/reference/programmable-logic/pynq-z1/pynq-rm.pdf (besucht am 13. 02. 2023).

[Dig23] Digilent. *Digilent PYNQ-Z1 Product Page*. Englisch. 2023. URL: https://digilent.com/reference/programmable-logic/pynq-z1/ (besucht am 28. 02. 2023).

[FIN23] FINN. *FINN-Onnx*. 7. Feb. 2023. URL: https://finn.readthedocs.io/en/v0.2b/internals.html.

[Hag+22] Fynn Hagen u. a. *Neural Network on an FPGA for Speech Command Recognition on an Autonomous Vehicle*. Englisch. Techn. Ber. Hochschule Bremen, 8. März 2022. 49 S. URL: https://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/speech-recognition-fpga-neural-network-wp_embeds-2022.html (besucht am 30. 03. 2022). In preparation.

[Hin+12] Geoffrey E. Hinton u. a. *Improving neural networks by preventing co-adaptation of feature detectors*. 3. Juli 2012. URL: https://arxiv.org/pdf/1207.0580.pdf.

[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: https://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/neuronale-netze-fpgas-projekt-1920.html (besucht am 30. 03. 2022).

[Ko+15] Tom Ko u. a. „Audio augmentation for speech recognition". In: *Sixteenth annual conference of the international speech communication association*. 2015.

[Li+16] Xiaqing Li u. a. „Performance Analysis of GPU-Based Convolutional Neural Networks". In: *2016 45th International Conference on Parallel Processing (ICPP)*. 2016, S. 67–76. DOI: 10.1109/ICPP.2016.15.

[Liu+20] Yatian Liu u. a. *PDM-to-PCM Microphone Signal Conversion on FPGA Using CIC and FIR Filters*. 2020. URL: https://yatian-liu.github.io/public/PDM_PCM_Signal_Conversion_FPGA.pdf.

[Mra23a] Mirza Mrahorovic. *Xilinx /finn-examples*. 2023. URL: https://github.com/Xilinx/finn-examples/blob/main/finn_examples/notebooks/4_keyword_spotting.ipynb.

[Mra23b] Mirza Mrahorovic. *Xilinx /finn-examples*. 2023. URL: https://github.com/Xilinx/finn/blob/41740ed1a953c09dd2f87b03ebfde5f9d8a7d4f0/notebooks/end2end_example/cybersecurity/1-train-mlp-with-brevitas.ipynb (besucht am 25. 02. 2023).

[Mül+21] 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: https://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/bnns-on-fpgas-driving-projekt-2021.html (besucht am 30. 03. 2022).

[Mül21] Felix Müller. „Dynamisches Tiling auf schwachen FPGAs zur Objekterkennung mithilfe kleiner neuronaler Netze". Bachelorthesis. Hochschule Bremen, 23. Juni 2021. URL: https://homepages.hs-bremen.de/~jbredereke/de/forschung/veroeffentlichungen/mueller-bsc-thesis-2021.html (besucht am 30. 03. 2022).

[ONN23] ONNX. *ONNX*. 7. Feb. 2023. URL: https://onnx.ai/.

[Wes+22] Sören Westphal u. a. *Projekt ANN-KNX – Akustische Ansteuerung von KNX-Systemen mit neuronalen Netzen SoSe22*. Techn. Ber. Hochschule Bremen, 11. Aug. 2022. 80 S.

[Wes+23] Sören Westphal u. a. *Projekt ANN-KNX – Akustische Ansteuerung von KNX-Systemen mit neuronalen Netzen WiSe22-23*. Techn. Ber. Hochschule Bremen, 14. Feb. 2023. 80 S.

[Xil18] Inc. Xilinx. *Zynq-7000 SoC Data Sheet: Overview*. Englisch. DS190. v1.11.1. Juli 2018.

[Xil22] Inc. Xilinx. *AXI DMA v7.1 LogiCORE IP Product Guide*. Englisch. PG021. Apr. 2022.

[Xil23a] Xilinx. *Brevitas Documentation*. Englisch. 2023. URL: https://xilinx.github.io/brevitas/index.html (besucht am 27. 02. 2023).

[Xil23b] Xilinx. *Brevitas GitHub*. Englisch. 2023. URL: https://github.com/Xilinx/brevitas (besucht am 27. 02. 2023).

[Xil23c] Xilinx. *FINN Dataflow Builder API Documentation*. Englisch. 2023. URL: https://finn-dev.readthedocs.io/en/latest/source_code/finn.builder.html (besucht am 28. 02. 2023).

[Xil23d] Xilinx. *FINN-Examples*. Englisch. 2023. URL: https://github.com/Xilinx/finn-examples (besucht am 23. 02. 2023).

[Xil23e] Xilinx. *FINN-Homepage*. Englisch. 2023. URL: https://xilinx.github.io/finn/ (besucht am 23. 02. 2023).

[Xil23f] Xilinx. *PYNQ Project Page, Development Boards & Downloadable SD Card Images*. Englisch. 2023. URL: http://www.pynq.io/board.html (besucht am 28. 02. 2023).

[Xil23g] Xilinx. *Xilinx FINN System Requirements*. Englisch. 2023. URL: https://finn.readthedocs.io/en/latest/getting_started.html#system-requirements (besucht am 28. 02. 2023).

[Xil23h] Xilinx. *Xilinx Vivado System Requirements*. Englisch. 2023. URL: https://docs.xilinx.com/r/2022.1-English/ug973-vivado-release-notes-install-license/System-Requirements (besucht am 28. 02. 2023).

[Xil23i] Dev-Tree Xilinx FINN Compiler Repository. Englisch. 2023. URL: https://github.com/Xilinx/finn/tree/dev (besucht am 28. 02. 2023).