Deutsches Zentrum für Luft- und Raumfahrt
German Aerospace Center

Hochschule München
University of Applied Sciences

Analyse neuronaler Netze zur Objekterkennung auf Mikrocontrollern mit integrierter Neural Processing Unit

Leistungs- und Speichervergleich von neuronalen Netzarchitekturen

---

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Engineering (B.Eng.)

im Studiengang Mechatronik - Bachelor

Vorgelegt von:	Ralf Schmidt
Matrikelnummer:	29772621
Erstprüfer:	Prof. Dr.-Ing. Michael Hermann
Zweitprüfer:	Prof. Dr. Fabian Flohr
Betreuer:	Tim Albert Klaas
Abgabedatum:	30. September 2026

Eigenständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe und keine anderen als die angegebenen Hilfsmittel benutzt und die verwendete Literatur vollständig aufgeführt sowie Zitate kenntlich gemacht habe. Ich versichere ferner, dass die Arbeit noch nicht zu anderen Prüfungen vorgelegt wurde.

Oberschleißheim, 30.09.2026	
Ort, Datum	Unterschrift

Abstract

Name des Studenten:

Ralf Schmidt

Titel der Arbeit:

Analyse neuronaler Netze zur Objekterkennung auf Mikrocontrollern mit integrierter Neural Processing Unit

Keywords:

Mikrocontroller, NPU, Edge AI, Objekterkennung, TinyML, STM32N6570-DK

Abstract:

Mit der steigenden Anzahl von Machine-Learning-Anwendungen rückt auch die benötigte Hardware immer weiter in den Vordergrund. Eine übliche Anwendung ist die Auswertung eines vorab trainierten neuronalen Netzwerks zur Objekterkennung. Kann die gegebene Aufgabe lokal auf dem Microcontroller gelöst werden, entfällt die Notwendigkeit einer Verbindung zu einem externen Server und der Energie- und Platzbedarf sinkt gegenüber einer bereits lokalen Lösung mit externer Hardware (FPGA, Grafikkarte) erheblich.

Aus diesem Grund werden zunehmend Microcontroller mit spezieller Hardware, sogenannte NPUs angeboten. Diese sind speziell für die Arbeit mit Faltungsnetzen ausgelegt.

Diese Arbeit bestimmt anhand zeitgemäßer neuronaler Netze zur Objekterkennung die Leistungsfaktoren der integrierten NPU und versucht einen näheren Einblick in die Funktionsweise und die Taktung der NPU zu bieten. Als Testplattform dient ein STM32N6570-DK. Auf diesem wird ein Testsatz von Modellen hinsichtlich ihres Ressourcenverbrauchs und ihrer Laufzeiten analysiert. Zudem werden verschiedene Architekturen mit diversen Parametern getestet um die Funktionsweise der NPU genauer zu erforschen. So sollen Einsatzmöglichkeiten und eine optimierte Architektur für Neuronale Netze genauer ermittelt werden.

Inhaltsverzeichnis

Eigenständigkeitserklärung i

Abstract ii

Abbildungsverzeichnis v

Tabellenverzeichnis vi

Abkürzungsverzeichnis vii

1 Einführung 1

2 Theoretische Grundlagen und Funktionsweisen 4

2.1 NPU . . . 4

2.1.1 Aufbau . . . 4

2.1.2 Ablauf . . . 6

2.2 Neuronale Netze . . . 8

2.2.1 YOLO26n . . . 8

2.2.2 YOLOv8n . . . 9

2.2.3 MobileNetV3Small . . . 10

2.2.4 ToF-CNN . . . 11

2.3 Optimierungsverfahren . . . 14

2.3.1 Quantisierung . . . 14

2.3.2 Pruning . . . 15

3 Methodik 17

3.1 Prüfparameter . . . 17

3.2 Testablauf der existierenden Modelle . . . 17

3.3 Experimente zur Anpassung an die NPU Struktur . . . 19

4 Erwartungen der Experimente 21

4.1 Experiment 1 - Abhängigkeit zur Eingangsgröße . . . 21

4.2 Experiment 2 - Abhängigkeit von der Anzahl der Filter . . . 24

4.3 Experiment 3 - Genauere Auflösung zu Exp. 2 . . . 24

4.4 Experiment 4 - Unterschiede in Kernel Dimensionen 25
4.5 Experiment 5 - Abhängigkeit zur Anzahl der Schichten 25

5 Ergebnisse **27**
5.1 Vergleich der Modelle 27
5.1.1 Inferenzzeit 27
5.1.2 RAM-Usage 29
5.1.3 Gewichtsgröße 30
5.1.4 Modellgröße 31
5.1.5 Accuracy 31
5.2 Erweiterte Anpassung aus NPU-Struktur 33
5.2.1 Experiment 1 - Abh. zur Eingangsgröße 33
5.2.2 Experiment 2 - Abh. zur Anzahl der Filter 35
5.2.3 Experiment 3 - genauere Auflösung von Exp. 2 37
5.2.4 Experiment 4 - Abh. zur Filter Dimension 38
5.2.5 Experiment 5 - Abh. zur Anzahl der Schichten 42

6 Diskussion **45**
6.1 Einfluss der Modelloptimierung 45
6.2 Einfluss der NPU-Architektur 46
6.2.1 Einfluss der Eingangsgröße 46
6.2.2 Einfluss der Filteranzahl 47
6.2.3 Grenzbereich der Filteranzahl 47
6.2.4 Einfluss von Kernelgröße und Stride 48
6.2.5 Einfluss der Netzwerktiefe 48
6.3 Bewertung der praktischen Leistungsfähigkeit 49
6.4 Limitationen 49

7 Fazit **51**

Literatur **53**

Abbildungsverzeichnis

1.1 Übersicht Cloud- und Edge-Computing 2

2.1 Ablaufdiagramm einer Inferenz mit MCU / NPU Belastung [17] . . . 7
2.2 Ablaufdiagramm einer Inferenz im Controller Modus [17] 8
2.3 Konzept der Datenerstellung ToF-CNN 12
2.4 Übersicht ToF-CNN Architektur 13

5.1 Vergleich der Inferenzzeiten vor und nach der Anpassung an die NPU 27
5.2 Vergleich der RAM-Nutzung vor und nach der Anpassung an die NPU 29
5.3 Vergleich der Weight-Größen vor und nach der Anpassung an die NPU 30
5.4 Vergleich der Modellgrößen vor und nach der Anpassung an die NPU 31
5.5 Vergleich der Accuracy der YOLO-Modelle vor und nach der Optimierung 32
5.6 Accuracy Vergleich der Klassifizierungs Modelle 32
5.7 Inferenzzeit bei Steigender Eingangsgröße 34
5.8 Linearität vor und nach dem Sprung in Inferenzzeit 35
5.9 Inferenzzeit bei steigender Filteranzahl 35
5.10 Inferenzzeit anteilig NPU/CPU 36
5.11 RAM-Nutzung 36
5.12 Modell-Größe 37
5.13 Inferenz am Grenzfall - 512 38
5.14 RAM Nutzung am Grenzfall - 512 38
5.15 Inferenz mit Stride 1 39
5.16 Inferenz mit Stride 2 40
5.17 Inferenz mit Stride 3 41
5.18 Inferenz mit Stride 4 42
5.19 Inferenz in Abhängigkeit von Anzahl Faltungsebenen 43
5.20 RAM-Nutzung der Aktivierungen in Abhängigkeit von Anzahl Faltungsebenen 43
5.21 Speichergröße der Gewichte in Abhängigkeit von Anzahl Faltungsebenen 44
5.22 Modelgröße in Abhängigkeit von Anzahl Faltungsebenen 44

Tabellenverzeichnis

2.1 Übersicht der Processing Units auf Neural Art 14 [17] 5
2.2 Spezielle Units der NPU [17] 5
2.3 Arbeitsspeicher Übersicht der NPU 6
2.4 Epochen der NPU [17] 6
2.5 Netzwerkaufbau ToF-CNN 12
2.6 Vergleich zwischen Pruned und Nicht Pruned CNN 16

4.1 Übersicht über die Begrenzungen der NPU-Conv2D-Parameter [18] . 25

5.1 Senkungsfaktoren der Modelle nach der Optimierung. 28
5.2 Inferenzzeit nach Eingangsgröße 33
5.3 Inferenzzeit im Randbereich 34

Abkürzungsverzeichnis

CNN Convolutional Neural Network

COCO Common Objects in Context

CPU Central Processing Unit

DMA Direct Memory Access

FPGA Field-Programmable Gate Array

FPS Frames Per Second

FSM Finite State Machine

GPU Graphics Processing Unit

HW Hardware

INT8 Integer mit 8 Bit

MAC Multiply-Accumulate Operation

MCU Microcontroller Unit

ML Machine Learning

mAP mean Average Precision

NAS Neural Architecture Search

NMS Non-Maximum Suppression

NN Neural Network

NPU Neural Processing Unit

RAM Random Access Memory

ReLU Rectified Linear Unit

SRAM Static Random Access Memory

SW Software

ToF Time of Flight

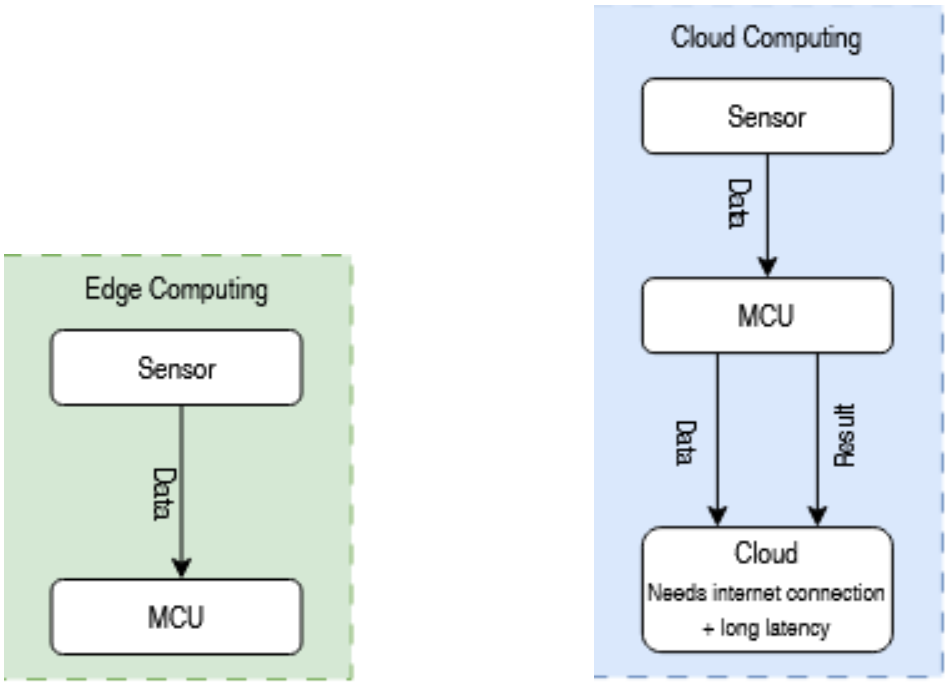
TOPS Trillion Operations Per Second

YOLO You Only Look Once

1 Einführung

Der Einsatz von Methoden des maschinellen Lernens (Machine Learning, ML) hat in den vergangenen Jahren in zahlreichen technischen Anwendungsbereichen stark zugenommen. Insbesondere neuronale Netze (NNs) ermöglichen die automatisierte Verarbeitung und Interpretation großer Mengen an Sensordaten und bilden dadurch eine wichtige Grundlage für Anwendungen mit Objekterkennung und Bildklassifikation. Mit der zunehmenden Verbreitung solcher Anwendungen verändern sich jedoch auch die Anforderungen an die dafür erforderlichen Rechenhardware. Neben der reinen Rechenleistung spielen insbesondere der Energieverbrauch, der verfügbare Speicher und die physikalischen Größenbegrenzungen eine entscheidende Rolle, wenn ML-Modelle in eingebetteten Systemen eingesetzt werden sollen. Eine Möglichkeit zur Ausführung neuronaler Netze besteht darin, die Berechnung auf einem externen Servern oder einer dedizierten Grafikkarte durchzuführen. Diese weisen allerdings meist einen großen Stromverbrauch und einen großen physikalischen Fußabdruck auf. Die Ausführung über eine Datenverbindung auf einen externen Server ist das sogenannte Cloud Computing [1, 3]. Die Sensordaten müssen in diesem Fall an ein externes System übertragen und die Ergebnisse anschließend zurück an den Controller übermittelt werden. Dieser Ansatz setzt jedoch eine geeignete Kommunikationsverbindung voraus und verursacht zusätzliche Latenzen sowie einen erhöhten Energiebedarf durch die Datenübertragung. Darüber hinaus können Anforderungen an Datenschutz, Verfügbarkeit und Echtzeitfähigkeit die Verwendung einer externen Infrastruktur einschränken. Eine lokale Verarbeitung auf dem Endgerät auch bekannt als Edge-Gerät, bietet hier eine Alternative. Das sogenannte Edge Computing [15]. Hierbei werden die Auswertungen und Entscheidungen direkt am Ort der Datenerfassung ausgeführt.

Für die lokale Ausführung rechenintensiver neuronaler Netze werden bislang häufig leistungsfähigere Komponenten wie Grafikkarten oder Field Programmable Gate Arrays (FPGAs)[11] eingesetzt. Diese bieten zwar eine hohe Rechenleistung und können speziell für die parallele Verarbeitung neuronaler Netze optimiert werden, benötigen jedoch im Vergleich zu einem klassischen Mikrocontroller zusätzlichen Bauraum und Energie. Für Anwendungen mit strengen Anforderungen an Größe und



(a) Edge Computing (b) Cloud Computing

Abbildung 1.1: Übersicht Cloud- und Edge-Computing

Energieverbrauch stellt dies eine wesentliche Einschränkung dar. Gleichzeitig führt die zunehmende Verbreitung von ML-Anwendungen zu einem wachsenden Bedarf an leistungsfähigeren eingebetteten Recheneinheiten. Als Reaktion auf diese Entwicklung werden moderne Mikrocontroller (MCUs) zunehmend um spezialisierte Hardware zur Beschleunigung neuronaler Netze erweitert. Ein Beispiel hierfür ist die Neural Processing Unit (NPU). Sie ist auf die effiziente Verarbeitung der für neuronale Netze typischen Operationen ausgelegt. Insbesondere Faltungsnetze (Convolutional Neural Networks, CNN), die bei der Bildverarbeitung und Objekterkennung eine zentrale Rolle spielen, können von einer solchen Beschleunigung profitieren. Dadurch eröffnet sich die Möglichkeit, komplexere neuronale Netze direkt auf einem Mikrocontroller auszuführen, ohne auf externe Rechenhardware oder eine permanente Netzwerkverbindung angewiesen zu sein. Die tatsächliche Nutzbarkeit einer integrierten NPU hängt dabei nicht ausschließlich von ihrer theoretischen Rechenleistung ab[4]. Von Bedeutung sind unter anderem der Speicherbedarf der verwendeten Modelle, deren Laufzeit sowie die Unterstützung unterschiedlicher Netzwerkoperationen und Datenformate. Zusätzlich können Optimierungsverfahren wie Quantisierung, Pruning, die Anpassung der Netzwerkarchitektur oder die gezielte Nutzung der verfügbaren Hardware die Ausführung wesentlich beeinflussen. Für die Beurteilung der praktischen Einsatzmöglichkeiten ist daher eine Untersuchung anhand aktueller Modelle erforderlich. Ziel dieser Arbeit ist es, die Leistungsfähigkeit der in einem STM32N6 integrierten NPU anhand aktueller neuronaler Netze zur Objekterkennung und Klassi-

fizierung systematisch zu untersuchen. Hierzu wird ein Testsatz verschiedener Modelle hinsichtlich ihres Ressourcenverbrauchs und ihrer Laufzeiten analysiert. Darüber hinaus soll anhand einiger Experimente einige Anwendungsregeln zur Erstellung eines optimierten Netzes gefunden werden. Der Fokus liegt dabei besonders auf der viel benutzten Faltunsschicht. Auf diese Weise sollen die praktischen Leistungsgrenzen der Plattform bestimmt und daraus die Einsatzmöglichkeiten eines NPU-beschleunigten Mikrocontrollers für Anwendungen der lokalen Objekterkennung abgeleitet werden. Damit soll die Arbeit einen Beitrag zur Bewertung der Frage leisten, inwieweit moderne Mikrocontroller durch ihre NPU die bisher eingesetzte externe Rechenhardware in ML-Anwendungen ersetzen können.

2 Theoretische Grundlagen und Funktionsweisen

2.1 NPU

2.1.1 Aufbau

Eine NPU ist ein, auf häufig benutzte Berechnungen in Neuronalen Netzen, spezialisierter Prozessor. Im Vergleich zu einer CPU welche auf generelle Aufgaben optimiert ist, hat die NPU Hardware-Strukturen, die viele der mathematischen Berechnungen eines Netzdurchlaufes(Inferenz) parallel durchführen können. Durch diese Spezialisierung können NNs mit geringerer Latenz und meist geringerem Stromverbrauch [4] durchgeführt werden.

In der Welt der Embedded Systems werden NPUs für NN-Inferenzen genutzt. Eine Inferenz beschreibt hierbei den Ablauf von den eingespeisten Daten bis hin zu der Vorhersage bzw. der Klassifikation. Dabei werden Operationen wie Faltungen, Matrixsmultiplikationen und Aktivierungsfunktionen benutzt. Da die NPU diese Operationen mit besserer Effizienz berechnen kann, ist sie besonders für den Einsatz auf ressourcenarmen Systemen geeignet. Das STM32N6570-DK nutzt einen Neural-ART 14 als NPU [17]. Dieser ist speziell auf quantisierte CNNs ausgelegt und ermöglicht die Durchführung einer Inferenz mit minimaler Belastung der CPU. Quantisiert bedeutet hier, dass das CNN lediglich Gewichte mit Werten des Formats 8-bit-Integer aufweist. Die CPU dient im Zusammenspiel mit der NPU lediglich als Taktgeber bzw. als Kontrolleinheit.

Die NPU nutzt eine Streaming-basierte Architektur in der Hardware mit speziellen Hardware Beschleunigern. Sie fungieren als intelligente DMA-Einheiten (Direct Memory Access). Sie können selbstständig Daten aus dem NPU-internen oder externen Speicher lesen und schreiben. Sobald die NPU konfiguriert und initialisiert ist, kontrolliert sie autonom den Speicher-zu-Speicher Transfer. Sie zieht Daten aus dem externen Speicher, übergibt diese an geeignete Processing Units und schreibt

nach der Berechnung die Ergebnisse erneut in den externen Speicher. Sobald der Prozess begonnen hat, bleiben die Einstellungen erhalten bis alle DMA-Ausgänge eine ”end-of-transfer event notification” gegeben haben. Diese dienen also primär der Synchronisierung des Datenflusses. Dadurch werden Inkonsistenzen und Datenkorruption verhindert. Die Neural-Art 14 besitzt 4 Arten von Processing Units:

Processing Unit	**Beschreibung**
CONV_ACC	Haupteinheit, die Faltungsoperationen durchführt. Maximum von ∼578 GOPS @1GHz
POOL_ACC	Einheit zum Durchführen von Pooling Operationen wie min, max und average pooling.
ACTIV_ACC	Führt die Aktivierungsfunktion für das NN aus: Logistic, TanH, ReLU, PReLU, ...
ARITH_ACC	Führt die arithmetischen Operationen wie Elementweise Addition, Subtraction, Multiplication, etc. aus.

Tabelle 2.1: Übersicht der Processing Units auf Neural Art 14 [17]

Desweiteren besitzt die NPU noch 4 weitere spezielle Units die das Memory Subsystem unterstützen.

Unit	**Beschreibung**
DEC_UNIT	Dekomprimiert die Gewichte der Faltungskerne.
RECBUF_UNIT	Unit zum Einfügen eines Buffers zwischen zwei processing Units. Verhindert damit potenzielle Blockade, „Starvation“.
EPOCH_CTRL	Eine finite state machine (fsm) die Anweisungen aus einem Commandstream (blob) zur Ausführung von Epochen entschlüsselt.
DEBUG & TRACE UNIT	Debug Unit zur Sammlung verschiedener interner Signale zur Überwachung.

Tabelle 2.2: Spezielle Units der NPU [17]

Das Memory Subsystem sorgt dafür das die Daten rechtzeitig zur Verarbeitung bereit sind. Die benötigte Größe des Speichers ist abhängig von der Komplexität des NNs. Also z.B. die Anzahl an Schichten und Dimensionen der Eingangsdaten. Der Host (hier die CPU) teilt sich diesen Speicher mit der NPU.

Anzumerken ist, dass das STM32N6570-DK über einen NPU-internen und externen RAM verfügt. In STM32Cube AI-Studio werden diese genauer aufgeführt. Hierbei werden nur Speicher aufgezählt, die von der NPU nutzbar sind:

Treiber	**Name**	**Größe KB**	**Intern/Extern**
AXISRAM2	cpuRAM2	1024	Extern
AXISRAM3	npuRAM3	448	Intern
AXISRAM4	npuRAM4	448	Intern
AXISRAM5	npuRAM5	448	Intern
AXISRAM6	npuRAM6	448	Intern
xSPI1	hyperRAM	32768	Extern
xSPI2	octoFlash	114688	Extern

Tabelle 2.3: Arbeitsspeicher Übersicht der NPU

2.1.2 Ablauf

In der Regel kann ein NN nicht vollständig in der verfügbaren Hardware wiedergegeben werden. Daher wird es in kleinere Unterteile zerlegt, welche passend dimensioniert sind. Diese nennen sich NPU Epochen oder nur Epochen. Die Epochen werden durch den in STM32Cube AI Studio integrierten Compiler automatisch erstellt. Insgesamt gibt es 4 verschiedene Arten von Epochen

Name	**Beschreibung**
Meta-Epoche	Bezeichnet eine Menge von HW-Epochen die über Befehlsströme (Command Streams) gesteuert werden. Wird durch die Epochen-Controller-Einheit gesteuert.
HW-Epoche	Bezeichnet eine Menge an Operationen die vollständig auf den Hardware-Ressourcen der NPU abgebildet werden.
SW-Epoche	Bezeichnet eine Menge an Operationen die an den Host ausgelagert werden. Diese werden nicht hardwarebeschleunigt ausgeführt.
Hybrid-Epoche	Spezieller Fall bei dem ein Teil der Operationen in Software ausgeführt wird aber durch zuvor definierte HW-Epochen unterstützt wird.

Tabelle 2.4: Epochen der NPU [17]

Jede Epoche hat 3 Phasen:

- ”Pre-Op” ist die Initialisierungsphase
- ”HW-Op” behandelt den Datentransfer bis zur „end-of-transfer notification“
- ”Post-Op” ist die Deinitialisierungsphase.

Nachdem das Modell in Epochen unterteilt ist, übergibt die CPU die Befehlsströme der einzelnen Epochen an die NPU. Grundsätzlich ist der Mikrocontroller an der Pre-Op und der Post-Op Phase der NPU beteiligt. Wenn es sich um eine SW-Epoche handelt, ist sie vollständig daran beteiligt. Sollte das gesamte Netz auf der Hardware der NPU abbildbar sein, gibt es nur HW- und Meta-Epochen. In diesem Fall ist die CPU ca. 10 – 15% der Inferenz Zeit beschäftigt.

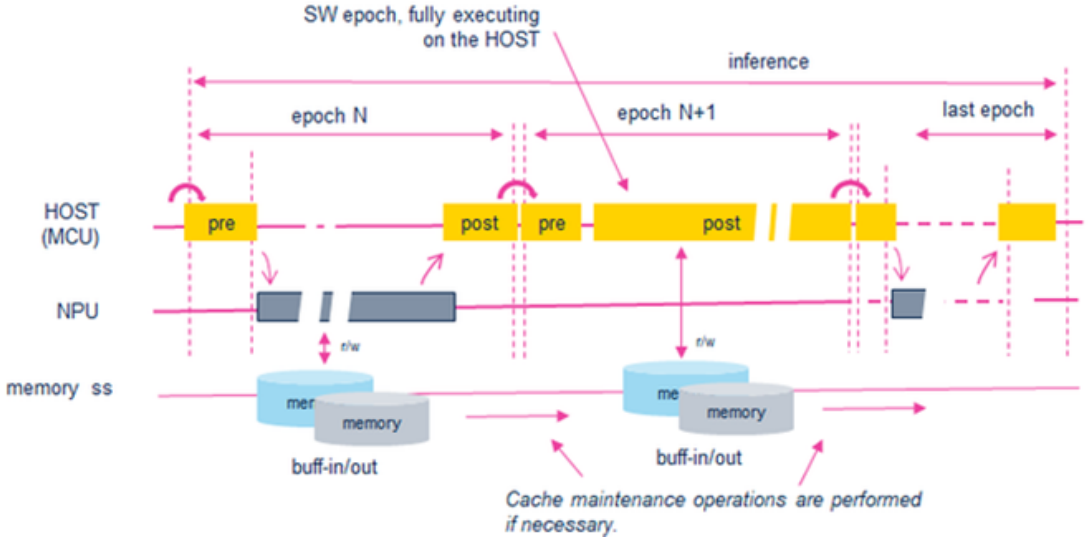


Abbildung 2.1: Ablaufdiagramm einer Inferenz mit MCU / NPU Belastung [17]

Es ist auch möglich einen Epochencontroller zu aktivieren. Dieser beschreibt eine Finite State Machine (FSM) innerhalb der NPU. Er kann einfache Mikroinstruktionen aus dem Befehlsstrom, auslesen und ausführen. Dieser enthält grundlegende Steuerbefehle wie Lesen, Speichern, Abfragen, schrittweises Ausführen und Stoppen. Dadurch kann der Epochencontroller die für die Inferenz benötigten Verarbeitungseinheiten direkt innerhalb der Hardware konfigurieren, ohne dass dafür ständig der MCU eingreifen muss. Die CPU des Mikrocontrollers wird dadurch entlastet und kann währenddessen andere Aufgaben übernehmen.

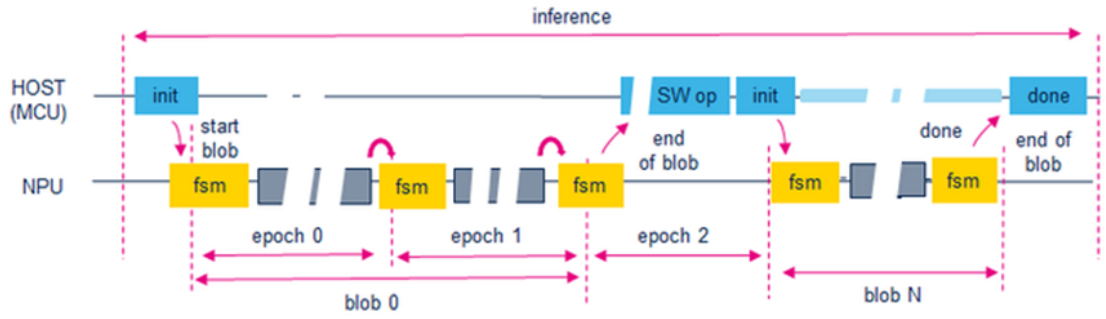


Abbildung 2.2: Ablaufdiagramm einer Inferenz im Controller Modus [17]

2.2 Neuronale Netze

Im Verlauf der Arbeit wurden folgende Neuronale Netze betrachtet:

2.2.1 YOLO26n

YOLO26n [8] ist die kleinste Variante der aktuellen YOLO26-Modellfamilie und wurde speziell mit Blick auf Echtzeit- und Edge-Anwendungen entwickelt. Das „n“ steht dabei für „nano“ und kennzeichnet die ressourcenärmste Modellvariante. Wie andere YOLO-Modelle besteht das Netzwerk grundsätzlich aus einem Backbone zur Merkmalsextraktion, einem Neck zur Zusammenführung von Merkmalen verschiedener Auflösungen und einem Detection Head zur Bestimmung von Objektposition und Objektklasse. Eine wesentliche Neuerung von YOLO26 im Vergleich zu YOLO8 ist der End-to-End-Ansatz, bei dem der standardmäßig verwendete Postprocessing-Schritt, die Non-Maximum Suppression (NMS) entfällt.

Die veröffentlichten YOLO26-Modelle wurden nicht von Grund auf mit dem COCO-Datensatz [10] trainiert. Zunächst erfolgte ein Pretraining auf Objects365v1 über 150 Epochen, anschließend wurden die Modelle auf dem COCO-Datensatz feinjustiert.

Die heutige Bedeutung von YOLO26 liegt insbesondere in der zunehmenden Verlagerung von künstlicher Intelligenz auf Edge-Geräte und eingebettete Systeme. Genau für diese Einsatzgebiete ist die Reduzierung von Rechen- und Speicheraufwand sowie die Vereinfachung des Inferenzablaufs relevant. Das von Ultralytics angegebene [25] YOLO26n besitzt lediglich etwa 2,4 Millionen Parameter und 5,4 GFLOPs und erreicht auf dem COCO-Datensatz eine mAP (mean Average Precision) von 40,9 bei einer Eingangsauflösung von 640 × 640 Pixeln.

Für die Untersuchung auf dem STM32N6570-DK wurde YOLO26n ausgewählt, da es ein aktuelles, für Edge-Anwendungen optimiertes Objekterkennungsnetz darstellt. Gleichzeitig ist die geringe Modellgröße interessant für die Untersuchung der Neural-ART-NPU, da sich dadurch untersuchen lässt, wie sich ein modernes, vergleichsweise komplexes Objekterkennungsnetz auf der begrenzten Hardware eines Mikrocontrollers verhält. Besonders relevant ist dabei die Frage, welche Operationen die NPU tatsächlich hardwarebeschleunigt ausführen kann und welche gegebenenfalls auf dem Host-Prozessor ausgeführt werden müssen. YOLO26n stellt somit den modernen und auf Edge-Inferenz optimierten Vertreter innerhalb des Vergleichs dar.

2.2.2 YOLOv8n

YOLOv8n ist die kleinste Variante der YOLOv8-Modellfamilie von Ultralytics und wurde als leichtgewichtiges Echtzeit-Objekterkennungsnetz entwickelt. Die Architektur besteht wie die anderen YOLO-Modelle aus Backbone, Neck und Detection Head.

Das YOLOv8n-Modell wurde ursprünglich ebenfalls auf großen Datensätzen wie COCO vortrainiert. Für eine eigene Anwendung kann das vortrainierte Modell anschließend mittels Transfer Learning auf einen spezifischen Datensatz angepasst werden. Ultralytics stellt hierfür bereits vortrainierte Gewichte zur Verfügung, die anschließend mit einem eigenen Datensatz weitertrainiert werden können. Dadurch muss das Netzwerk nicht vollständig von zufälligen Gewichten ausgehend trainiert werden, was insbesondere bei kleineren Datensätzen von Vorteil ist.

YOLOv8 besitzt trotz seines bereits vergleichsweise hohen Alters weiterhin eine große Bedeutung. Es wurde zu einem weit verbreiteten Referenzmodell für Echtzeit-Objekterkennung und wird aufgrund seiner guten Kombination aus Erkennungsgenauigkeit, Geschwindigkeit und vergleichsweise geringer Modellgröße häufig für Edge-AI-Anwendungen eingesetzt. Das von Ultralytics angegebene [7, 26] YOLOv8n besitzt etwa 3,2 Millionen Parameter und 8,7 GFLOPs. Auf COCO erreicht es bei 640×640 Pixeln eine mAP von 37,3.

Für die Analyse auf dem STM32N6570-DK wurde YOLOv8n insbesondere als Vergleichsmodell zu YOLO26n ausgewählt. Beide Netzwerke lösen dieselbe Aufgabe, unterscheiden sich jedoch in ihrer Architektur und insbesondere in der Ausführung des Detection Heads. Dadurch kann untersucht werden, wie sich die Weiterentwicklung der YOLO-Architektur auf die Ausführbarkeit und Effizienz auf der Neural-ART-NPU auswirkt. YOLOv8n dient damit als etablierte Referenz, anhand derer die Eigenschaften des neueren YOLO26n besser eingeordnet werden können. Gleichzeitig

ist das Modell klein genug, um eine Untersuchung auf einem Mikrocontroller-basierten Edge-System überhaupt realistisch durchführen zu können.

2.2.3 MobileNetV3Small

MobileNetV3-Small [5] unterscheidet sich grundlegend von den beiden YOLO-Modellen. Während YOLO26n und YOLOv8n für die Objektdetektion entwickelt wurden, handelt es sich bei MobileNetV3-Small um ein Klassifikationsnetzwerk. Das Netzwerk wurde speziell für Geräte mit begrenzten Rechenressourcen entwickelt.

Eine Besonderheit bei der Entwicklung von MobileNetV3 ist, dass die Architektur nicht ausschließlich manuell entworfen wurde. Die Autoren verwendeten eine Kombination aus hardwarebewusster Neural Architecture Search (NAS) und dem NetAdapt-Verfahren, um die Architektur gezielt hinsichtlich Rechenaufwand und Latenz auf mobilen Prozessoren zu optimieren.

Das ursprünglich vorgestellte MobileNetV3-Small besitzt lediglich etwa 2,54 Millionen Parameter und benötigt ungefähr 56,5 MFLOPs pro Bild bei der ursprünglichen Modellkonfiguration. Die deutlich verringerte Anzahl an MFLOPs entsteht durch:

- **Unterschiedliche Parameterverteilung:** Ein signifikanter Anteil der $2,54$ Millionen Parameter befindet sich in den finalen Fully-Connected-Schichten des Klassifikations-Heads. Im Gegensatz zu Faltungsschichten, deren Gewichte über die gesamte Bildfläche ($H \times W$) wiederverwendet werden, wird jeder Parameter in einer Fully-Connected-Schicht pro Inferenz nur genau einmal multipliziert (FLOPs $\approx$ Parameter).
- **Aufgabenbedingte Architekturunterschiede:** Als Klassifikationsnetzwerk reduziert MobileNetV3-Small die räumliche Auflösung über Global Average Pooling frühzeitig auf 1×1. YOLO-Netzwerke hingegen müssen für die Objekterkennung hochauflösende Merkmalskarten (z. B. 80×80 Pixel) im Neck über mehrere Skalen hinweg aufrechterhalten, was zu einer hohen FLOP-Dichte führt.

Das Modell mit den veröffentlichten ImageNet-Gewichten erreichte eine Top-1-Accuracy von etwa 67,7 %. Damit zeigt das Modell, dass auch mit einem sehr geringen Rechenaufwand eine für viele Anwendungen ausreichende Klassifikations-Leistung erreicht werden kann.

Die Bedeutung von MobileNetV3-Small liegt insbesondere bei Mobile-, Embedded- und Edge-AI-Anwendungen. Das Modell wurde bereits bei seiner Entwicklung auf

eine möglichst effiziente Ausführung auf ressourcenbeschränkter Hardware ausgelegt. Damit stellt es einen interessanten Kontrast zu den komplexen YOLO-Modellen dar.

Für die Analyse auf dem STM32N6570-DK wurde MobileNetV3-Small deshalb als leichtgewichtiges Klassifikationsnetz ausgewählt. Während YOLO26n und YOLOv8n relativ komplexe Objekterkennungsaufgaben mit Bounding Boxes durchführen, besitzt MobileNetV3-Small eine deutlich einfachere Ausgangsstruktur. Dadurch kann untersucht werden, wie sich ein kleines und für mobile Hardware optimiertes CNN auf der Neural-ART-NPU verhält. Außerdem ermöglicht der Vergleich eine Untersuchung, ob die NPU mit einem klassischen Klassifikationsnetz effizienter umgehen kann als mit den komplexeren Detection-Netzwerken.

2.2.4 ToF-CNN

Als kleinstes Modell wurde ein CNN-Klassifizierungsmodell auf der Basis von Time-of-Flight-Sensordaten entwickelt. Ziel ist hiermit zu sehen auf welche Geschwindigkeiten die NPU kommen kann, sofern man sich an den in Abschnitt 5 gefundenen Einschränkungen orientiert. Das Modell wurde auf 5 Klassen ausgelegt:

- Würfel
- Kugel
- Tasse
- Zylinder
- Kein Objekt

Der Datensatz (siehe `Anhang: tof_cnn_dataset.csv`) für das Netzwerk wurde über den Mikrocontroller gesammelt. Zunächst wurden die ToF Daten über den Microcontroller an den PC gesendet und die wirkliche Klasse manuell beschriftet. Hierbei speichert ein Python Script jede Sekunde einen ToF-Frame in die CSV mit dem Label der ersten Klasse. Nach 300 Sekunden wird die nächste Klasse gestartet. Der Sensor konnte währenddessen frei um das Objekt bewegt und die Distanz bzw. der Blickwinkel verändert werden. Die Distanzen variierten dabei zwischen 50 und 500 mm. Dadurch ergibt sich ein Datensatz mit 300 Samples pro Klasse. (siehe Abbildung 2.3) Dieser Datensatz wurde danach in 80 % Trainingsdaten und 20 % Validierungsdaten aufgeteilt.

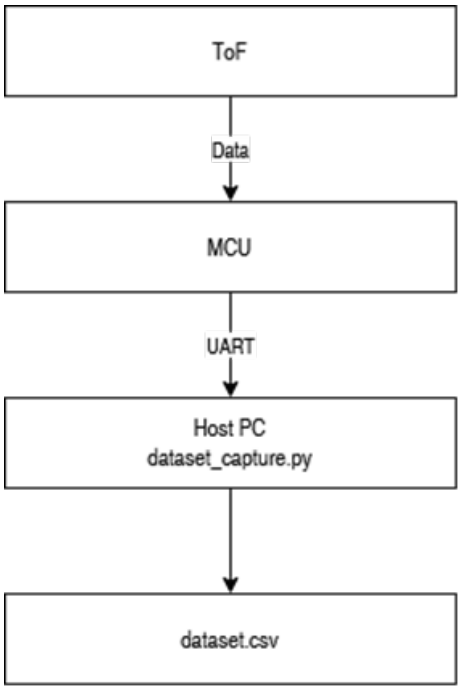


Abbildung 2.3: Konzept der Datenerstellung ToF-CNN

Das Netz wurde mit der folgenden Netzwerkarchitektur erstellt:

Schicht	**Ausgabe/Funktion**	**Aktivierung**
Input	Eingabedaten mit input_shape	-
Conv2D	8 Filter, 3x3 Kern	ReLU
Conv2D	16 Filter, 3x3 Kern	ReLU
Flatten	Umwandlung der Feature Maps in einen Vektor	-
Dense	32 Neuronen	ReLU
Output	5 Neuronen (Anz. Klassen)	Softmax

Tabelle 2.5: Netzwerkaufbau ToF-CNN

Mit den folgende Dimensionen für das CNN.

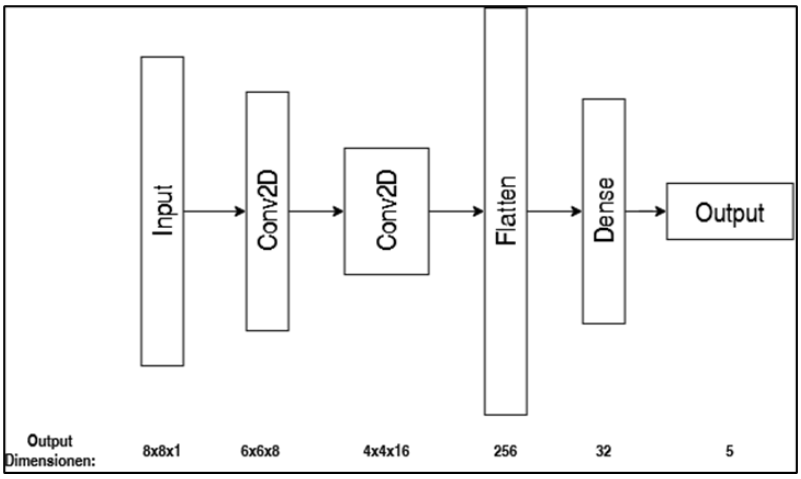


Abbildung 2.4: Übersicht ToF-CNN Architektur

Die **erste Faltungsschicht** verwendet 8 Filter mit einer Größe von 3×3. Jeder Filter untersucht einen kleinen lokalen Bereich der 8×8-ToF-Matrix. Dadurch können zunächst einfache Merkmale aus den Rohdaten extrahiert werden. Dazu gehören beispielsweise lokale Entfernungsänderungen, Kanten innerhalb der Distanzkarte oder zusammenhängende Bereiche ähnlicher Entfernung. Die geringe Anzahl von lediglich acht Filtern ist dabei bewusst gewählt. Aufgrund der geringen räumlichen Auflösung der 8×8-ToF-Eingabedaten ist eine starke Vergrößerung der Kanalzahl nur eingeschränkt sinnvoll. Untersuchungen mit 8×8-ToF-Sensoren zeigen, dass eine Erhöhung der Anzahl der Filter zwar die Modellkapazität steigern kann, gleichzeitig jedoch zu einer erheblichen Zunahme von Inferenzzeit und Energieverbrauch führt, während der Zugewinn an Klassifikationsgenauigkeit vergleichsweise gering ausfallen kann [13].

Die **zweite Faltungsschicht** erweitert die Anzahl der Feature Maps von 8 auf 16. Während die erste Schicht eher grundlegende lokale Strukturen erfasst, kann die zweite Schicht daraus komplexere Merkmale bilden. Beispielsweise können Kombinationen mehrerer lokaler Distanzmuster relevant sein, um zwischen unterschiedlichen Objektformen oder Positionen zu unterscheiden.

Nach den beiden Faltungsschichten werden die Feature Maps mit **Flatten** in einen eindimensionalen Vektor überführt. Dieser enthält die Merkmale, die zuvor aus der räumlichen ToF-Struktur extrahiert wurden.

Die anschließende **Dense-Schicht** mit 32 Neuronen übernimmt die Kombination dieser Merkmale. Hier kann das Netzwerk beispielsweise lernen, dass eine bestimmte Kombination aus räumlicher Ausdehnung, Entfernung und Position typisch für eine bestimmte Objektklasse ist. Die relativ geringe Größe der Dense-Schicht reduziert gleichzeitig den Speicher- und Rechenaufwand.

Das Modell soll eine minimalistische und realistische Version eines CNNs darstellen. Das Modell besitzt mit rund 9,6 k trainierbaren Parametern weniger als 10.000 Parameter. Damit ist es im Vergleich zu den anderen Modellen praktisch vernachlässigbar. Da es auf eine komplett andere, viel kleinere Eingangsgröße ausgelegt ist, erreicht es wesentlich schnellere Inferenzen.

Im Vergleich stellt dieses Modell also ein sehr kleines und stark spezialisiertes Netz dar. Das ToF-CNN zeigt, wie gering die Inferenzzeit für eine sehr kleine, NPU-günstige Netzwerkarchitektur werden kann. Durch die Spezialisierung und die geringe Größe sind die Inferenzzeiten teils um den Faktor 1000 geringer als bei den anderen Modellen.

2.3 Optimierungsverfahren

Im Bezug auf NNs werden im Verlauf der Arbeit häufig die Begriffe Quantisierung und Pruning aufgegriffen. Diese werden für das Verständnis hier genauer erklärt. Darüber hinaus werden Vermutungen angestellt wie sich diese Optimierungen auf das Modell auswirken.

2.3.1 Quantisierung

Da die Neural-ART-NPU keine Floating-Point-Berechnungen unterstützt und auf Festkommazahlen ausgelegt ist, müssen für eine vollständige Hardwarebeschleunigung der geeignete Integer-Datentyp verwendet werden. Es ist eine Reduktion in Modellgröße vom Faktor 4 zu erwarten. Das kommt aus der Typenänderung von Float 32 zu Int 8.

Die hier angewendete Quantisierung verwendet “Post-Training Quantization” (PTQ) [6, 12]. Hierbei werden Gleitkommawerte durch einen Skalierungsfaktor s auf einen diskreten Wertebereich von 8-Bit-Integern abgebildet. Bei einer vereinfachten symmetrischen INT8-Quantisierung wird der quantisierte Wert $Q(x)$ durch

$$Q(x) = \text{clip}\left(\text{round}\left(\frac{x}{s}\right), -128, 127\right) \tag{2.1}$$

bestimmt. Dabei bezeichnet x den ursprünglichen Gleitkommawert, s den Skalierungsfaktor und $Q(x)$ den resultierenden quantisierten Integerwert. Der Skalierungsfaktor

kann bei einer symmetrischen Quantisierung aus dem maximalen Betrag des Wertebereichs bestimmt werden:

$$s = \frac{\max(|x_{\min}|, |x_{\max}|)}{127}. \tag{2.2}$$

Die Funktion round() führt dabei die Rundung auf den nächstgelegenen ganzzahligen Wert durch, während clip() sicherstellt, dass der Wertebereich von -128 bis 127 nicht überschritten wird. Durch diesen Vorgang wird der kontinuierliche Wertebereich der Gleitkommadarstellung auf die diskreten 256 möglichen Werte einer 8-Bit-Integerdarstellung abgebildet.

Bei der PTQ werden die für diese Abbildung erforderlichen Quantisierungsparameter nach dem Training des Netzwerks bestimmt. Für Aktivierungen werden hierzu typischerweise repräsentative Eingabedaten verwendet, mit denen die auftretenden Wertebereiche während einer Kalibrierung bestimmt werden. Dadurch kann verhindert werden, dass einzelne Ausreißer den Wertebereich unnötig vergrößern und somit die verfügbare Quantisierungsauflösung für den überwiegenden Teil der Werte verschlechtern [6].

2.3.2 Pruning

Pruning bezeichnet ein Verfahren zur Reduktion der Größe und Komplexität eines neuronalen Netzwerks, bei dem ausgewählte Parameter oder strukturelle Bestandteile des Netzwerks entfernt werden. Grundsätzlich wird zwischen unstrukturiertem und strukturiertem Pruning unterschieden [27].

Beim unstrukturierten Pruning werden einzelne Gewichte aus dem Netzwerk entfernt beziehungsweise auf null gesetzt. Häufig werden hierzu Verfahren verwendet, bei denen Gewichte anhand ihrer Größe, beziehungsweise ihres Betrags bewertet werden. Die Dimensionen und damit die grundlegende Struktur der Layer bleiben dabei unverändert. Eine tatsächliche Reduktion der physikalischen Modellgröße oder der Inferenzzeit ist daher nicht zwangsläufig gegeben, sondern hängt davon ab, ob die verwendete Speicherrepräsentation und Hardware die entstehende Sparsität ausnutzen können.

Beim strukturierten Pruning werden hingegen vollständige strukturelle Einheiten wie Filter, Kanäle oder Neuronen entfernt. Dadurch verändern sich die Dimensionen der betroffenen Layer, wodurch sowohl die Anzahl der Parameter als auch die erforderlichen Rechenoperationen reduziert werden können. Aufgrund der regulären

resultierenden Netzwerkstruktur kann strukturiertes Pruning leichter von konventioneller Hardware und Software ausgenutzt werden und damit zu einer tatsächlichen Reduktion des Speicher- und Rechenaufwands führen [27].

Das Pruning kann einmalig oder iterativ durchgeführt werden. Bei einem iterativen Verfahren werden zunächst ausgewählte Parameter oder Strukturen entfernt und anschließend die Modellgenauigkeit auf einem Validierungsdatensatz überprüft. Abhängig vom Ergebnis kann das Modell anschließend weiter gepruned und gegebenenfalls erneut trainiert beziehungsweise feinjustiert werden. Auf diese Weise kann ein Kompromiss zwischen Modellkomplexität und Modellgenauigkeit bestimmt werden.

Für das Pruning wird im Gegensatz zur Quantisierung keine einheitliche Verbesserung für alle Modelle erwartet. Der erreichbare Vorteil hängt wesentlich von der verwendeten Pruning-Methode, dem Grad der Reduktion und der Struktur des jeweiligen Netzwerks ab. Insbesondere bei unstrukturiertem Pruning muss dieses von der Hardware, beziehungsweise dem Compiler explizit unterstützt werden müsste [9].

Alle Modelle mit Ausnahme des ToF-CNNs werden zwar vermutlich in irgendeiner Form gepruned sein, allerdings ist das nicht nachweisbar, da ST keine Optimierungsverfahren veröffentlicht hat.

Für das ToF-CNN wird der zusätzliche Nutzen des Prunings als begrenzt erwartet. Das Modell besitzt bereits eine vergleichsweise geringe Anzahl an Parametern und benötigt nur einen kleinen Speicherbereich. Eine weitere Reduktion der Gewichte würde daher zwar den Speicherbedarf weiter verringern, der absolute Effekt wäre jedoch gering. Gleichzeitig kann das Entfernen von Gewichten in einem kleinen Netzwerk einen vergleichsweise großen Einfluss auf die Modellfunktion haben. Für das ToF-CNN wurde das Pruning ursprünglich explizit getestet. Die Ergebnisse sind der Vollständigheit halber in der nachfolgenden Tabelle 2.6 aufgeführt:

	Speicherverbrauch Weights	**Speicherverbrauch .tflite Datei**	**Genauigkeit**
Nicht Pruned	10 kB	15,65 kB	74,31
Pruned (nicht-strukturiert)	10 kB	14,85 kB	63,89

Tabelle 2.6: Vergleich zwischen Pruned und Nicht Pruned CNN

Da kein relevanter Nutzen aus dem Pruning hervorging, wurde dieses Optimierungsverfahren für das ToF-CNN nicht angewendet.

3 Methodik

3.1 Prüfparameter

Folgende Prüfparameter werden betrachtet:

- Modellgröße
- Inferenzzeit
- RAM Nutzung
- mAP / Top-1 Accuracy

Die Modellgröße bezieht sich hierbei auf die physikalische Größe der .tflite Datei und nicht auf die Menge der Parameter.

3.2 Testablauf der existierenden Modelle

Die Prüfung der Neuronalen Netze auf den MCU wurde mithilfe von „STM32Cube AI-Studio“ Version 1.3.0 oder der ST Developer Cloud unter [19] durchgeführt. Hiermit bietet ST eine Pipeline die das einfache Evaluieren ermöglicht. Die mAP wurden über einer Validierung mit den unten angegebenen Datensätzen ermittelt. Alle anderen Parameter stammen aus der ”Default”-Konfiguration der Software. Dadurch wurden Implementierungsfehler ausgeschlossen. Die Software übernimmt auch die Übertragung des .tflite Models in nutzbaren C-Code. STM32Cube AI-Studio und die Developer-Cloud wiesen während den Versuchen im Bereich Inferenzzeit eine Abweichung von max. 2% auf. Daher wurden diese Abweichungen vernachlässigt. In den restlichen Parametern waren die Abweichungen ebenfalls nur sehr gering und wurden daher ebenfalls vernachlässigt.

Die nicht optimierten Modelle sind zu finden unter:

- Yolov8n: Export über das Python Ultralytics Modul.
- Yolo26n: Export über das Python Ultralytics Modul.

- MNetv3_small: (siehe `Anhang: mobilenet_v3_small.onnx`)
- ToF_CNN: (siehe `Anhang: tof_cnn_model.tflite`)

Sie unterscheiden sich von den orginal-veröffentlichten Modellen [5, 8]. Der größte Unterschied ist die Anpassung der Eingangsgröße und der Ausgangsklassen. Yolo-Modelle sind regulär auf 80 Ausgangsklassen trainiert. MobileNetV3-Small sogar auf 1000. Da die Objekterkennung von ST aber eher eine Personenerkennung darstellt, wurden die Modelle dementsprechend angepasst um nur eine Ausgangsklasse "Person" zu besitzen.

Die optimierten Modelle kommen mit Ausnahme des ToF-CNN Models aus dem Modelzoo von ST unter [21].

Um genauer zu sein aus:

- Yolo8n: [23]
- Yolo26n: [22]
- MNetv3_small: [20]

Das optimierte ToF-CNN ist ebenfalls zu finden unter:

(`Anhang: tof_cnn_model_optimized.tflite`)

Die Datasets sind zu finden unter:

- Yolo8n, Yolo26n: [24]
- MNetv3_small: [2]
- ToF-CNN: (siehe `Anhang: tof_cnn_dataset.csv`)

Zunächst wurden alle Modelle ohne jegliche Optimierungen auf dem STM32N6570-DK getestet. Zwar ist die NPU nur auf INT8 Berechnungen ausgelegt, jedoch ermöglicht der in STMCube AI-Studio eingebaute Compiler auch eine Ausführung mit Float32 Werten. Hierbei ist zu beachten, dass dadurch alle Berechnungen über den Host, also die CPU, durchgeführt werden müssen. Damit dienen diese Ergebnisse als Basis für den Vergleich mit der NPU.

Danach wurden die optimierten Modelle mit den gleichen Randbedingungen auf dem Board getestet und anhand der vorher genannte Prüfparameter kategorisiert. Diese Ergebnisse sollen einen Überblick bieten, welchen Einfluss eine gute Optimierung auf die Anwendungsmöglichkeiten hat. Die optimierten Modelle stellen dabei keine optimalen Modelle dar und können ggf. noch weiter verbessert werden.

3.3 Experimente zur Anpassung an die NPU Struktur

ST beschreibt selbst, dass die NPU auf manche Operationen und Parameter besser ausgelegt ist als auf andere [18]. Um diese Aussagen zu überprüfen und um die Arbeit von Shiyuan Zhang [16] zu erweitern wurde erneut die klassischen 2 NN-Strukturen erforscht. Zunächst eine Conv2D - ReLU - Pool Schicht-Struktur. Diese ist eine typisch vorkommende Struktur in den meisten CNNs. Danach wurde diese Struktur ausgedehnt um die Grenzen einer HW-Epoche zu finden. Das in [16] analysierte Fully Connected Layer wurde ausgelassen, da es bereits zu Genüge dokumentiert wurde.

Die Schicht-Struktur hat aber meines Erachtens nicht alle Parameter zur Genüge ausgetestet. Daher diese Erweiterung.

Im Ersten Experiment sollen die Auswirkungen der Eingangs-Dimensionen bewertet werden. Dabei haben die 3 Schichten folgende Parameter:

- Conv2D: Eingangsdimensionen variieren, Filteranzahl 16, Kernel Size: 3x3, Stride 1
- ReLU: Standard ReLU Aktivierungsfunktion
- MaxPool2D: 2x2 Pooling, Stride 2

Die Eingangsdimensionen variieren von 32x32x1 bis 1024x1024x1. Im Ersten Schritt wurden die Größen jedes Mal verdoppelt. Da im Bereich zwischen 512x512 und 1024x1024 Unregelmäßigkeiten in der Inferenzzeit aufgetaucht sind, wurde hier nochmals die Auflösung erhöht und der Input in Schritten von +64 bis 1024 erhöht. Diese Auflösung reichte alleine aber auch noch nicht aus um eine sinnvolle Schlussfolgerung zu erreichen. Daher wurde im Bereich 480x480 bis 576x576 nochmals die Auflösung erhöht und die Eingangsgröße in +16 Schritten erhöht. Um ein endgültiges Ergebnis treffen zu können wurde auch noch ein einzelner Messpunkt bei 514x514 eingefügt.

Im zweiten Experiment sollten dann die Auswirkungen von der Kernelanzahl auf die Inferenz gemessen werden. Dabei wurde die Eingangsgröße auf 64x64x1 beschränkt und die Filteranzahl von 16 - 480 verändert. Hierbei wurden ebenfalls die Anzahl immer verdoppelt um einen möglichen Linearen Ansatz zu erkennen.

Im dritten Experiment sollte dann der Sprung in Inferenzzeit etwas genauer definiert werden. Da die Anzahl der Filter einen Einfluss auf die Speichergröße bewies, wurde danach am Grenzwert von 512 und 514 der Eingangsgröße die Filteranzahl variiert um den Spike der Inferenzzeit besser einschränken zu können.

Im vierten Experiment wurde dann die Auswirkung der Kernel Größen und die Veränderung der Stride/Schrittweite gemessen. Hierbei hatten die 3 Schichten folgende Parameter:

- Conv2D: Eingangsgröße 256x256, Filteranzahl 16, Stride 1-4, Kernel variiert zwischen 1x1 - 5x5
- ReLU: Standard ReLU Aktivierungsfunktion
- MaxPool2D: 2x2 Pooling, Stride 2

Und im letzten und fünften Experiment wird die Struktur hintereinander gestapelt um zu sehen ab welchen Punkt eine Epoche oder ein Commandstream an sein Limit kommt.

Hierbei werden die Parameter wieder abgeändert zu:

- Conv2D: Eingangsgröße 256x256, Filteranzahl 16, Stride 1, Kernel 3x3
- ReLU: Standard ReLU Aktivierungsfunktion
- MaxPool2D: 2x2 Pooling, Stride 2

Diese Architektur wird dann jeweils hintereinander gestapelt und die Inferenzzeit, Modellgröße, RAM-Anforderungen und Epochen Mapping überprüft.

4 Erwartungen der Experimente

Hier werden lediglich die Erwartungen an die Experimente erklärt, die eine erwartete Abhängigkeit aufweisen sollten. Zu Experiment 3 werden keine eigenen Erwartungen getroffen, da es sich um eine experimentelle Betrachtung des Verhaltens handelt. Es können dabei keine Erwartungen getroffen werden, weil das Verhalten den vorherigen Erwartungen widerspricht. Die möglichen Gründe für dieses Verhalten werden später (unter Abschnitt 6) genauer erläutert.

4.1 Experiment 1 - Abhängigkeit zur Eingangsgröße

Im ersten Experiment wird eine klare Abhängigkeit der Inferenzzeit von der Eingangsgröße vermutet. Da die Anzahl der, für die Faltung notwendigen Operationen, mit zunehmender räumlicher Auflösung ansteigt, wird zunächst auch eine steigende Inferenzzeit erwartet.

Für eine allgemeine Conv2D-Schicht lässt sich die Anzahl der Multiply-Accumulate-Operationen (MACs) durch

$$N_{\mathrm{MAC}} = H_{\mathrm{out}} \cdot W_{\mathrm{out}} \cdot K_H \cdot K_W \cdot C_{\mathrm{in}} \cdot C_{\mathrm{out}} \tag{4.1}$$

beschreiben.

Dabei bezeichnen H_{out} und W_{out} die räumlichen Dimensionen der Ausgabe, K_H und K_W die Kernelhöhe und Kernelbreite sowie C_{in} und C_{out} die Anzahl der Eingangs- und Ausgangskanäle.

Im ersten Experiment wird ein 3×3-Kernel mit einem Eingangskanal, 16 Filtern, einem Stride von 1 und einem “same” Padding verwendet. Das Padding wird durch

$$P = \left\lfloor \frac{K}{2} \right\rfloor \tag{4.2}$$

für ungerade Kernelgrößen berechnet. Mit $K_H = K_W = K$. Für den hier verwendeten 3×3-Kernel ergibt sich somit

$$P = \left\lfloor \frac{3}{2} \right\rfloor = 1 \tag{4.3}$$

Damit kann die Formel mit $H_{\text{in}} = H_{\text{out}} = H$ und $W_{\text{in}} = W_{\text{out}} = W$ vereinfacht werden. Die räumlichen Dimensionen der Conv2D-Ausgabe entsprechen damit denen der Eingabe. Somit ergibt sich

$$N_{\text{MAC}} = H \cdot W \cdot 3 \cdot 3 \cdot 1 \cdot 16 \tag{4.4}$$

und damit

$$\boxed{N_{\text{MAC}} = 144HW} \tag{4.5}$$

Für quadratische Eingaben mit einer Seitenlänge S gilt $H = W = S$. Daraus folgt:

$$\boxed{N_{\text{MAC}} = 144S^2} \tag{4.6}$$

Eine Verdopplung der Seitenlänge führt zu einer Vervierfachung der theoretisch erforderlichen MACs. Beispielsweise benötigt eine Eingabe mit 256×256 Pixeln

$$N_{\text{MAC}} = 144 \cdot 256^2 = 9\,437\,184 \tag{4.7}$$

MACs. Für eine Eingabe mit 512×512 Pixeln ergibt sich hingegen

$$N_{\text{MAC}} = 144 \cdot 512^2 = 37\,748\,736 \tag{4.8}$$

MACs. Damit gilt allgemein

$$N_{\text{MAC}} \propto S^2. \tag{4.9}$$

Neben der Anzahl der Rechenoperationen verändert sich durch eine größere Eingangsgröße auch der Speicherbedarf der Aktivierungen. Die Anzahl der Werte der Conv2D-Ausgabe beträgt

$$N_{\text{Activation}} = H \cdot W \cdot C_{\text{out}}. \tag{4.10}$$

Da in diesem Experiment $C_{\text{out}} = 16$ gilt, ergibt sich

$$\boxed{N_{\text{Activation}} = 16HW} \tag{4.11}$$

Bei einer INT8-Darstellung benötigt jeder Aktivierungswert ein Byte. Der Speicherbedarf einer einzelnen Conv2D-Ausgabe beträgt daher

$$\boxed{M_{\text{Activation}} = 16HW \text{ Byte}} \tag{4.12}$$

Für eine Eingabe von 256×256 ergibt sich somit

$$M_{\text{Activation}} = 16 \cdot 256^2 = 1\,048\,576 \text{ Byte} \approx 1 \text{ MiB}. \tag{4.13}$$

Für 512×512 ergibt sich entsprechend

$$M_{\text{Activation}} = 16 \cdot 512^2 = 4\,194\,304 \text{ Byte} \approx 4 \text{ MiB}. \tag{4.14}$$

Der Speicherbedarf der Aktivierungen wächst somit ebenfalls quadratisch mit der Seitenlänge:

$$M_{\text{Activation}} \propto S^2. \tag{4.15}$$

Im Gegensatz dazu bleibt die Anzahl der Gewichte der Conv2D-Schicht bei einer Veränderung der Eingangsgröße konstant. Sie berechnet sich zu

$$N_{\text{Weights}} = K_H \cdot K_W \cdot C_{\text{in}} \cdot C_{\text{out}}. \tag{4.16}$$

Für die verwendete 3×3-Faltung ergibt sich

$$N_{\text{Weights}} = 3 \cdot 3 \cdot 1 \cdot 16 = 144. \tag{4.17}$$

Unter Berücksichtigung eines Bias pro Ausgangskanal ergeben sich insgesamt

$$N_{\text{Parameters}} = 144 + 16 = 160 \tag{4.18}$$

Parameter.

Bei einer INT8-Darstellung entspricht dies also 160 Byte für Gewichte und Bias. Die Modelgröße der eigentlichen Schichten-Parameter ist somit unabhängig von der Eingangsgröße.

Für das erste Experiment ergeben sich damit drei unterschiedliche Abhängigkeiten:

$$\boxed{N_{\text{MAC}} \propto S^2} \tag{4.19}$$

$$\boxed{M_{\text{Activation}} \propto S^2} \tag{4.20}$$

$$\boxed{M_{\text{Model}} \approx \text{konstant}} \tag{4.21}$$

Es wird daher erwartet, dass die Inferenzzeit mit zunehmender Eingangsgröße ansteigt. Die Inferenzzeit muss jedoch nicht proportional zur reinen Anzahl der MACs steigen. Zusätzlich beeinflussen die Datenbewegung, die Wiederverwendung von Eingabedaten, der verfügbare Speicher sowie die Abbildung der Operationen auf die Hardware der NPU die tatsächliche Laufzeit.

Insbesondere bei großen Eingabedimensionen können Speicher- oder Mapping-Grenzen erreicht werden. In diesem Fall kann der Compiler die Verarbeitung in mehrere Teiloperationen beziehungsweise Hardware-Epochen aufteilen. Dadurch können sprunghafte Änderungen der Inferenzzeit auftreten, obwohl die theoretische Anzahl der MACs weiterhin kontinuierlich mit der Eingangsgröße ansteigt.

4.2 Experiment 2 - Abhängigkeit von der Anzahl der Filter

Da die Anzahl der Filter die Tiefe des Ausgangs festlegt, und laut Gleichung (4.4), die Anzahl der Rechenoperationen direkt abhängig ist mit der Filteranzahl, wird ein nahezu lineares Verhältnis in Inferenzzeit erwartet. Da der Speicher laut Gleichung (4.10) ebenfalls abhängig von der Anzahl der Filteranzahl ist, wird hier ebenfalls ein lineares Verhältnis erwartet. Dieses Verhalten ist vermutlich nicht über den gesamten Bereich konsistent. Das Speicherlimit des NPU-internen RAMs ist irgendwann erreicht und es muss ein anderes Compiler-Mapping eingesetzt werden. Dadurch können auch Inferenzsprünge auftreten. Experiment 3 liefert dazu einen genaueren Einblick.

4.3 Experiment 3 - Genauere Auflösung zu Exp. 2

Da Experiment 3 eine genauere Betrachtung des Verhaltens aus Abschnitt 4.2 darstellt, ist die Erwartung einen klaren Sprung oder Übergang zwischen dem verschiedenen Compiler-Mapping erkennen zu können. Dort wird ein großer Inferenzsprung erwartet.

4.4 Experiment 4 - Unterschiede in Kernel Dimensionen

In [18] wird eine klare Empfehlung für Kernel-Größen und Stride-Größen gestellt. Als kurze Zusammenfassung, siehe Tabelle 4.1

Parameter	**Empfehlung**
Kernel (vertikal)	Maximale Größe: 3. Danach werden die Kernel in kleinere Kernel aufgeteilt.
Kernel (horizontal)	Maximale Größe: 6 bei Stride 1 und max. 12 bei Stride 2. Beste Größen: 3, 6, 12.
Stride (vertikal)	Vermeiden von Strides, die nicht 1, 2, 4 sind → Ineffizient.
Stride (horizontal)	Vermeiden von Strides, die nicht 1, 2, 4 sind → Ineffizient.
Sonstiges	Kernelgrößen mit 1 (vertikal oder horizontal) laufen besonders effizient. Stride sollte nicht größer als Kernelgröße sein → Datenverlust.

Tabelle 4.1: Übersicht über die Begrenzungen der NPU-Conv2D-Parameter [18]

Daraus lässt sich vermuten, dass die Ergebnisse diesen Empfehlungen folgen sollten.

4.5 Experiment 5 - Abhängigkeit zur Anzahl der Schichten

Es wird zunächst eine Abhängigkeit der Inferenzzeit von der Anzahl der hintereinander ausgeführten Schichten vermutet, da mit jedem zusätzlichen Conv2D-Schicht weitere Rechenoperationen ausgeführt werden müssen. Aufgrund der am Ende verwendeten MaxPool2D-Schicht mit einer Poolgröße von 2×2 und einem Stride von 2 halbiert sich die Ausgangsgröße der letzten Schicht.

Für die erste Conv2D-Schicht mit einer Eingangsgröße von $256 \times 256 \times 1$, einem 3×3-Kernel und 16 Filtern ergibt sich entsprechend der in (4.4) hergeleiteten Beziehung:

$$N_{\text{MAC,input}} = 256^2 \cdot 3^2 \cdot 1 \cdot 16 \tag{4.22}$$

Neue Ebenen werden direkt hinter der ersten Conv2D-Schicht eingefügt. Ein Padding von 1 verhindert weiterhin ein Schrumpfen der Ausgangsgrößen.

Da sich mit jedem zusätzlichen Block ein weiterer Schicht mit dem Rechenaufwand N_{MAC} anreiht, wächst die Anzahl der Operationen nicht geometrisch, sondern linear mit der Anzahl der Schichten L. Für die Gesamtzahl der MACs bei L Schichten gilt somit:

$$N_{\mathrm{MAC,total}} = N_{\mathrm{MAC,input}} + L \cdot N_{\mathrm{MAC}} \tag{4.23}$$

Die Gesamtzahl der Operationen wächst somit ab der zweiten Schicht direkt proportional bzw. linear mit jeder zusätzlichen Schicht an. Der zusätzliche Rechenaufwand bleibt danach pro hinzugefügter Schicht konstant.

Damit wird grundsätzlich eine kontinuierliche und gleichmäßige Zunahme der Inferenzzeit mit zunehmender Anzahl der Schicht erwartet. Der Zusammenhang mit der Netzwerktiefe ist in diesem Fall linear.

Zusätzlich zur reinen Rechenlast verändert sich mit steigender Anzahl der Schichten auch die Menge und Struktur der im Netzwerk verwendeten Aktivierungen. Es steigt mit jeder zusätzlichen Schicht die Anzahl, der im gesamten Netzwerk enthaltenen Aktivierungstensoren sowie, der für deren Verwaltung und Verarbeitung notwendige Speicherbedarf. Für die NPU ist dabei nicht nur die Größe einzelner Tensoren relevant, sondern auch, wie diese durch den Compiler auf die verfügbaren Speicherbereiche und Hardware-Epochen abgebildet werden können.

Es wird daher vermutet, dass bis zu einer bestimmten Netzwerktiefe eine weitgehend kontinuierliche Zunahme der Inferenzzeit beobachtet werden kann. Wird jedoch eine Grenze des für das Mapping verfügbaren Speichers, des Commandstreams oder der Hardware-Epochen erreicht, kann sich die Abbildung des Netzwerks auf die NPU verändern. In diesem Fall kann der Compiler gezwungen sein, weitere Teiloperationen beziehungsweise zusätzliche Epochen zu erzeugen oder Daten zwischen Speicherbereichen umzuordnen. Dies kann zu einem diskreten Sprung der Inferenzzeit führen, obwohl sich die theoretische Anzahl der MACs nur geringfügig verändert.

Experiment 5 soll somit insbesondere untersuchen, ob mit zunehmender Anzahl gestapelter Conv2D-ReLU-MaxPool-Blöcke zunächst die erwartete Zunahme der Inferenzzeit aufgrund der zusätzlichen Rechenoperationen auftritt und ab einer bestimmten Netzwerktiefe zusätzlich eine Grenze des NPU-Mappings erreicht wird. Hierzu werden neben der Inferenzzeit auch die resultierende Modelgröße, die RAM-Anforderungen sowie die Anzahl und Struktur der erzeugten Hardware-Epochen betrachtet.

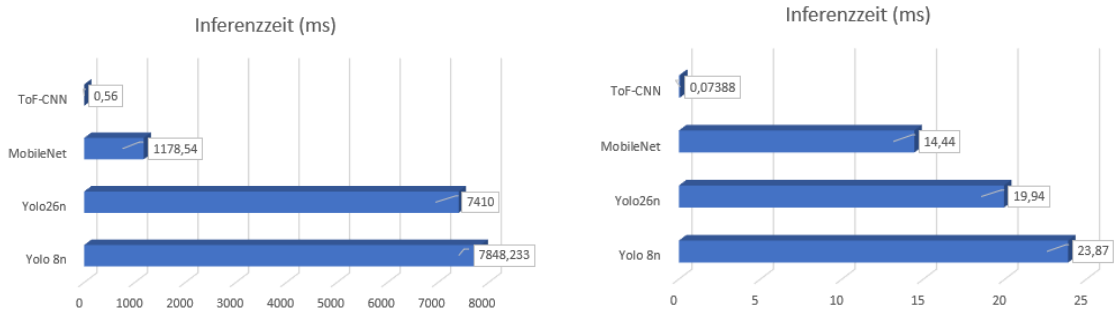
5 Ergebnisse

Die reinen Ergebnisse werden als separate CSV-Datei an diese Arbeit angehängt (siehe `Anhang_A.csv`). Alle Ergebnisse wurden aus dem STM32Cube AI-Studio oder der ST Developer Cloud (zu finden unter [19]) bezogen.

5.1 Vergleich der Modelle

Die in den einzelnen Unterpunkten getroffenen Aussagen, sind im Regelfall nur Vermutungen. Da das von ST optimierte Modell als Vergleich gewählt wurde, ist nicht genau dokumentiert welche Optimierungen getroffen wurden. Daher ist eine genaue Begründung ohne weitere Versuche oder tiefgründige Analyse der Ebenen unmöglich.

5.1.1 Inferenzzeit



(a) Inferenzzeit der Originalmodelle (b) Inferenzzeit nach der Anpassung

Abbildung 5.1: Vergleich der Inferenzzeiten vor und nach der Anpassung an die NPU

Um die Abbildungen 5.1a und 5.1b in Zusammenhang zu setzen, zeigt die Tabelle 5.1 den Faktor um den die Inferenzzeit gesunken ist.

Hierbei fällt auf, dass Yolo26 die größte, anteilige Beschleunigung aufweist. Das hängt mit der Architektur des Modells zusammen. Laut Ultralytics [8, 25] weißt es eine

Model	Faktor Inferenzsenkung
Yolov8n	328,79
Yolo26n	371,61
MNetv3	206,44
ToF-CNN	7,58

Tabelle 5.1: Senkungsfaktoren der Modelle nach der Optimierung.

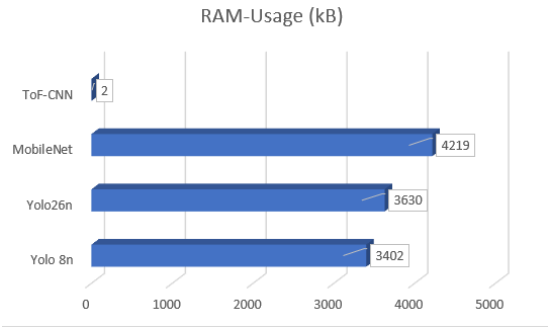
schnellere Inferenz aufgrund eines kleineren ”Detection Heads” und ein aktualisiertes Trainingprinzip auf. Dabei ist anzumerken, dass das reguläre Yolo26n Modell bisher nicht ohne weiteres im STM32Cube AI-Studio importierbar ist. Das liegt an 2 nicht unterstützten Softmax Schichten. Diese werden allerdings mithilfe der Parameter: nms=False, simplyfy=true in Ultralytics ”model.export Funktion”’ entfernt/ersetzt. Desweiteren beinhaltet die Optimierung auch ein Neutraining auf lediglich eine Ausgangsklasse. In diesem Fall ”Person”. Die anderen Optimierungsschritte sind leider nicht mehr so ersichtlich, da ST diese nicht veröffentlicht hat. Diese könnten in einer zukünftigen Ausarbeitung über Netron genauer analysiert werden. Es kann aber mit Sicherheit gesagt werden, dass das Modell auf Int8 quantisiert wurde.

Die Beschleunigung des ToF-CNNs dagegen kommt lediglich von der Quantisierung. Die Architektur wurde bereits mit den Ergebnissen aus Abschnitt 5.2 im Hinterkopf gewählt. Daher war eine weitere Optimierung nicht notwendig. Trotzdem wäre eine weitere Spezialisierung möglich indem die Erkenntnisse aus [16] mit einbezogen werden und die Filteranzahl auf 4 reduziert wird, um eine Parallele Verarbeitung der NPU zu ermöglichen. Dies wurde hier jedoch nicht angewendet, da eine Inferenzzeit von ca. 0,07 ms und damit max. 13,534 FPS vollkommend ausreichend sind. Dabei wird die CPU lediglich für 0,026 ms genutzt. Die restliche Inferenzzeit (ca. 65% der Gesamtzeit) wird selbstständig von der NPU ausgeführt. Das Modell ist vollständig echtzeitfähig.

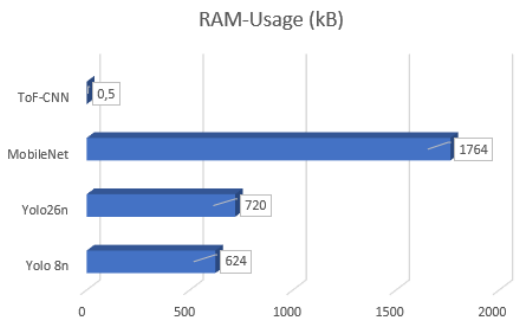
M-Net’s Beschleunigung ist prozentual geringer mit einem Faktor von 206,44. Aber auch hier ist eine Inferenzzeit von 14,44 ms gleichbedeutend mit 69,25 FPS. Damit ist eine Echtzeitfähigkeit ebenfalls durchaus realistisch.

Selbst das nach der Optimierung langsamste Modell, das Yolov8n erreicht noch eine umgerechnete Inferenzzeit von ca. 42 FPS. Damit kann es durchaus für Echtzeitanwendungen in Betracht gezogen werden. Auch hier wäre jedoch eine weitere Beschleunigung über eine Reduktion der Eingangsdimensionen denkbar.

5.1.2 RAM-Usage



(a) RAM-Nutzung der Originalmodelle



(b) RAM-Nutzung nach der Anpassung

Abbildung 5.2: Vergleich der RAM-Nutzung vor und nach der Anpassung an die NPU

Von Abbildung 5.2a zu 5.2b ist bei Yolo26n eine Reduktion vom Faktor ≈ 5 und bei Yolo8n vom Faktor $\approx 5,5$ sichtbar. Das ist höher als die, mit dem Faktor 4 vermutete Reduktion, die durch reine Quantisierung vorkommen würde. Dies lässt sich theoretisch durch strukturiertes Pruning erklären. Die Ausgangsklasse ist bereits bei dem getesteten Orginalmodell auf 1 begrenzt und hat daher keinen Einfluss.

Das ToF-CNN zeigt genau das vermutete Verhalten von einer 4x kleineren RAM-Nutzung aufgrund der Quantisierung.

Eine Ausnahme bildet hier das MobileNetv3. Dort ist eine Reduktion um den Faktor 2,4 zu beobachten. Das ist weniger als eine Quantisierung hervorrufen würde. Eine mögliche Erklärung wäre eine, von ST durchgeführte, Änderung von beispielsweise:

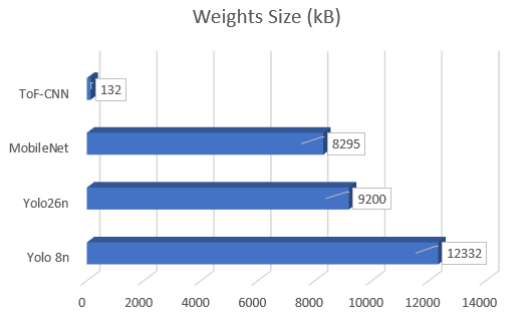
- Architektur
- Anzahl der Layer
- Kanalanzahl
- Klassenzahl

Andere Gründe für die inkonsistente Reduktion können z.B. sein:

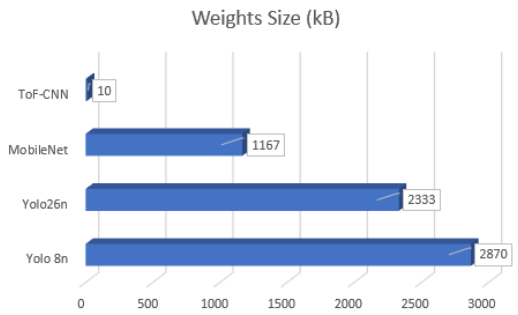
- Aktivierungsbuffer
- Speicheraufteilung
- zusätzliche Metadaten
- veränderte Netzwerkarchitektur
- Pruning
- Kompression

Auch hier gilt: Die genauen Hintergründe für die Reduktionen können nur mithilfe einer tiefreichenden Analyse beider Modelle und deren Schichten erklärt werden.

5.1.3 Gewichtsgröße



(a) Weight-Größen der Originalmodelle



(b) Weight-Größen nach der Anpassung

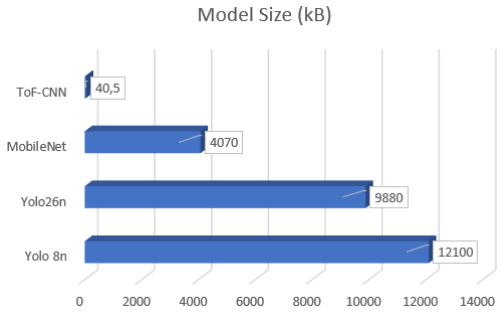
Abbildung 5.3: Vergleich der Weight-Größen vor und nach der Anpassung an die NPU

Auch von Abbildung 5.3a zu Abbildung 5.3b ist eine klare Reduktion sichtbar.

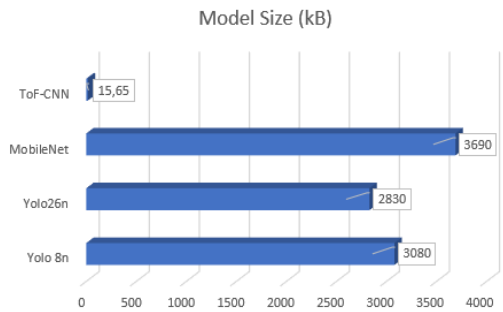
- ToF: Faktor 13,2
- MNetv3: Faktor $\approx 7,1$
- Yolo26n: Faktor $\approx 3,9$
- Yolo8n: Faktor $\approx 4,3$

Die Faktoren des ToF-CNNs und des MNets unterscheiden sich stark von denen der YOLO Modelle. Ihre Reduktion ist nicht wie erwartet um den Faktor ≈ 4 was bei reiner Quantisierung auftreten würde. Der Hintergrund könnte mit der NPU-spezifischen Gewichtsdarstellung zusammenhängen. Wie bereits in Tabelle 2.1 beschrieben, besitzt die NPU eine eigene Möglichkeit Kernel-Gewichte zur Laufzeit zu dekomprimieren. Der erhöhte Faktor kann also durch eine, durch den Compiler durchgeführte, Kompression der Gewichte kommen.

5.1.4 Modellgröße



(a) Modellgröße der Originalmodelle



(b) Modellgröße nach der Anpassung

Abbildung 5.4: Vergleich der Modellgrößen vor und nach der Anpassung an die NPU

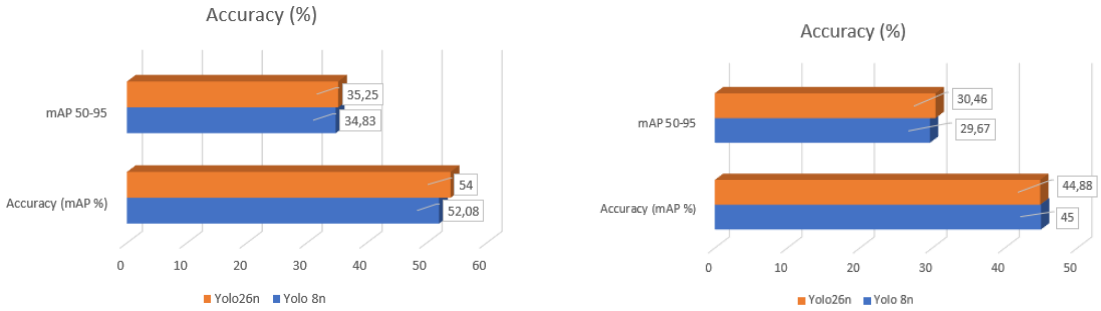
Hier ist ebenfalls eine klare Reduktion in der Modellgröße erkennbar. Auch hier bildet MNetv3 eine klare Außnahme. Die Modellgröße verringert sich lediglich um $\approx 9,3\%$.

- ToF: Faktor $\approx 2,6$
- MNetv3: Faktor $\approx 1,1$
- Yolo26n: Faktor $\approx 3,5$
- Yolo8n: Faktor $\approx 3,9$

Auch hier gibt es erneut eine Abweichung vom vermuteten Faktor 4. Hier können erneut Vermutungen aufgestellt werden. Die .tflite Dateien können einen Teil an Metadaten aufweisen, sodass die tatsächliche Repräsentation der Gewichte prozentual kleiner ausfällt. Auch können konstante Tensoren auftreten, die nicht als Gewichte betrachtet werden und daher nicht von der Quantisierung betroffen werden. Eine genaue Aussage, woher diese Faktoren kommen, ist aber nicht möglich.

5.1.5 Accuracy

Hier muss zwischen 2 verschiedenen Arten der Accuracy unterschieden werden. Zunächst die mAP und die mAP50-95 der Yolo-Modelle. Da es sich hier um Detektionsmodelle handelt, wird dabei die Genauigkeit der Bounding Boxen mit einbezogen.



(a) Accuracy vor der Optimierung (b) Accuracy nach der Optimierung

Abbildung 5.5: Vergleich der Accuracy der YOLO-Modelle vor und nach der Optimierung

In beiden Fällen ist der erwartete Genauigkeitsverlust zu sehen. Die mAP sinkt in beiden fällen um knapp 10 % wobei der Verlust bei Yolov8 3% weniger beträgt. In der mAP50-95 verlieren beide Modelle annähernd gleich viel. 4,79 % bei Yolo26n und 5,19% bei Yolov8. Welche Gründe das haben kann, ist nicht genau bekannt.

Im Vergleich zu den Detektionsmodellen dazu wird bei den Klassifikationsmodellen lediglich die Klasse mit dem höchsten Prediction Wert ausgewertet. Im Folgendenden bezeichnet als TTop-1 Accuracy"[14].

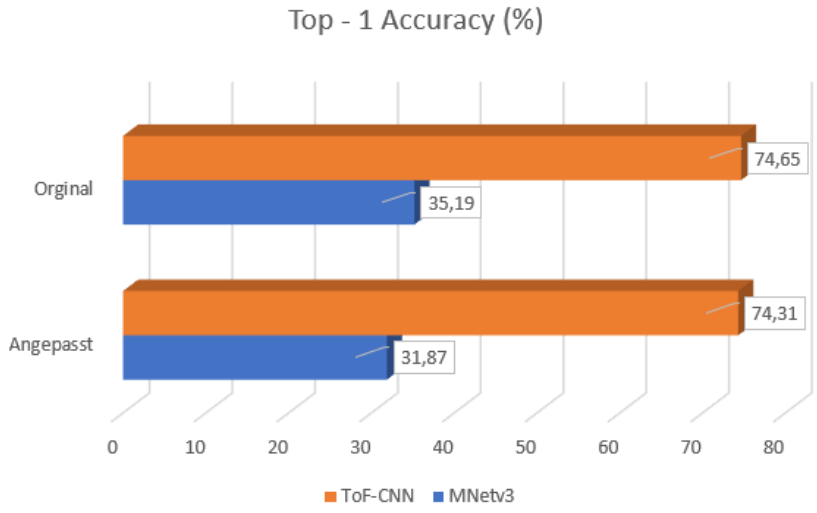


Abbildung 5.6: Accuracy Vergleich der Klassifizierungs Modelle

Auch hier tritt ein geringer Genauigkeitsverlust auf. Im ToF-CNN ist der Verlust wesentlich geringer mit 0,34 % als die 3,32 % des MNetv3. Das kann an einem längeren Training nach der Quantisierung liegen oder an einer weiteren Anpassung an die Netzstruktur nach/vor der Quantisierung. Im Falle des ToF-CNNs wird das Modell nach der Quantisierung erneut am gesamten Datenset nachtrainiert um eine

bessere Genauigkeit zu gewährleisten. Dieses Training könnte beim MNetv3 eventuell kleiner oder komplett ausgefallen sein. Auch können die Gewichte unterschiedlich stark von der Quantisierung betroffen sein. Wenn der Wertebereich der Gewichte kleiner ist, kann bei der Quantisierung eine höhere Auflösung und damit ein geringerer Genauigkeitsverlust auftreten.

5.2 Erweiterte Anpassung aus NPU-Struktur

Alle der hier durchgeführten Experimente sind nur mit Annahmen begründbar. Die genaue Zusammensetzung der NPU ist nicht veröffentlicht und damit können lediglich Vermutungen durch die von ST bereitgestellten Informationen getroffen werden.

5.2.1 Experiment 1 - Abh. zur Eingangsgröße

Eingangsgröße	**Inferenzzeit (ms)**
64x64	2,144
128x128	0,293
256x256	1,143
512x512	4,535
1024x1024	1700,671

Tabelle 5.2: Inferenzzeit nach Eingangsgröße

Eine interessante Beobachtung in Tabelle 5.2 ist vorallem, dass die Eingangsgrößen von 64x64 langsamer verarbeitet wird als 128x128 oder sogar 256x256. Dieses Ergebnis zeigt, dass die Inferenzzeit der Neural-ART-NPU nicht ausschließlich durch die theoretische Anzahl der Rechenoperationen bestimmt wird. Insbesondere im Bereich kleiner Eingabedimensionen treten Abweichungen von der erwarteten Skalierung auf, die auf Mapping-, Speicher- oder Setup-Effekte zurückzuführen sein können.

Außerdem wurde ein großer Sprung zwischen den Eingangsgrößen 512 und 1024 in der Inferenzzeit sichtbar. Dieser wurde in den nächsten 2 Tabellen näher betrachtet.

Bei einer geringeren Steigerung von je 64 Eingangsgrößen mehr pro Dimension ergab sich folgendes:

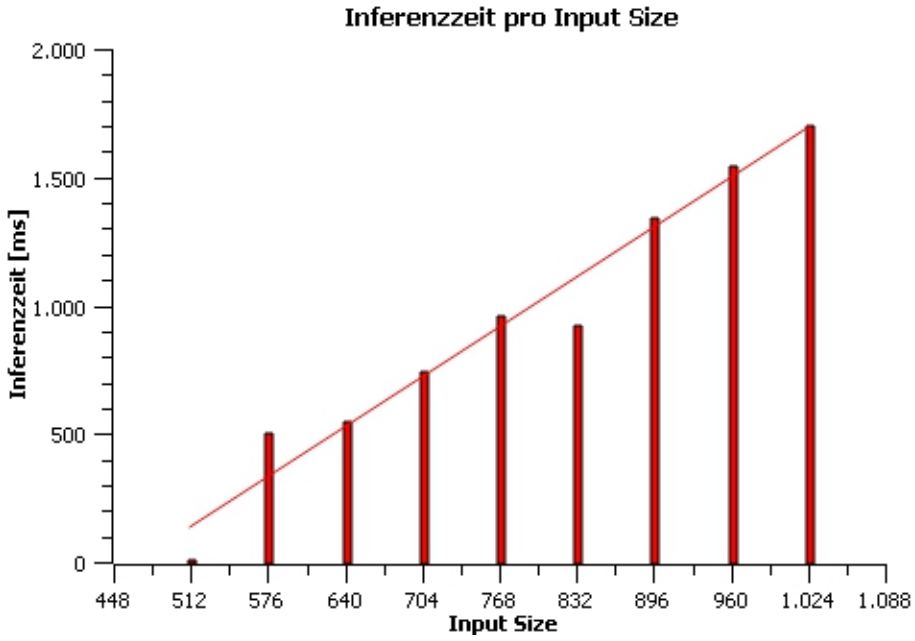


Abbildung 5.7: Inferenzzeit bei Steigender Eingangsgröße

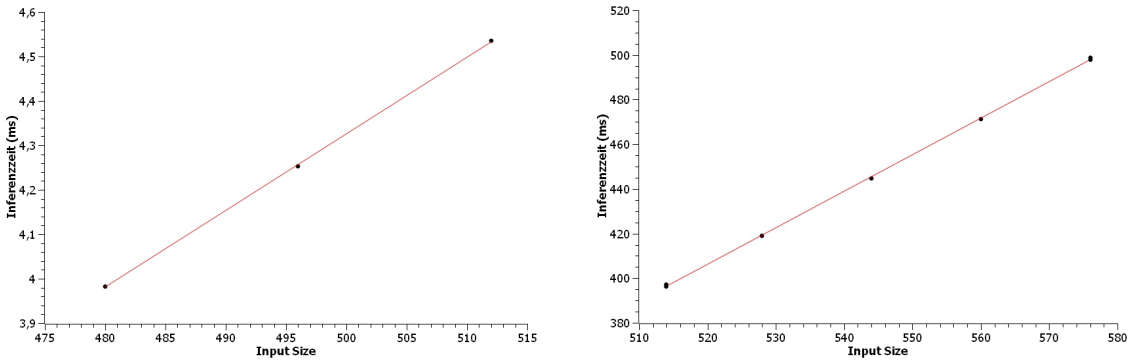
Hier ist ein stetiger Anstieg (siehe lineare Näherung) der Inferenzzeit zu sehen. Allerdings erklärt das weiterhin noch nicht den Ansprung zwischen 512 und 576. Daher wurde eine weitere Testreihe begonnen.

Eingangsgröße	**Inferenzzeit (ms)**
480x480	3,983
496x496	4,253
512x512	4,535
514x514	397,0
528x528	419,0
544x544	444,7
560x560	471,3
576x576	498,6

Tabelle 5.3: Inferenzzeit im Randbereich

Hierbei ist recht gut sichtbar, dass die Steigung vor 512 und nach 514 jeweils wieder annähernd Linear ansteigt. Die Inferenzzeit steigt dazwischen aber auf das etwa 87,5-Fache.

Durch Inspektion der RAM-Nutzung in AI-Studio kann aber auch der Grund vermutet werden. In diesem Bereich wird die RAM-Nutzung nicht mehr im NPU internen RAM durchgeführt, sondern im externen XSPI-RAM. Dadurch verlangsamt sich der Datentransfer und somit auch die Inferenz.



(a) Vor dem Sprung (b) Nach dem Sprung

Abbildung 5.8: Linearität vor und nach dem Sprung in Inferenzzeit

5.2.2 Experiment 2 - Abh. zur Anzahl der Filter

In Experiment 2 wurde der Einfluss der Filteranzahl auf das Modell überprüft.

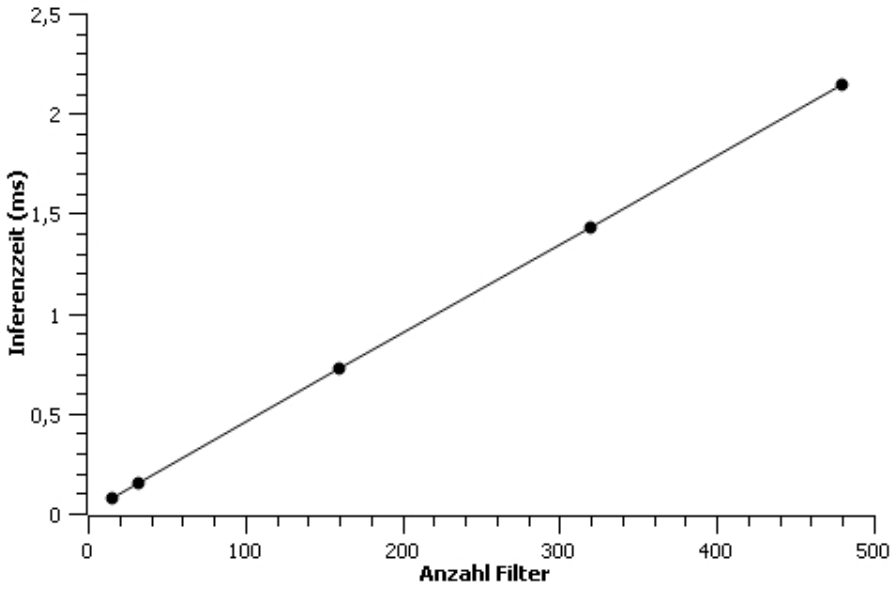


Abbildung 5.9: Inferenzzeit bei steigender Filteranzahl

In Abbildung 5.9 ist eine klare Linearität zu erkennen. Wie in Abbildung 5.10 zu sehen ist, bleibt die Aufteilung der Laufzeit zwischen NPU und CPU immer recht konstant bei 90/10.

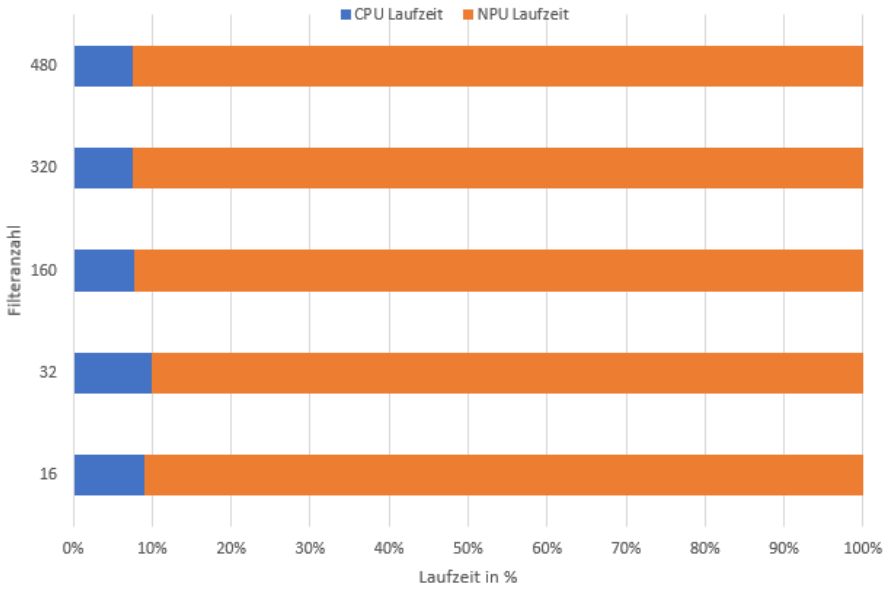


Abbildung 5.10: Inferenzzeit anteilig NPU/CPU

Die RAM-Nutzung durch die Aktivierungen sieht wie folgt aus:

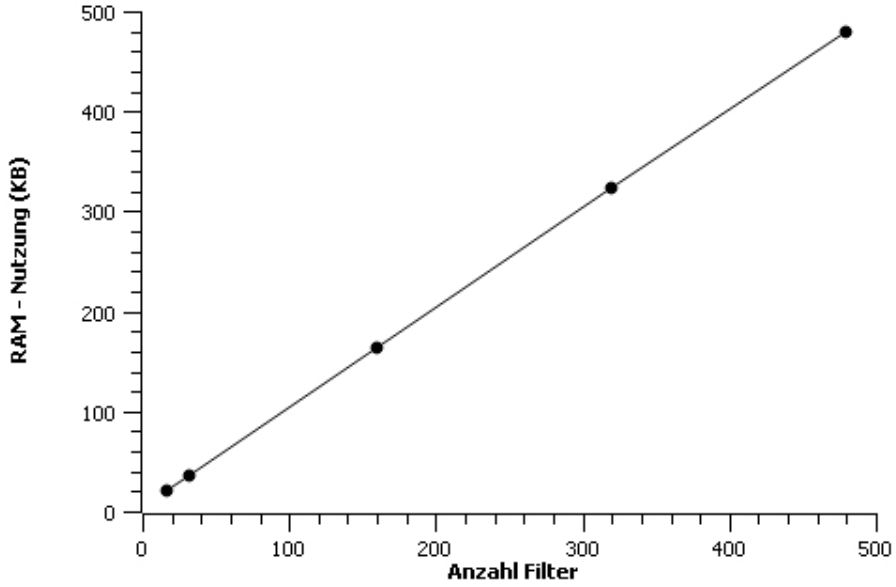


Abbildung 5.11: RAM-Nutzung

Auch in Abbildung 5.11 ist wieder ein linearer Verlauf gemäß der Filteranzahl zu sehen. Siehe Lineare Annäherung der Abbildung. Gleiches ist auch für die Modellgröße in Abb. 5.12 zu erkennen.

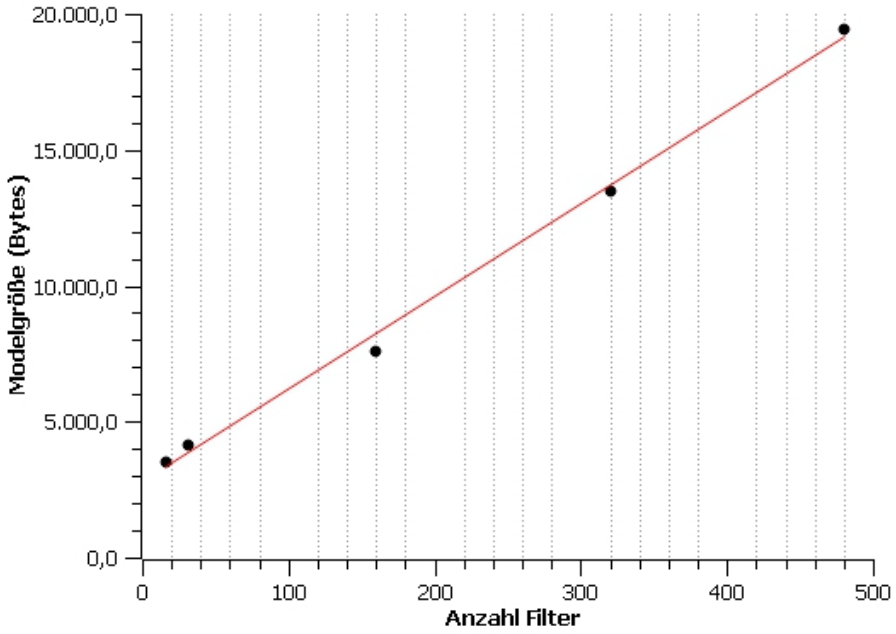


Abbildung 5.12: Modell-Größe

Der Compiler mappt hierbei weiterhin die gesamte Netzstruktur auf eine HW-Epoche.

5.2.3 Experiment 3 - genauere Auflösung von Exp. 2

Experiment 3 versucht einen genaueren Blick auf das Verhältnis von Filteranzahl an dem aus Abschnitt 5.2.1 gefundenen Sprung zu bieten.

Zunächst wurde an der Untergrenze der Eingangsgröße 512 folgendes beobachtet:

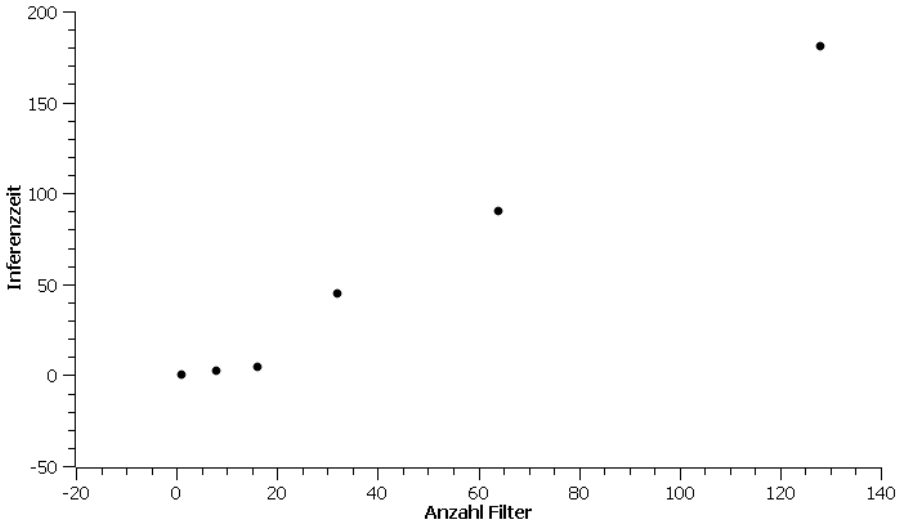


Abbildung 5.13: Inferenz am Grenzfall - 512

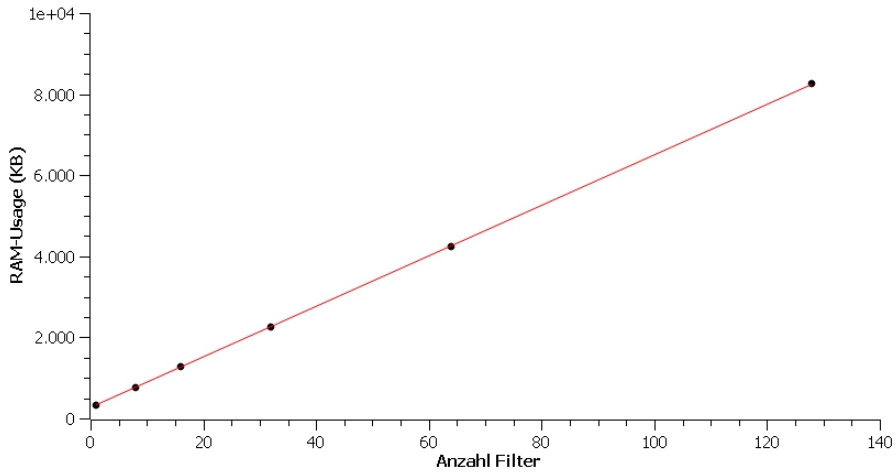


Abbildung 5.14: RAM Nutzung am Grenzfall - 512

In Abb. 5.13 konnte erneut ein Sprung nach 16 Filtern beobachtet werden. Sowohl die Regionen Vor als auch Nach dem Sprung scheinen wieder linear zu sein.

Die RAM-Nutzung in Abb. 5.14 weißt jedoch keinen Sprung auf, sondern kann wieder als grob linear angenommen werden.

5.2.4 Experiment 4 - Abh. zur Filter Dimension

In Experiment 4 wurden die Auswirkungen von Filtergröße und Stride ausgewertet.

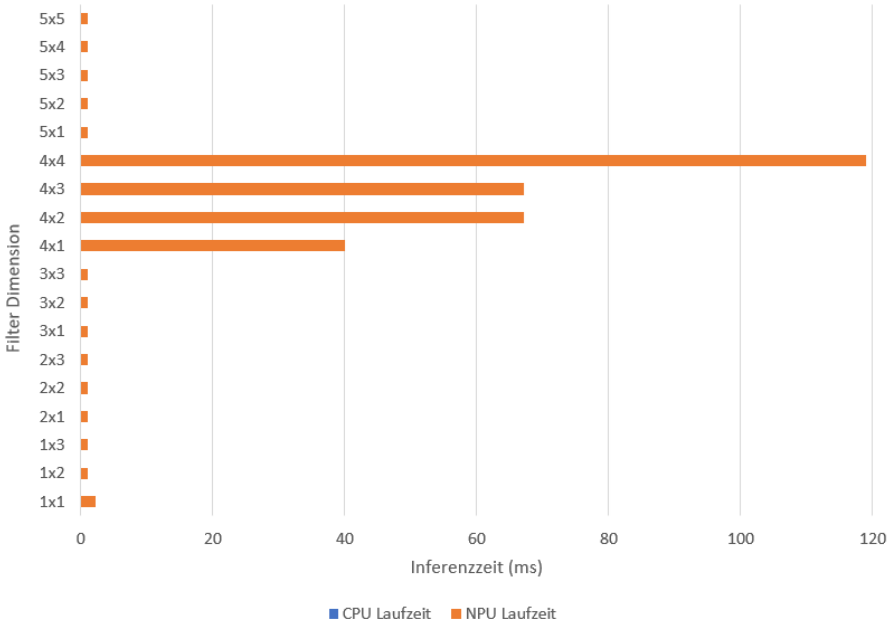


Abbildung 5.15: Inferenz mit Stride 1

Auffällig hier ist der starke Ansprung an Inferenz bei den Filterbreiten 4 dieser ist zumindest nicht von ST beschrieben. Bei genauerer Betrachtung über AI-Studio lässt sich aber wieder eine Theorie aufstellen. Die RAM-Aufteilung zeigt an, dass 4 MB vom XSPI RAM genutzt wurden. Die NPU-RAMs werden dabei größtenteils nicht benutzt. Warum dieses Verhalten auftritt ist bisher noch unklar. Der Zugriff auf den XSPI-RAM legt, wie bereits bei Experiment 1 vermutet, einen starker Anstieg an Inferenzzeit nahe. Der Grund dahinter ist vermutlich die NPU die auf einen externen Speicher zugreifen muss und der Datentransfer dort stark beschränkt ist. Ein kausaler Zusammenhang kann anhand der Messungen jedoch nicht vollständig bewiesen werden. Die weiteren Stride-Variationen weisen diesen Sprung nicht auf. Was zusätzlich auffällt, ist das der Compiler meistens ab der Filterbreite 4 eine zweite HW-Epoche vorsieht. Diese kümmert sich primär um die Zusammenführung der vorher getrennten Faltungsrechnung und dem Pooling. Die Faltungsrechnung wird ebenfalls bei einer Filterbreite von 5 aufgeteilt, aber hier ist die Berechnung innerhalb einer HW-Epoche möglich. Der Grund zu diesem Verhalten ist ebenfalls noch unklar.

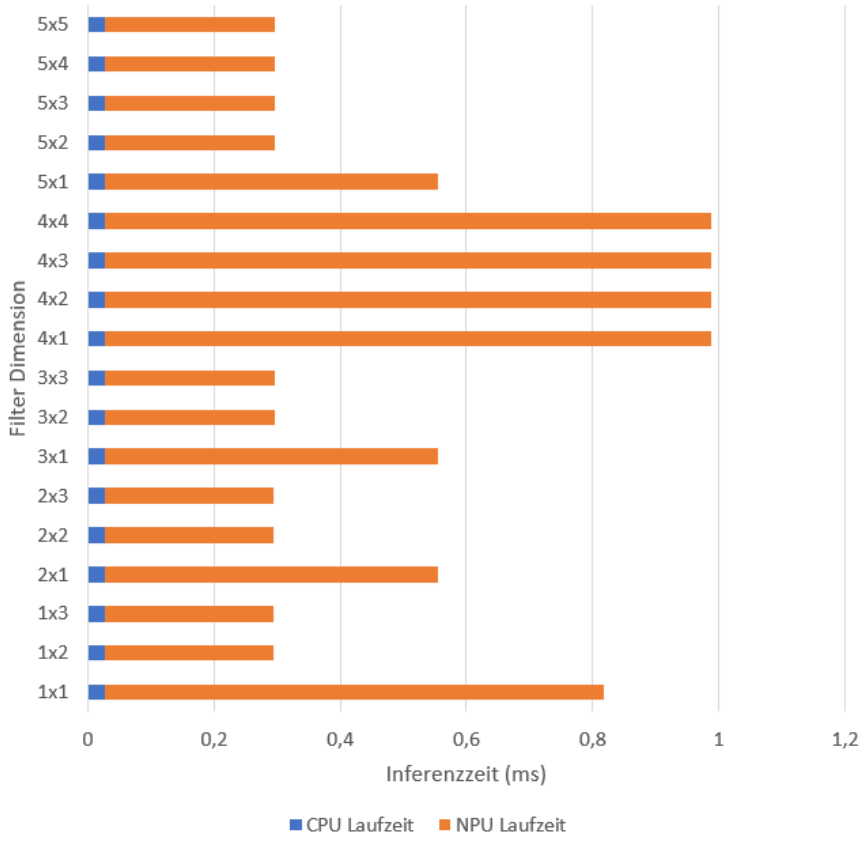


Abbildung 5.16: Inferenz mit Stride 2

Stride 2 zeigt wiederrum ein größtenteils erwartetes Verhalten. Es gibt erneut einen Anstieg der Inferenzzeit um die Kernelbreite 4. Ein erneuter Blick in AI-Studio zeigt auch hier wieder eine geteilte Filterberechnung und eine 2. HW-Epoche. Allerdings wird hier nicht der XSPI-RAM zugewiesen und daher ist der Sprung nicht so ausgeprägt. Bei der Filterbreite 5 wird die Berechnung ebenfalls wieder getrennt, aber erneut in eine HW-Epoche ausgeführt. Da ansonsten alle berechnungen innerhalb einer HW-Epoche durchgeführt werden können, gibt es nahezu keine Abweichungen in der Inferenzzeit außerhalb der eindimensionalen Filter. Die von ST beschriebene hohe Effizienz eindimensionaler Kernel konnte im vorliegenden Versuchsaufbau nicht beobachtet werden. Dies ist jedoch nicht zwingend als Widerspruch zur Herstellerdokumentation zu interpretieren, da die dort genannten optimalen Bedingungen, insbesondere bezüglich der Anzahl der Eingangs- und Ausgangskanäle, im Experiment nicht erfüllt wurden.

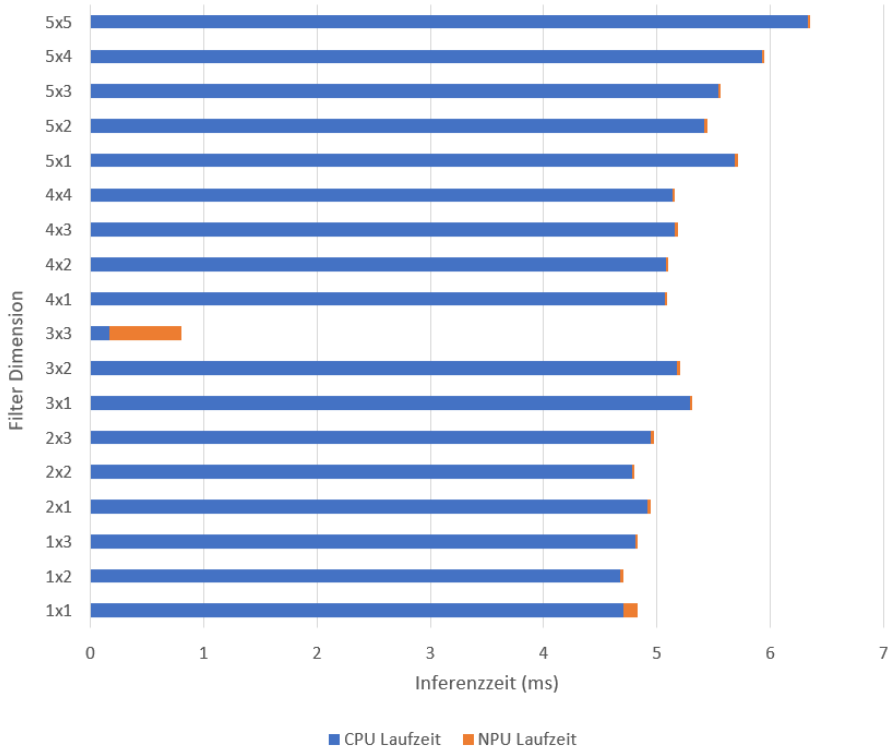


Abbildung 5.17: Inferenz mit Stride 3

Hier macht die ST-Empfehlung aus [18] durchaus Sinn. Eine Stride von 3 wird von der NPU nicht nativ unterstützt. Daher werden fast alle Operationen auf dem Host bzw. der CPU durchgeführt. Dadurch steigt die Inferenzzeit auch stark an. Aber auch hier stellen die Filterhöhe von 1 immer die höchste Inferenzzeit dar. Als einzige Ausnahme ist hier die Filtergröße 3x3. Hier wird die Berechnung auf eine SW-HW-Epoche gemappt. Das sorgt für wesentlich schnellere Berechnung mit weniger CPU-Nutzung. Der Hintergrund ist auch hier nicht bekannt.

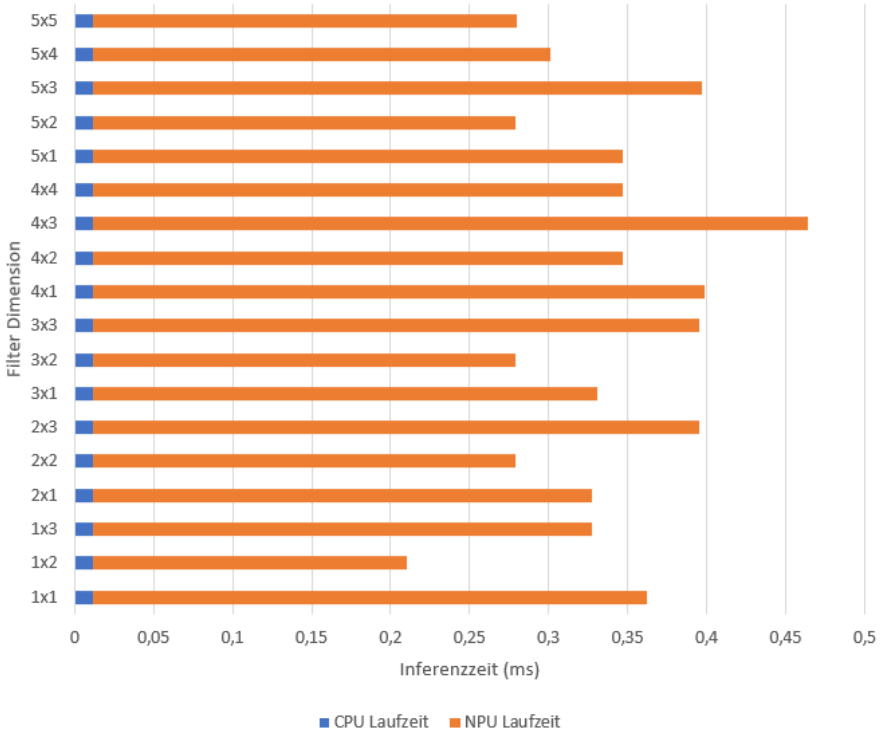


Abbildung 5.18: Inferenz mit Stride 4

In Stride 4 wird wieder alles wie bei Stride 1/2 gemappt. Die Schwankungen sind zwar deutlicher sichtbar, allerdings ist das lediglich der Nebeneffekt einer schnelleren Inferenz.

Allgemein lässt sich sagen, dass die Inferenzzeit mit höherer Schrittweite abnimmt. Ein Stride von 3 ist nicht empfohlen, da dieser nicht auf der NPU abgebildet werden kann. Desweiteren sollte bei Stride 1 Filter der Breite 4 grundsätzlich vermieden werden. Filter mit den Höhe 1 sind i.d.R. etwas langsamer als Filter mit Höhe 2. Der genaue Grund ist allerdings auch hier nicht bekannt. Die CPU ist während des Experimentes immer gleich lange an der Inferenz beteiligt.

5.2.5 Experiment 5 - Abh. zur Anzahl der Schichten

Zunächst ist hier zu beachten, dass die CPU-Laufzeit nahezu konstant bleibt und mit 0,088 ms so gering ist, dass sie vernachlässigt werden kann.

Anschließend wird die Inferenzzeit genauer betrachtet:

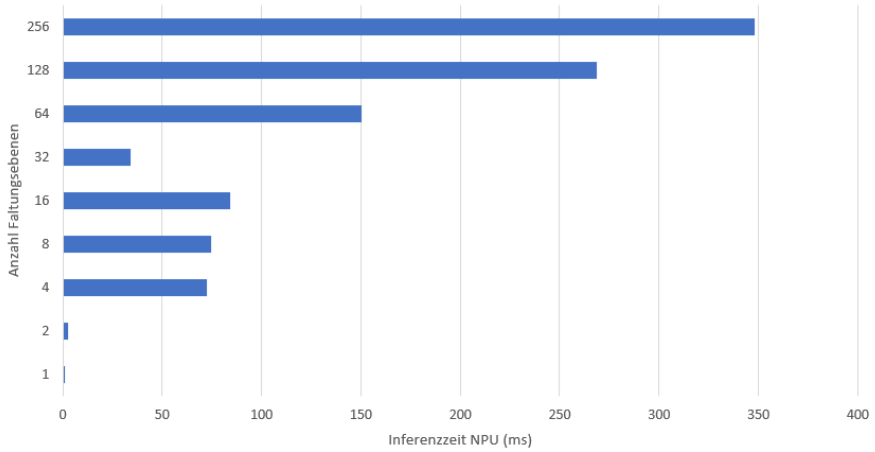


Abbildung 5.19: Inferenz in Abhängigkeit von Anzahl Faltungsebenen

Hierbei ist eine starker Anstieg der Inferenzzeit bei 4 aufeinanderfolgenden Ebenen zu erkennen. Laut AI-Studio wird hier wieder 1 MB aus dem XSPI-RAM benutzt. Dadurch steht der NPU vermutlich wieder ein langsamerer Datentransfer zur Verfügung. Dieser 1 MB bleibt über die Testreihe konstant (Ausnahme bei 32 Ebenen). Die Inferenz steigt wie vermutet mit der steigenden Ebenenzahl, da mehr Berechnungen durchgeführt werden müssen. Der Einbruch bei 32 Ebenen könnte daher kommen, dass die 1 MB aus dem XSPI-RAM nicht genutzt werden. Warum diese hier nicht vorkommen, ist allerdings nicht erkennbar. Werden die Conv2D-Ebenen und die zugehörigen Aktivierungsfunktionen jeweils in einer eigenen HW-Epoche abgebildet, ergibt sich zusammen mit der abschließenden Pooling-Ebene die Anzahl der HW-Epochen zu

$$N_{\text{HW-Epochen}} = 2 \cdot N_{\text{Conv2D}} + 1 \tag{5.1}$$

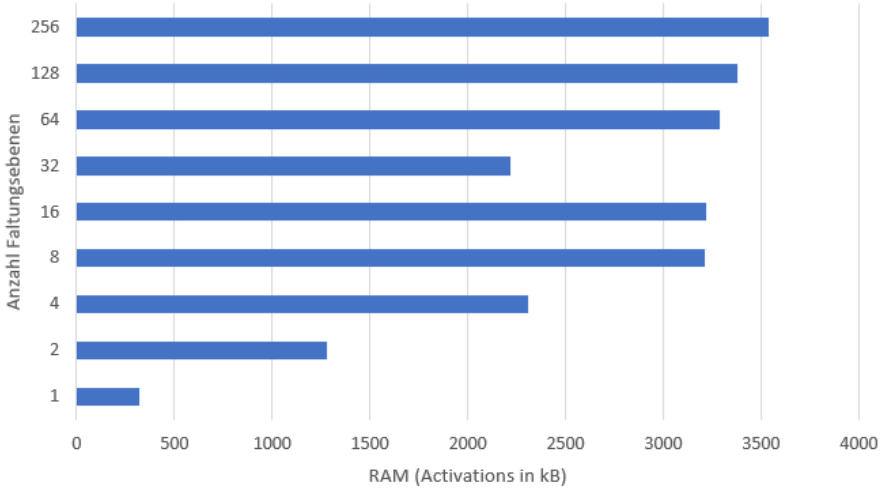


Abbildung 5.20: RAM-Nutzung der Aktivierungen in Abhängigkeit von Anzahl Faltungsebenen

Die RAM-Nutzung steigt zunächst annähernd linear an. Ab 8 Faltungsebenen steigt die RAM-Nutzung wesentlich langsamer an. Es gibt auch einen kleinen Einbruch bei 32 Faltungsebenen. Dieser entspricht vermutlich den vorher beschriebenen, fehlenden 1 MB XSPI-RAM.

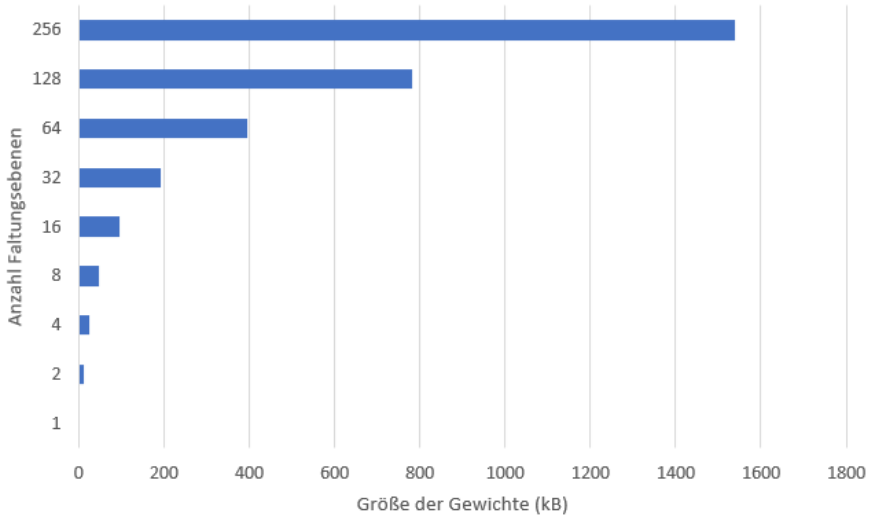


Abbildung 5.21: Speichergröße der Gewichte in Abhängigkeit von Anzahl Faltungsebenen

Wie ebenfalls bereits vermutet steigt die für die Gewichte benötigte Speichergröße linear mit der Ebenenanzahl an. Da die Anzahl der Faltungsebenen stetig verdoppelt wird, verdoppelt sich auch die benötigte Speichergröße.

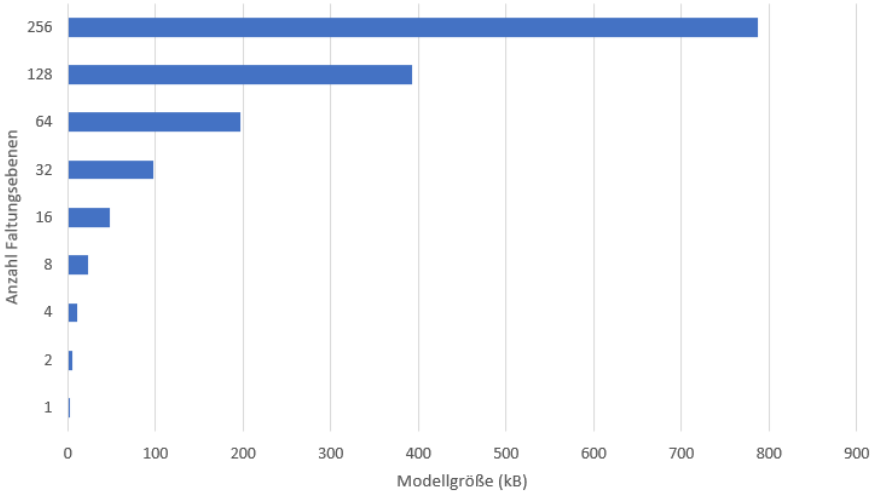


Abbildung 5.22: Modelgröße in Abhängigkeit von Anzahl Faltungsebenen

Auch hier ist ein klarer linearer Zusammenhang zu erkennen. Auch dieser war bereits vorher erwartet.

6 Diskussion

Die durchgeführten Untersuchungen zeigen, dass die tatsächliche Leistungsfähigkeit eines neuronalen Netzes auf einem Mikrocontroller mit integrierter NPU nicht nur anhand der theoretischen Anzahl von Rechenoperationen oder der Anzahl der Modellparameter beurteilt werden kann. Neben der eigentlichen Rechenlast haben insbesondere die verwendete Datenrepräsentation, die Speicheranforderungen und das durch den Compiler erzeugte Mapping auf die verfügbaren Hardware-Ressourcen einen wesentlichen Einfluss auf die resultierende Inferenzzeit.

6.1 Einfluss der Modelloptimierung

Die Untersuchung der verschiedenen Modelle zeigt einen deutlichen Einfluss auf Speicherbedarf und Inferenzzeit. Insbesondere bei den YOLO-Modellen konnte die Inferenzzeit durch die optimierten Modellvarianten um mehrere Größenordnungen reduziert werden. Dabei ist jedoch zu beachten, dass die beobachtete Beschleunigung nicht ausschließlich der Quantisierung zugeschrieben werden kann.

Die theoretische Reduktion des Speicherbedarfs von einer FP32- auf eine INT8-Darstellung beträgt für die reinen numerischen Gewichte den Faktor vier. Die in den Messungen teilweise deutlich höheren Reduktionsfaktoren zeigen daher, dass neben der Quantisierung weitere Veränderungen wirksam sein müssen. Hierzu können unter anderem Gewichtskompression, Pruning, Änderungen der Netzwerkstruktur oder Anpassungen der Ausgangsschicht beitragen. Insbesondere bei den von ST bereitgestellten optimierten Modellen ist eine eindeutige Zuordnung der einzelnen Effekte jedoch nicht möglich, da die vollständige Optimierungspipeline dieser Modelle nicht veröffentlicht wurde.

Die Ergebnisse zeigen somit einen deutlichen praktischen Vorteil der optimierten Modelle, erlauben jedoch keine isolierte Bewertung einzelner Optimierungsverfahren. Für eine solche Untersuchung wäre jeweils ein kontrollierter Versuchsaufbau erforderlich, bei dem zunächst ausschließlich die Quantisierung, anschließend ausschließlich das Pruning und schließlich weitere architektonische Anpassungen untersucht werden.

Abschließend lässt sich jedoch sagen, dass eine Optimierung nötig ist um eine echtzeitfähige Anwendung zu erstellen. Die Inferenzzeiten mit reiner Berechnung auf der CPU sind, bei den komplexeren Netzen, im Sekundenbereich und damit für viele Anwendungen unbrauchbar.

6.2 Einfluss der NPU-Architektur

Die Ergebnisse der fünf strukturellen Experimente zeigen, dass die reale Inferenzleistung der Neural-ART-NPU stark von der konkreten Netzwerkstruktur abhängt. Insbesondere die Compilerbedingte Abbildung auf die NPU-Hardware ist hierbei entscheidend. Diese strukturellen Einschränkungen konnten zwar beobachtet werden, jedoch ist eine eindeutige Erklärung nicht möglich. Zwar konnte eine Korrellation zwischen dem verwendeten RAM-Speicher und der Inferenzzeit beobachtet werden, allerdings konnte dieser Zusammenhang nicht bewiesen werden, da die Daten zur Speicherverwendung während der Inferenz nur sehr begrenzt sind. Hier könnte mit einem spezialisierteren Pipeline eine eindeutigere Korrelation dargestellt werden.

6.2.1 Einfluss der Eingangsgröße

Im ersten Experiment steigt die theoretische Anzahl der MACs bei einer quadratischen Erhöhung der Eingangsauflösung ebenfalls quadratisch. Entsprechend wäre zunächst eine kontinuierliche Zunahme der Inferenzzeit zu erwarten.

Die Messungen zeigen jedoch deutliche Abweichungen von diesem idealisierten Zusammenhang. Besonders auffällig ist der Übergang von 512×512 auf 514×514. Die Inferenzzeit steigt hierbei von $4{,}535\,\mathrm{ms}$ auf etwa $397\,\mathrm{ms}$ und damit auf das rund 87,5-Fache. Gleichzeitig zeigen die Messreihen vor und nach diesem Übergang jeweils wieder einen annähernd linearen Verlauf.

Dieses Verhalten deutet darauf hin, dass an dieser Stelle nicht die reine Anzahl der Rechenoperationen für den Laufzeitsprung verantwortlich ist. Vielmehr ist von einer Änderung des durch den Compiler erzeugten Mappings auszugehen. Möglich sind insbesondere Änderungen bei der Aufteilung der Berechnung, bei der Nutzung der verfügbaren Speicherbereiche oder bei der Anzahl beziehungsweise Struktur der erzeugten Epochen.

Der exakte interne Grenzwert der NPU kann aus den vorliegenden Messungen jedoch nicht bestimmt werden, da die interne Hardwarearchitektur der Neural-ART-NPU nicht vollständig offengelegt ist. Die Ergebnisse belegen damit einen

deutlichen Zusammenhang zwischen der Eingangsgröße und dem Mappingverhalten des Compilers, erlauben jedoch keine eindeutige Aussage über die physikalische Ursache des Grenzwerts.

6.2.2 Einfluss der Filteranzahl

Das zweite Experiment bestätigt die theoretisch erwartete lineare Abhängigkeit der Rechenlast von der Anzahl der Ausgangskanäle beziehungsweise Filter. Gleichzeitig steigen sowohl die benötigte Aktivierungsspeichermenge als auch die Größe der Gewichte annähernd linear an.

Interessant ist hierbei, dass der Compiler den untersuchten Aufbau weiterhin in einer einzelnen HW-Epoche abbilden konnte. Dies deutet darauf hin, dass innerhalb des untersuchten Parameterbereichs keine weitere Aufteilung des Modells erforderlich war.

Damit zeigt dieses Experiment, dass eine Erhöhung der Filteranzahl zunächst relativ gut mit der bestehenden NPU-Struktur skaliert. Erst wenn weitere Ressourcen- oder Mapping-Grenzen erreicht werden, ist mit unregelmäßigen Änderungen der Inferenzzeit zu rechnen.

6.2.3 Grenzbereich der Filteranzahl

Das dritte Experiment zeigt einen deutlichen Sprung der Inferenzzeit im Bereich der zuvor beobachteten Grenze. Auch hier sind vor und nach dem Sprung annähernd lineare Bereiche erkennbar.

Bemerkenswert ist, dass die gemessene RAM-Nutzung keinen vergleichbar ausgeprägten Sprung zeigt. Dies spricht gegen eine rein von der Gesamtmenge des benötigten Speichers abhängige Änderung. Wahrscheinlicher ist, dass die genaue Verteilung der Daten und Operationen auf die verfügbaren Hardware-Ressourcen einen entscheidenden Einfluss besitzt.

Da die internen Mappingregeln des Neural-ART-Compilers nicht vollständig transparent sind, kann diese Vermutung durch die durchgeführten Messungen nicht eindeutig bestätigt werden.

6.2.4 Einfluss von Kernelgröße und Stride

Die Ergebnisse des vierten Experiments zeigen eine starke Abhängigkeit der Inferenzzeit von Kernelgröße und Stride. Insbesondere Stride 3 führt zu einer Auslagerung der Rechenoperationen auf den Host und damit zu einer schlechteren Ausführungszeit. Dies stimmt mit der Herstellerdokumentation überein, nach der Strides von 1, 2 und 4 bevorzugt werden sollen und andere Strides nicht nativ unterstützt beziehungsweise nur ineffizient abgebildet werden können [18].

Auch die auffällige Verschlechterung bei einer Kernelbreite von vier weist darauf hin, dass die theoretische Zahl der Rechenoperationen nicht allein für die Laufzeit verantwortlich ist. Die im AI Studio beobachtete zusätzliche HW-Epoche, sowie die teilweise Nutzung des XSPI-RAMs liefern eine plausible Erklärung für den Anstieg. Ein kausaler Zusammenhang kann aufgrund der fehlenden Transparenz des Compilers jedoch nicht vollständig nachgewiesen werden.

Die scheinbar geringere Effizienz eindimensionaler Kernel ist ebenfalls differenziert zu betrachten. Die Herstellerdokumentation beschreibt eindimensionale Kernel als spezielle Fälle, die effizienter verarbeitet werden können. Dieser Zusammenhang konnte nicht beobachtet werden. Das Gegenteil davon hingegen scheint der Fall zu sein. Für die bestimmte Filterdimension von 1×1 werden insbesondere günstige Kombinationen von Eingangs- und Ausgangskanälen genannt [18]. Diese Bedingungen werden im vorliegenden Experiment mit einem Eingangskanal und 16 Ausgangskanälen nicht erfüllt. Die Messergebnisse stellen daher keinen eindeutigen Widerspruch zum Sonderfall der Herstellerangabe dar, sondern zeigen vielmehr, dass die Effizienz stark von der konkreten Kanalstruktur und dem daraus resultierenden Mapping abhängt.

6.2.5 Einfluss der Netzwerktiefe

Das fünfte Experiment bestätigt zunächst die Erwartung, dass eine steigende Anzahl an Faltungsschichten zu einem höheren Rechenaufwand und damit grundsätzlich zu einer höheren Inferenzzeit führt. Durch das eingesetzte Padding von 1 bleiben die Feature Maps gleich groß. Der Rechenaufwand steigt ebenfalls wie erwartet linear.

Gleichzeitig verändert sich die RAM-Nutzung nicht proportional zur Anzahl der Schichten. Auch der Einbruch der RAM-Nutzung und der Inferenzzeit bei 32 aufeinanderfolgenden Ebenen stellt ein Rätsel dar. Im Bereich bestimmter Netzwerktiefen treten erneut deutliche Änderungen des Laufzeitverhaltens auf.

Dies lässt sich damit erklären, dass bei der Ausführung nicht die Summe aller theoretisch vorhandenen Aktivierungen gleichzeitig im Speicher gehalten werden muss. Durch Speicherwiederverwendung und die konkrete Abbildung des Berechnungsgraphen kann sich die tatsächlich benötigte Spitzenbelegung deutlich von der theoretischen Gesamtmenge unterscheiden. Auch die Nutzung des XSPI-Speichers zeigt, dass die Speicherhierarchie einen wesentlichen Einfluss auf die Laufzeit besitzt.

Damit bestätigt das Experiment die zentrale Erkenntnis der vorherigen Versuche: Die Anzahl der Layer beziehungsweise MACs ist zwar ein wesentlicher Einflussfaktor, ist aber keineswegs ausreichend um die tatsächliche Inferenzleistung auf der NPU vorherzusagen.

6.3 Bewertung der praktischen Leistungsfähigkeit

Die optimierten Modelle zeigen auf dem STM32N6570-DK sehr geringe Inferenzzeiten. Für das ToF-CNN wurde eine Inferenzzeit von ungefähr 0,074, ms gemessen, was rechnerisch etwa 13,500 Inferenzvorgängen pro Sekunde entspricht. Dieses Ergebnis zeigt das Potential der NPU bei einer sehr kleinen und gezielt angepassten Netzwerkarchitektur.

Das ToF-CNN stellt hierbei ein besonders kleines und stark spezialisierte Anwendungsbeispiel dar. Die erreichte Inferenzzeit zeigt das Potential mit dem sehr kleine CNNs auf der Plattform ausgeführt werden können.

Auch die optimierten MobileNet- und YOLO-Modelle erreichen eine, für viele Anwendungen vollkommen ausreichende Inferenzzeit. Für eine vollständige Aussage über die Echtzeitfähigkeit eines eingebetteten Gesamtsystems müssen jedoch zusätzlich Vorverarbeitung, Sensoranbindung, Speichertransfers, Postprocessing und Kommunikation berücksichtigt werden. In einer weiterführenden Arbeit, könnte eine genauere Beobachtung der optimierten Netzarchitektur, mehr Aufschluss darauf liefern, was eine optimale Netzstruktur für die NPU ausmacht.

6.4 Limitationen

Die wichtigste Einschränkung der Untersuchung besteht in der begrenzten Vergleichbarkeit der verwendeten Modellvarianten. Insbesondere bei den durch ST optimierten Modellen sind die Änderungen gegenüber den Ausgangsmodellen kaum dokumentiert.

Dadurch können einzelne Effekte, beispielsweise durch Quantisierung, Pruning oder eine veränderte Netzwerkstruktur, nicht voneinander getrennt werden.

Eine weitere Einschränkung besteht darin, dass der interne Hardwareaufbau und die genaue Mappingstrategie des Neural-ART-Compilers nicht vollständig öffentlich dokumentiert sind. Die in den Experimenten gefundenen Grenzwerte können daher beobachtet und mit Speicher- oder Epochenzuständen korreliert, aber nicht eindeutig auf einzelne interne Hardwarekomponenten zurückgeführt werden.

Darüber hinaus beziehen sich die ermittelten Laufzeiten auf die verwendete Hardware-, Compiler- und Softwarekonfiguration. Änderungen der ST Edge AI Version, der Taktfrequenzen, der Speicherzuordnung oder der Compilerparameter können das resultierende Mapping und damit auch die Laufzeit beeinflussen. Dies wurde zwar während der Experimente beachtet, kann aber zu einer stark begrenzten Reproduzierbarkeit führen.

Für weiterführende Untersuchungen wäre eine genauere Betrachtung der Quantisierung, des Prunings, der Änderung der Klassenzahl und der architektonische Anpassungen unabhängig voneinander sinnvoll. Da aufgrund zeitlicher Begrenzungen die meisten Tests ein- bis zweimal durchgeführt wurden, wäre eine Wiederholung der Messungen mit mehreren Durchläufen und einer statistischen Auswertung der Laufzeit sinnvoll, um Messschwankungen von Mappingeffekten zu unterscheiden.

Zusammenfassend zeigen die Ergebnisse, dass die praktische Leistungsfähigkeit eines Mikrocontrollers mit integrierter NPU wesentlich durch das Zusammenspiel von Netzwerkarchitektur, Datenformat, Speicherhierarchie und Compiler-Mapping bestimmt wird. Um eine optimale Nutzung der NPU zu ermöglichen, müssen strenge Regeln eingehalten werden. Diese sind bisher noch nicht ausreichend dokumentiert, aber durch diese Arbeit ein wenig ersichtlicher.

7 Fazit

Ziel dieser Arbeit war es, die praktische Leistungsfähigkeit eines Mikrocontrollers mit integrierter Neural Processing Unit anhand verschiedener neuronaler Netze zu untersuchen. Als Plattform wurde hierfür das STM32N6570-DK mit einer Neural-ART-14-NPU verwendet. Neben etablierten und aktuellen neuronalen Netzen zur Objekterkennung und Klassifizierung wurde zusätzlich ein eigenes CNN zur Klassifizierung von Time-of-Flight-Sensordaten entwickelt. Die Untersuchung umfasste dabei insbesondere die Inferenzzeit, den Speicherbedarf und das Verhalten des Compilers bei unterschiedlichen Netzwerkstrukturen.

Die Ergebnisse zeigen, dass die Verwendung der NPU insbesondere bei quantisierten CNNs zu einer erheblichen Reduktion der Inferenzlatenz führen kann. Die Quantisierung ist auch die Mindestvorraussetzung zur Nutzung der NPU. Die untersuchten optimierten Modelle konnten gegenüber den jeweiligen Ausgangsvarianten eine deutliche Beschleunigung (Faktoren 200-370), mit relativ geringem Genauigkeitsverlust erreichen. Dabei darf der beobachtete Geschwindigkeitsgewinn jedoch nicht ausschließlich der Quantisierung zugeschrieben werden, da bei den verwendeten optimierten Modellen (Yolov8n, Yolo26n, MNetv3-Small) zusätzliche Änderungen der Netzwerkstruktur, des Detection Heads oder weitere Optimierungen nicht vollständig nachvollziehbar sind.

Die Untersuchungen zur NPU-Struktur zeigen zudem, dass die tatsächliche Inferenzzeit nicht ausschließlich durch die Anzahl der theoretisch erforderlichen Rechenoperationen bestimmt wird. Besonders deutlich wurde dies durch die Veränderung der Eingangsauflösung. Während die theoretische Anzahl der MACs kontinuierlich quadratisch mit der Auflösung steigt, wurde bei 512×512 beziehungsweise 514×514 ein deutlicher Laufzeitsprung beobachtet. Vergleichbare Effekte traten bei bestimmten Filteranzahlen und Kerneldimensionen auf. Die Ergebnisse deuten darauf hin, dass die Speicherorganisation und das durch den Compiler gewählte Hardware-Mapping einen wesentlichen Einfluss auf die tatsächliche Leistung besitzen. Genaue Hintergründe zu den Mappingregeln, konnten allerdings nicht getroffen werden.

Auch die Untersuchung von Kerneldimensionen und Schrittweiten(Strides) bestätigt, dass eine gezielte Anpassung der Netzwerkarchitektur an die Eigenschaften der NPU unbedingt nötig ist, um einen Mehrwehrt aus der NPU zu ziehen. Insbesondere eine Stride von 3 führt zu einer stärkeren Auslagerung auf den Host. Gleichzeitig konnte gezeigt werden, dass die Angaben von ST zur effizienten Ausführung bestimmter Filterdimensionen nicht vollständig korrekt sind. Eine einheitliche Regel konnte über das Vermeiden von Stride 3 jedoch nicht gefunden werden.

Das selbst entwickelte ToF-CNN zeigt darüber hinaus mit seiner Inferenzzeit von 0,074 ms das große Potential eines kleinen spezialisierten und optimierten Netzes.

In Bezug auf die Genauigkeit zeigen die Untersuchungen insgesamt nur vergleichsweise geringe Verluste nach der Optimierung. Eine eindeutige Zuordnung dieser Veränderungen zu einzelnen Optimierungsverfahren ist aufgrund der spärlich dokumentierten Modelländerungen jedoch nicht möglich. Die Genauigkeitswerte sollten daher primär als Bezugswert der jeweils verwendeten Modellvarianten genutzt werden.

Die zentrale Erkenntnis der Arbeit ist somit, dass die Leistungsfähigkeit eines NPU-beschleunigten Mikrocontrollers nicht nur durch dessen theoretische Rechenleistung charakterisiert werden kann. Entscheidend ist vielmehr das Zusammenspiel aus Netzwerkarchitektur, Datenformat, Speicherbedarf und der Fähigkeit des Compilers, das Netzwerk effizient auf die vorhandenen Hardware-Ressourcen abzubilden. Insbesondere bei kleinen, quantisierten und an die NPU angepassten Netzwerken kann dadurch eine sehr hohe Inferenzleistung erreicht werden.

Die Ergebnisse zeigen, dass der Einsatz eines Mikrocontrollers mit integrierter NPU für lokale Machine-Learning-Anwendungen durchaus realistisch ist und auch die echtzeitgetreue Ausführung von komplexeren Modellen zur Objekterkennung ermöglicht. Die konkrete Eignung hängt von der Anwendung, der geforderten Genauigkeit, der Eingangsgröße, der zulässigen Inferenz sowie den verfügbaren Speicher- und Energie-Ressourcen ab.

Damit konnte die Arbeit zeigen, dass die Neural-ART-NPU des STM32N6570-DK insbesondere dann ein hohes Potential bietet, wenn neuronale Netze gezielt auf die Eigenschaften der NPU optimiert werden.

Literatur

[1] S. Branco, A. G. Ferreira und J. Cabral. „Machine Learning in Resource-Scarce Embedded Systems, FPGAs, and End-Devices: A Survey“. In: *Electronics* 8.11 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8111289. URL: https://www.mdpi.com/2079-9292/8/11/1289.

[2] COCO. *Coco Datasets (2017)*. URL: https://cocodataset.org/#download (besucht am 21. 09. 2026).

[3] L. Dutta und S. Bharali. „Tinyml meets iot: A comprehensive survey“. In: *Internet of Things* 16 (2021), S. 100461.

[4] F. Gül. „Energy efficiency of AI hardware: a systematic review of GPU, TPU, and NPU architectures in the LLM era“. In: *J. King Saud Univ. Comput. Inf. Sci.* 38 (2026), S. 499. DOI: 10.1007/s44443-026-00900-6. URL: https://doi.org/10.1007/s44443-026-00900-6.

[5] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le und H. Adam. *Searching for MobileNetV3*. 2019. arXiv: 1905.02244 [cs.CV]. URL: https://arxiv.org/abs/1905.02244.

[6] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam und D. Kalenichenko. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. arXiv: 1712.05877 [cs.LG]. URL: https://arxiv.org/abs/1712.05877.

[7] G. Jocher, A. Chaurasia und J. Qiu. *Ultralytics YOLOv8*. Version 8.0.0. 2023. URL: https://github.com/ultralytics/ultralytics.

[8] G. Jocher, J. Qiu, M. Liu, S. Lyu, F. C. Akyon und M. E. Kalfaoglu. *Ultralytics YOLO26: Unified Real-Time End-to-End Vision Models*. 2026. arXiv: 2606.03748 [cs.CV]. URL: https://arxiv.org/abs/2606.03748.

[9] S. J. Kwon, D. Lee, B. Kim, P. Kapoor, B. Park und G.-Y. Wei. „Structured Compression by Weight Encryption for Unstructured Pruning and Quantization“. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Juni 2020.

[10] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick und P. Dollár. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV]. URL: https://arxiv.org/abs/1405.0312.

[11] S. Mittal. „A survey of FPGA-based accelerators for convolutional neural networks“. In: *Neural Computing and Applications* 32 (2020), S. 1109–1139. DOI: 10.1007/s00521-018-3761-1. URL: https://doi.org/10.1007/s00521-018-3761-1.

[12] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen und T. Blankevoort. *A White Paper on Neural Network Quantization*. 2021. arXiv: 2106.08295 [cs.LG]. URL: https://arxiv.org/abs/2106.08295.

[13] J. Pleterski, G. Škulj, C. Esnault, J. Puc, R. Vrabič und P. Podržaj. „Miniature Mobile Robot Detection Using an Ultralow-Resolution Time-of-Flight Sensor“. In: *IEEE Transactions on Instrumentation and Measurement* 72 (2023), S. 1–9. DOI: 10.1109/TIM.2023.3318710.

[14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg und L. Fei-Fei. „ImageNet Large Scale Visual Recognition Challenge“. In: *International Journal of Computer Vision* 115.3 (2015), S. 211–252. DOI: 10.1007/s11263-015-0816-y.

[15] W. Shi, J. Cao, Q. Zhang, Y. Li und L. Xu. „Edge Computing: Vision and Challenges“. In: *IEEE Internet of Things Journal* 3.5 (2016), S. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[16] Shiyuan Zhang. „Model Deployment on the STM32N6: Feasibility Analysis and Performance Evaluation of the NPU“. 2025.

[17] STMicroelectronics. *Introduction to ST Neural-Art accelerator*. URL: https://stedgeai-dc.st.com/assets/embedded-docs/stneuralart_programming_model.html (besucht am 21. 09. 2026).

[18] STMicroelectronics. *ST Edge AI Core User Manual – Neural Art operators*. Version 4.0.1. URL: https://dev.st.com/edge-ai-docs/st-edge-ai-core/4.0.1/en/docs/user-manual/core-neural-art-operators.html (besucht am 21. 09. 2026).

[19] STMicroelectronics. *ST Edge AI Developer Cloud*. URL: https://stm32ai-cs.st.com/ (besucht am 21. 09. 2026).

[20] STMicroelectronics. *STM MNetV3_small Modell Optimiert*. URL: https://github.com/STMicroelectronics/stm32ai-modelzoo/tree/main/object_detection/ssdlite_mobilenetv3small_pt/ST_pretrainedmodel_public_dataset/coco_person/ssdlite_mobilenetv3small_pt_coco_person_300 (besucht am 21. 09. 2026).

[21] STMicroelectronics. *STM Modelzoo*. URL: https://github.com/STMicroelectronics/stm32ai-modelzoo/tree/main/object_detection (besucht am 21. 09. 2026).

[22] STMicroelectronics. *STM Yolo26n Modell Optimiert*. URL: https://github.com/stm32-hotspot/ultralytics/raw/refs/heads/main/examples/YOLOv8-STEdgeAI/stedgeai_models/object_detection/yolo26/yolo26_256_qdq_int8_od_coco-person-st.onnx (besucht am 21. 09. 2026).

[23] STMicroelectronics. *STM Yolov8n Modell Optimiert*. URL: https://github.com/stm32-hotspot/ultralytics/raw/refs/heads/main/examples/YOLOv8-STEdgeAI/stedgeai_models/object_detection/yolov8n_256_quant_pc_uf_od_coco-person-st.tflite (besucht am 21. 09. 2026).

[24] Ultralytics. *Ultralytics Coco128 Dataset*. URL: https://github.com/ultralytics/assets/releases/download/v0.0.0/coco128.zip (besucht am 21. 09. 2026).

[25] Ultralytics. *Ultralytics Yolo26*. URL: https://docs.ultralytics.com/models/yolo26 (besucht am 21. 09. 2026).

[26] Ultralytics. *Ultralytics Yolov8*. URL: https://docs.ultralytics.com/models/yolov8 (besucht am 21. 09. 2026).

[27] H. Yang Zhengwuand Zhang. „Comparative Analysis of Structured Pruning and Unstructured Pruning“. In: *Frontier Computing*. Hrsg. von J. C. Hung, N. Y. Yen und J.-W. Chang. Singapore: Springer Nature Singapore, 2022, S. 882–889. ISBN: 978-981-16-8052-6.