

Trustable Machine Learning Development
for Edge Devices

Miguel Costa

UMinho | 2025

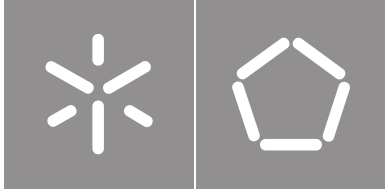


Universidade do Minho
Escola de Engenharia

Miguel Ângelo Peixoto Costa

Trustable Machine Learning Development for Edge Devices

Julho de 2025



Universidade do Minho
Escola de Engenharia

Miguel Ângelo Peixoto Costa

Trustable Machine Learning Development for Edge Devices

Tese de Doutoramento
Programa Doutoral em Engenharia Eletrónica e Computadores

Trabalho efetuado sob a orientação de:
Investigador Principal Doutor Sandro Emanuel Salgado Pinto
Professor Doutor Adriano José Conceição Tavares

Julho de 2025

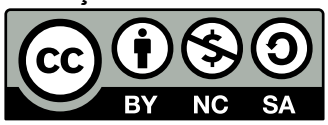
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositoriUM of Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

CC BY-NC-SA 4.0

`https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en`

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.0.2) .

Acknowledgements

I want to begin by expressing my deepest gratitude to my advisor, Prof. Dr. Sandro Pinto. His guidance was fundamental to the completion of this dissertation. I am particularly thankful for the knowledge he provided, the scientific rigor he instilled, and the methodological clarity he brought to this work. In addition, a special thanks for all the tranquility and motivation you conveyed in the most turbulent and anxious moments, and for the friendship and availability you showed throughout this journey.

I extend my heartfelt thanks to my labmates for their friendship, support, and the many joyful moments we shared. In particular, I am deeply thankful to Sérgio, who joined me in Thailand. His companionship through both the highs and lows made a significant difference. (And yes, what happened in Thailand, stays in Thailand!). A special thanks also to Prof. Dr. Tiago Gomes for his encouragement, support, and friendship, especially during the more difficult phases. I am truly fortunate to have developed friendships during this journey that I hope to carry with me for life.

To my parents, António and Esperança, thank you for your unconditional love and unwavering support, not only throughout this thesis but across my entire life. The values you instilled in me have shaped the person I am today, and for that, I am forever grateful.

To my closest friends (a.k.a "Coronas de Família") thank you for the joy, laughter, and countless unforgettable moments. It is heartwarming to see how our friendship, rooted in childhood, continues to thrive into adulthood. By the way, it was fun to see how we transitioned from hopping around bars to hopping around protein bars! A special thanks also to my best friend Fábio for all the laughs during our unexpected road trips. Let's also not forget my friends "Estudantes da FEUP" for all the joy and memories we made in our Techno parties.

Finally, I want to reserve a special thanks for someone who is also very special. If you ever read this, you will know that these words are for you. You were often my first source of comfort during the most turbulent moments of this journey. I promise to carry you with me for the rest of my life.

To my remaining family and friends who supported me throughout my academic journey: thank you, sincerely!

This thesis was supported by a PhD scholarship (grant SFRH/BD/146780/2019) from *Fundação para a Ciência e Tecnologia*.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

_____, _____

(Place) (Date)

_____

(Miguel Ângelo Peixoto Costa)

Resumo

Desenvolvimento de Sistemas de Machine Learning Robustos para Dispositivos no Edge da Rede

A consciencialização da sociedade sobre a privacidade de dados e a adoção de *Machine Learning* (ML) em cenários críticos têm impulsionado a transição da computação na nuvem para a computação de ML no dispositivo local. Entre os dispositivos conectados à Internet, os microcontroladores (MCUs) têm despertado particular interesse. As mais recentes arquiteturas suportam a aceleração de ML, mantendo o baixo consumo de energia que sempre caracterizou estes dispositivos (TinyML).

Perante a atual tendência de *edge computing*, os criadores de modelos de ML têm demonstrado preocupação com a sua propriedade intelectual. A implementação de modelos em dispositivos não confiáveis pode expor algoritmos proprietários, através de engenharia reversa, ou comprometer dados privados utilizados no treino. Os *Trusted Execution Environments* (TEEs) têm sido amplamente explorados para garantir a privacidade de modelos de ML em dispositivos com processadores x86 e Arm Cortex-A. Contudo, a viabilidade desta estratégia, baseada em TEEs, em MCUs continua por explorar.

Em paralelo, os TEEs não garantem a integridade das decisões dos modelos. Estudos anteriores demonstram que redes neuronais de precisão total são vulneráveis a *adversarial examples* e à deformação de objetos. No entanto, os sistemas de ML baseados em MCUs tendem a usar redes neuronais quantizadas (QNNs), cuja robustez a *adversarial examples* é, em parte, desconhecida. Para combater a vulnerabilidade à deformação de objetos foi proposto uma nova arquitetura de ML - *Capsule Networks* (CapsNets). Contudo, a viabilidade de usar CapsNets em MCUs permanece desconhecida.

Esta tese aborda estes desafios em três contribuições/artigos. Na primeira contribuição, propomos uma *framework* de código aberto que utiliza o TrustZone-M para proteger a privacidade de QNNs em MCUs Arm Cortex-M. Esta framework identifica as layers críticas do ponto de vista da privacidade, reduzindo ao mínimo o uso de memória e a computação dentro do TEE. Na segunda contribuição, realizámos uma avaliação empírica da robustez de QNNs otimizadas para TinyML perante *adversarial attacks* estado-da-arte. Além disso, avaliamos a eficácia de defesas previamente concebidas para redes neuronais de precisão total quando aplicadas a QNNs e que restrições os MCUs impõem à implementação destas defesas. Por fim, a nossa terceira contribuição apresenta uma *framework* de quantização para CapsNet e estende as bibliotecas estado-da-arte de ML para MCUs para suportar a execução de CapsNets quantizadas.

Palavras-chave: MCUs, TinyML, Quantização, Privacidade em ML, Integridade em ML, CapsNets

Abstract

Trustable Machine Learning Development for Edge Devices

The growing demand for data privacy and the adoption of Machine Learning (ML) in mission-critical applications is accelerating the shift from cloud-based to edge-based computation. Among edge devices, Microcontroller Units (MCUs) have attracted particular attention due to their emerging support for lightweight ML execution and power efficiency, giving rise to the field of TinyML.

Facing the current edge computing trend, ML model designers are raising concerns about their intellectual property. Deploying ML models in untrusted edge devices can expose proprietary algorithms by reverse engineering or leak private training data. While Trusted Execution Environments (TEEs) have been widely explored as a countermeasure on platforms such as Arm Cortex-A and x86, their applicability to low-power MCUs remains underexplored. The introduction of TrustZone-M in Arm Cortex-M MCUs makes it worth exploring if TEEs are also a feasible solution for ML model privacy in this class of devices.

If TEEs are the foundation for trustworthy ML, they are not enough to guarantee decision integrity. Previous research has shown that adversarial examples and object deformation are well-known vulnerabilities of full-precision Artificial Neural Networks (ANNs). However, TinyML systems powered by MCUs usually rely on Quantized Neural Networks (QNNs), and their vulnerability to adversarial examples remains misunderstood. Furthermore, if QNNs are indeed vulnerable to adversarial examples, it is fundamental to understand how TinyML shapes the defense design. Regarding object deformation, a new ML architecture - Capsule Networks (CapsNets) - has been proposed to tackle this issue. However, the feasibility of integer-precision CapsNets in MCUs is yet to be demonstrated.

This thesis addresses these research challenges in three contributions/articles. First, we propose an open-source framework that leverages TrustZone-M to protect the privacy of QNNs in Arm Cortex-M MCUs. Our framework determines the privacy-critical layers, reducing the memory footprint of the TEE to the bare minimum. In our second contribution, we performed a comprehensive empirical evaluation of the adversarial robustness of QNNs optimized for TinyML. We analyzed how they perform in the face of state-of-the-art attacks and how current defenses, for full-precision ANNs in server-like scenarios, perform and adapt to TinyML. Our last contribution proposes a quantization framework for CapsNets and extends MCU-compatible ML Application Programming Interfaces (APIs) to support quantized CapsNet inference.

Keywords: MCUs, TinyML, Quantized Neural Networks, ML Privacy, ML Integrity, CapsNets

Contents

List of Figures **xi**

List of Tables **xii**

List of Listings **xiv**

Acronyms **xv**

I Dissertation Summary **1**

1 Introduction **2**

1.1 Motivation 2

1.1.1 Machine Learning is Shifting to the Edge 2

1.1.2 TinyML: The New Trend in Artificial Intelligence? 3

1.1.3 Proprietary Machine Learning in Untrusted Edge Devices 4

1.1.4 Integrity of the Decision: Adversarial Attacks 5

1.1.5 Integrity of the Decision: Object Deformation 6

1.2 Contributions 7

1.2.1 Research Challenges 7

1.2.2 Research Questions 8

1.2.3 Main Contributions 9

1.2.4 Other Contributions 11

1.3 Thesis Outline 12

2 Background **13**

2.1 Microcontroller Systems 13

2.1.1 Arm Cortex-M 14

2.1.2 RISC-V RV32IMCXpulp . . . 16
2.2 Trusted Execution Environments . . . 17
2.2.1 Arm TrustZone-(M) . . . 17
2.3 Quantization in Machine Learning . . . 19
2.4 Adversarial Examples . . . 22
2.4.1 Attacks . . . 23
2.4.2 Defenses . . . 28
2.5 Capsule Networks . . . 31
2.5.1 Transforming Auto-Encoders . . . 32
2.5.2 Capsule Networks with Dynamic Routing . . . 32
2.5.3 Matrix Capsules with Expectation-Maximization Routing . . . 34

3 State of the Art **36**
3.1 Privacy of Machine Learning Models . . . 36
3.1.1 Full Artificial Neural Network on Trusted Execution Environment . . . 37
3.1.2 Extension of Trusted Execution Environments . . . 38
3.1.3 Model Refactoring . . . 38
3.1.4 Weight Obfuscation . . . 39
3.1.5 Selected Layers on Trusted Execution Environment . . . 39
3.2 QNN Robustness Against Adversarial Examples . . . 40
3.2.1 Defenses . . . 42
3.3 Capsule Networks at the Edge . . . 43
3.3.1 Software-Hardware Co-Design . . . 43
3.3.2 Hardware Solutions . . . 45
3.3.3 Software Solutions . . . 45
3.4 Gap Analysis . . . 47
3.4.1 Privacy of Machine Learning Models . . . 47
3.4.2 QNN Robustness Against Adversarial Examples . . . 49
3.4.3 Capsule Networks at the Edge . . . 52

4 Concluding Remarks **54**
4.1 Conclusions . . . 54
4.2 Future Work . . . 57

II Publications **59**

5 SecureQNN: Shielding the Intellectual Property of QNNs in TinyML Systems **61**
5.1 Introduction . . . 62

5.2 Methodology . . . 65
5.2.1 Scope . . . 65
5.2.2 Threat Model . . . 66
5.3 Background and Related Work . . . 67
5.3.1 Arm TrustZone(-M) . . . 67
5.3.2 TinyML: The new frontier in AI? . . . 68
5.3.3 Generative Adversarial Networks . . . 69
5.3.4 Related Work . . . 69
5.3.5 Gap Analysis . . . 71
5.4 SecureQNN . . . 73
5.4.1 Surrogate Dataset Generator . . . 74
5.4.2 Model Partitioning . . . 75
5.4.3 Trusted Execution . . . 77
5.5 Evaluation . . . 78
5.5.1 Surrogate Dataset . . . 78
5.5.2 Privacy vs. Memory Trade-Off . . . 81
5.5.3 Latency . . . 83
5.6 Discussion . . . 84
5.7 Conclusion . . . 85

6 An Empirical Evaluation of Attacks and Defenses for QNNs at the Deep Edge 86
6.1 Introduction . . . 87
6.2 Motivation and Methodology . . . 90
6.2.1 TinyML: The New Wave in AI . . . 90
6.2.2 Scope . . . 90
6.2.3 Adversary Model . . . 91
6.2.4 Selected Datasets and Models . . . 92
6.2.5 Selected Attacks and Defenses . . . 93
6.2.6 Empirical Evaluation . . . 95
6.2.7 Threats to Validity . . . 96
6.2.8 Summary of Observations . . . 97
6.3 Robustness Against Adversarial Examples . . . 97
6.3.1 Direct Attacks . . . 97
6.3.2 Transferability . . . 99
6.4 Defenses Against Adversarial Examples . . . 102
6.4.1 Train-Based Defenses . . . 103
6.4.2 Pre-Processing Defenses . . . 106
6.4.3 Train-Based vs. Pre-Processing Defenses . . . 108

6.5 Related Work and Gap Analysis . . . 108
6.5.1 Related Work . . . 108
6.5.2 Gap Analysis . . . 109
6.6 Discussion . . . 111
6.7 Conclusion . . . 111

7 Shifting Capsule Networks from the Cloud to the Deep Edge 115
7.1 Introduction . . . 116
7.2 Background . . . 118
7.2.1 Capsule Networks: Overview and Evolution . . . 118
7.2.2 Capsule Networks with Dynamic Routing . . . 120
7.2.3 Quantization . . . 121
7.2.4 CMSIS-NN . . . 123
7.2.5 PULP-NN . . . 124
7.3 Software Kernels . . . 124
7.3.1 Matrix Multiplication . . . 124
7.3.2 Squash Activation and Vector Norm . . . 127
7.3.3 Primary Capsule Layer . . . 128
7.3.4 Capsule Layer . . . 130
7.4 CapsNet Quantization . . . 132
7.5 Evaluation . . . 133
7.5.1 Quantization . . . 135
7.5.2 Software Kernels . . . 135
7.6 Related Work . . . 139
7.6.1 Gap Analysis . . . 140
7.7 Conclusion . . . 141

Bibliography 142

List of Figures

1 Relationship between research challenges, research questions, and scientific contributions 9

2 Comparison of deep edge, edge, and cloud computation 14
3 Classic architecture of a Trusted Execution Environment based on Arm TrustZone-M . . . 18
4 Adversarial example in image data 22
5 Basic principle of Boundary Attack 28
6 Basic principle of Capsule Networks 31
7 General architecture of a Capsule Network with dynamic routing between capsules . . . 33

8 SecureQNN threat model 66
9 Training of a Generative Adversarial Network 69
10 High-level architecture of SecureQNN 73
11 Workflow of the Model Partitioning module 76
12 Architecture of a Machine Learning system provisioned by SecureQNN 77
13 Fréchet Inception Distance over training steps for CIFAR-10 Generative Adversarial Networks 79
14 CIFAR-10 samples vs. synthetic samples 80
15 Fréchet Inception Distance over training steps for VWW Generative Adversarial Networks 80
16 VWW samples vs. synthetic samples 81

17 Accuracy *vs.* L_∞ distortion for Projected Gradient Descent attack 100

18 Basic principle of Capsule Networks 118
19 General architecture of a Capsule Network with dynamic routing between capsules . . . 120
20 Matrix multiplication in Arm Cortex-M microcontrollers using *mat_mult_q7_trb* 125

List of Tables

1 Single Instruction Multiple Data of Armv7E-M and Armv8-M for QNNs 15
2 Single Instruction Multiple Data of RISC-V RV32IMCXpulp for QNNs 17
3 Metrics to quantify the distortion of adversarial examples 23
4 Main characteristics of state-of-the-art adversarial attacks 24
5 Main characteristics of state-of-the-art adversarial defenses 29

6 Gap analysis: privacy of Machine Learning models. 48
7 Gap analysis: adversarial robustness of QNNs 50
8 Gap analysis: Capsule Networks at the edge 52

9 Gap analysis: privacy of Machine Learning models 72
10 Evaluation of Generative Adversarial Networks 79
11 Minimum TEE memory footprint vs. maximum accuracy of surrogate QNN 81
12 TEE memory footprint of a minimalist application built upon TF-M v1.7 82
13 Decision latency per sample for different privacy settings 83

14 Benchmarking suite for QNNs adversarial robustness 92
15 Adversarial example attacks 94
16 Adversarial example defenses 95
17 Test accuracy of the models before adversarial attack 97
18 Quantization impact on direct attacks 98
19 Quantization impact on transferability 100
20 Gradient cosine similarity: ANNs vs. QNNs 101
21 Accuracy of ANNs/QNNs enhanced with train-based defenses 103
22 Evaluation of train-based adversarial defenses (L_2 distortion) 104
23 Accuracy of ANNs/QNNs enhanced with pre-processing defenses 106
24 Evaluation of Feature Squeezing 106
25 Evaluation of Pixel Defend 107

26 Gap analysis: adversarial robustness of QNNs 110
27 Configuration of adversarial attacks 112
28 Configuration of adversarial defenses 113
29 Evaluation of train-based adversarial defenses (all distortion metrics - Part 1) 113
30 Evaluation of train-based adversarial defenses (all distortion metrics - Part 2) 114

31 Baseline reference Capsule Networks for evaluation 134
32 Evaluation of the quantization framework for Capsule Networks 135
33 Evaluation of matrix multiplication functions for Arm Cortex-M MCUs 136
34 Evaluation of matrix multiplication functions for RISC-V MCUs 136
35 Evaluation of primary capsule functions for Arm Cortex-M MCUs 137
36 Evaluation of primary capsule functions for RISC-V MCUs 138
37 Evaluation of capsule functions for Arm Cortex-M MCUs 138
38 Evaluation of capsule functions for RISC-V MCUs 139
39 Gap analysis: Capsule Networks at the edge 141

List of Listings

1 Dynamic routing in Capsule Networks 34
2 Expectation-Maximization in Capsule Networks 35
1 Dynamic routing in Capsule Networks 121
2 *mat_mult_q7_simd(a, b, shift, r)* for Arm Cortex-M 126
3 *mat_mult_q7_simd(a, b, shift, r)* for RISC-V 127
4 Newton-Raphson method to calculate the square root of n 128
5 *capsule_layer_q7(args)* for Arm Cortex-M and RISC-V 130
6 Quantization framework for Capsule Networks 132
7 Quantization algorithm for Capsule Networks 133

Acronyms

ADC Analog-to-Digital Converter
ADMM Alternating Direction Method of Multipliers
ANN Artificial Neural Network
API Application Programming Interface
APU Application Processor Unit
ASIC Application-Specific Integrated Circuit

BA Boundary Attack
BIM Basic Iterative Method

C&W Carlini and Wagner
CA Client Application
CAGR Compound Annual Growth Rate
CapsNet Capsule Network
CNN Convolutional Neural Network
COTS Commercial Off-the-Shelf
CPU Central Processing Unit

DPU Deep Processing Unit
DSP Digital Signal Processing

EAD Elastic-Net Attack to DNNs
EBM Energy-Based Model
EM Expectation-Maximization

FAB Fast Adaptive Boundary

FEECA Fast Energy-Efficient CapsNet Accelerator
FGSM Fast Gradient Sign Method
FID Fréchet Inception Distance
FLOP Floating Point Operation
FPGA Field-Programmable Gate Array
FPU Floating-Point Unit

GAN Generative Adversarial Network
GeoDA Geometric Decision-Based Attack
GPU Graphics Processing Unit

HWC Height-Width-Channel

I2C Inter-Integrated Circuit
IDAU Implementation-Defined Attribution Unit
IFGSM Iterative Fast Gradient Sign Method
IFSM Iterative Fast Gradient Method
IoT Internet of Things
ISA Instruction Set Architecture

JSMA Jacobian-based Saliency Map Approach

LAKP Look-Ahead Kernel Pruning
LUT Lookup Table

MAC Multiply-and-Accumulate
MCU Microcontroller Unit
MIA Membership Inference Attack
MIM Momentum Iterative Method
ML Machine Learning
MLaaS Machine Learning as a Service
MMU Memory Management Unit
MVE M-Profile Vector Extension

NS Non-Secure

OMG Offline Model Guard

OS Operating System

PbMA Parameter-based Memory Allocation
PDF Probability Density Function
PE Processing Element
PGD Projected Gradient Descent
PIM Processing-In-Memory
pKVM protected Virtual Machine
PSA Platform Security Architecture

QNN Quantized Neural Network

RAM Random-Access Memory
REE Rich Execution Environment
ReLU Rectified Linear Unit
RGB Red-Green-Blue

S2M Single-to-Multiple
SA Square Attack
SAT Sinkhorn Adversarial Training
SAU Security Attribution Unit
SCR Secure Configuration Register
SGD Stochastic Gradient Descent
SIMD Single Instruction Multiple Data
SoC System on Chip
SP Service Provider
SPI Serial Peripheral Interface
SPSA Simultaneous Perturbation Stochastic Approximation
SRAM Static Random Access Memory
STE Straight Through Estimator
SVHN Street View House Numbers

TA Trusted Application
TAE Transforming Auto-Encoder
TCB Trusted Computing Base
TEE Trusted Execution Environment
TF-M Trusted Firmware-M

TPU Tensor Processing Unit

TZASC TrustZone Address Space Controller

TZMA TrustZone Memory Adapter

UART Universal Asynchronous Receiver-Transmitter

UAV Unmanned Aerial Vehicle

VWW Visual Wake Words

xPU Heterogeneous System

ZOO Zeroth-Order Optimization

Part I

Dissertation Summary

1 Introduction

Growing concerns about user data privacy and using Machine Learning (ML) in mission-critical scenarios are driving the shift of ML computation from the cloud to the deep edge. Among edge devices, Microcontroller Units (MCUs) have garnered significant attention due to their widespread availability, substantially lower cost, and reduced power consumption compared to Application Processor Units (APUs), Graphics Processing Units (GPUs), and Tensor Processing Units (TPUs). Quantization has proven effective in deploying lightweight neural networks on MCUs with minimal accuracy loss. However, limited research exists on methods to ensure the integrity of decisions and protect the intellectual property of ML models on MCUs. Furthermore, the applicability of existing defense mechanisms - designed to address these challenges in cloud-centric environments utilizing GPUs and APUs - have yet to be comprehensively evaluated for resource-constrained MCUs.

1.1 Motivation

1.1.1 Machine Learning is Shifting to the Edge

The proliferation of big data and significant advancements in computational power turned ML into a key technology to drive innovation and assist humans across diverse domains, including healthcare [1], finance [2], and mobility [3, 4]. Considering that ML computation tends to be computationally intensive, most ML systems to date follow a cloud-centered architecture, where data collected by edge devices is transmitted over the network to a cloud server. The cloud performs the ML computation, sending the inference decision back to edge devices, responsible for actuating over the environment [5, 6].

However, growing concerns related to user data privacy and the appliance of ML in mission-critical scenarios have prompted a paradigm shift toward deploying ML computations closer to the data source, at the deep edge [5–18]. Edge computing mitigates privacy risks for the user by eliminating the need to transfer personal and private data over a network to a third party, thus reducing exposure to potential breaches and increasing the user's confidence in ML-based systems. In addition, the inference latency becomes independent from the network bandwidth and the availability of cloud resources, making ML

systems predictable as required in real-time and mission-critical scenarios. Given the advantages of edge computing, academia and industry have been developing software and hardware solutions to shift ML workloads to the deep edge, providing it with the intelligence required to infer and actuate over the surrounding environment autonomously [6, 8, 14, 18–23].

1.1.2 TinyML: The New Trend in Artificial Intelligence?

Among the array of devices commonly deployed at the edge of the network, MCUs have garnered significant attention. Modern MCU architectures, featuring Single Instruction Multiple Data (SIMD) to accelerate ML [18, 21], coupled with developer-friendly ML Application Programming Interfaces (APIs) like CMSIS-NN [18], PULP-NN [19], and TensorFlow Lite Micro [20], are enabling ML deployment on devices that are orders of magnitude cheaper and less power-hungry than APUs or GPUs. Although MCUs are highly resource-constrained, Artificial Neural Network (ANN) compression techniques such as quantization have shown to significantly enhance inference latency while maintaining negligible impact on model accuracy [18, 19]. These features, aligned with the ongoing trend to shift ML to the edge is making TinyML - a branch of ML focused on hardware and software solutions to enable ML deployment in devices consuming 1 milliWatt or less [24] - one of the hottest topics in ML research [24–29].

According to a 2022 report by ABI Research [30], the TinyML Software-as-a-Service market generated over US$220 million in revenue that year and is anticipated to reach a billion-dollar valuation by 2030. This company also estimates that the TinyML market will expand from 15.2 million shipments in 2020 to 2.5 billion by 2030 [31]. Regarding the MCU market size, a report from Mordor Intelligence [32] estimates that this market is expected to reach US$51.81 billion by 2029.

In academia, this interest catalyzed significant efforts to develop software and hardware solutions for TinyML and contributed to the increasing interest around RISC-V Instruction Set Architecture (ISA). Among other works, Flamand et al. [21] proposed a custom MCU (GAP8) featuring SIMD to accelerate the computation of Quantized Neural Networks (QNNs), competing with the already established Arm Cortex-M family. This MCU became so relevant that GreenWaves Technologies started mass production and even proposed a newer iteration (GAP9) [22]. Garofalo et al. [19] contributed significantly to the interest around these MCUs by proposing an ML API (PULP-NN) to ease the deployment of int-8 QNNs on these devices. Scherer et al. [23] developed a RISC-V hardware accelerator capable of performing 3,200 inferences per second with an energy consumption of just 12.2 milliwatts.

TinyML has found applications in a wide array of domains. A collaborative effort between more than 30 entities from both industry and academia [24] highlighted the importance of TinyML in domains that include (i) word spotting, (ii) object recognition, (iii) object counting, and (iv) industry monitoring and control. Nevertheless, more recently, the use of TinyML extended [25–29] to (i) healthcare, (ii) smart agriculture, (iii) wildfire detection, and (iv) vehicular services, among others.

1.1.3 Proprietary Machine Learning in Untrusted Edge Devices

While moving ML models to the deep edge enhances user privacy and ensures predictable inference latency, it also introduces significant risks to the intellectual property of model owners, as these models are often deployed on untrusted devices. In effect, developing accurate and efficient ML models requires years of expertise and access to large-scale datasets, resources that are often unavailable to most ML users. Therefore, this task typically falls to big data companies, which have been selling their models in the form of Machine Learning as a Service (MLaaS) following a cloud-centered architecture [13, 16]. With the ongoing shift of ML to the edge, these Service Providers (SPs) are now concerned about deploying proprietary ML models in untrusted devices. Unauthorized access to these models can lead to the exposure of proprietary algorithms by reverse engineering [33] or even the leakage of training data [33–35] through the well-known Membership Inference Attacks (MIAs). Since training data is often provided to SPs under strict confidentiality agreements, such breaches risk not only violating user privacy but also resulting in legal repercussions for SPs.

Trusted Execution Environments (TEEs) have gained prominence as a primary solution to protect the intellectual property of ML models in untrusted devices [9, 11–13, 15–17, 36–41]. The widespread availability of Commercial Off-the-Shelf (COTS) TEEs in APUs, such as those based on Arm Cortex-A and x86 architectures, has made them the preferred choice for protecting the intellectual property of ML models on edge devices. However, relying on TEE-based protection presents several challenges. First, when securing computations within TEEs, it is crucial to minimize the Trusted Computing Base (TCB) to reduce the attack surface and enhance security. This requirement, however, is at odds with the memory-intensive nature of ML computations, which demand substantial resources. Second, integrating ML accelerators - such as GPUs or TPUs - with COTS TEEs, namely Intel-SGX and Arm TrustZone, requires careful design of the interface to balance security and performance effectively. Although initial research has explored the extension of TEEs to Heterogeneous Systems (xPUs), this area remains in its infancy, with many challenges yet to be addressed.

Over the last six years, several TEE-based approaches to protect the intellectual property of ML models have been proposed. To reduce the TCB, most works [9, 12, 13, 15] propose to save the whole ANN in non-secure memory as cyphertext, being the model decrypted and executed layer-by-layer within the TEE. Despite its popularity, this approach induces frequent cryptography operations and demands two context switches per ANN layer, inherently impacting the inference latency. To further optimize ML computation within TEEs, some works have proposed to (i) perform model loading and prediction in parallel [36]; (ii) extend TEEs [17, 37], (iii) execute only critical layers within the secure-world [11, 38–40]; (iv) obfuscate the most relevant ANN weights [16]; or (v) refactor a model in a bident structure [41].

Regardless of significant advancements, prior works focused exclusively on x86 and Arm Cortex-A processors. None have addressed Arm Cortex-M MCUs, which power the majority of Internet of Things (IoT) and TinyML systems. With the introduction of TrustZone-M technology in the recent Armv8-M ISA [42, 43], it is reasonable to question the feasibility of adapting previous works to Cortex-M devices. However,

Arm TrustZone-M differs substantially from Arm TrustZone and Intel-SGX technologies used in Cortex-A and x86 processors [42–45], presenting unique application design requirements. Moreover, Arm Cortex-M MCUs offer significantly lower computational throughput and limited memory, ranging from kilobytes to just a few megabytes, compared to the gigabyte-scale memory of Cortex-A and x86 systems. These resource constraints are compounded by the lack of GPU support, a feature commonly leveraged in prior works to mitigate the overhead of cryptographic operations and context switches between the Rich Execution Environment (REE) and the TEE.

1.1.4 Integrity of the Decision: Adversarial Attacks

Although a TEE-based approach provides the foundation for enabling trustworthy ML execution, by protecting the privacy of neural networks, a TEE *per se* is not sufficient to protect the integrity of the decision process. The integrity of the decision is compromised if the system unknowingly renders decisions that the attacker wants. With effect, previous research has shown that full-precision ANNs are vulnerable to adversarial examples [46–48] - inputs carefully crafted to fool the neural network into making an obvious misclassification. Although TEEs can prevent direct white-box attacks, adversarial attacks have also proven successful in gray- and black-box scenarios, where the attacker lacks access to the internals of the neural network and can only manipulate the inputs and observe the output behavior.

The susceptibility of full-precision ANNs to adversarial attacks has been extensively studied, with researchers engaged in an ongoing arms race of attacks and defenses. However, little is known about the effect of quantization on adversarial attacks. From the myriad of techniques used to speed up inference and reduce the memory footprint of ML in TinyML devices, quantization gained supremacy with the introduction of SIMD to accelerate integer-precision matrix multiplication in Arm Cortex-M [18] and RISC-V PULP MCUs [21]. Furthermore, quantization proved to have negligible accuracy degradation.

Although some research exists on the adversarial robustness of QNNs [8, 14, 49–52], the findings are scattered and lack a consistent basis for ground comparison. While some studies argue that QNNs are somewhat more robust to adversarial examples than full-precision ANNs [51], others state the opposite [50]. A common shortfall in these works is the absence of a comprehensive discussion on the scope of their findings. For instance, GPUs operate better with integer precision than with floating-point. Therefore, QNNs are deployed not only on ultra-low-power MCUs but also frequently on powerful GPUs. As GPUs have far more memory and computing power than ultra-low-power MCUs, QNNs deployed on these devices show significant differences in the base architecture, kernel filter size, network depth, and, above all, quantization policy. Consequently, providing a clear picture of the quantization policy and the target computing devices when evaluating QNNs is fundamental.

To the best of our knowledge, a comprehensive evaluation of the adversarial robustness of QNNs tailored for the TinyML scenario remains an open research challenge. Moreover, existing research on defense mechanisms for QNNs is still in its early stages, and their effectiveness and applicability within the constraints of the TinyML context remain unexplored.

1.1.5 Integrity of the Decision: Object Deformation

Due to the scarce computational power and memory resources in TinyML systems, the ML models deployed in these environments typically fall to small neural networks with classic dense and convolutional layers. With effect, the two QNNs representing image processing tasks within the TinyML Perf benchmark [24] - the most prevalent and relevant benchmark for TinyML hardware, resulting from the collaborative effort of more than 30 entities from industry and academia - are convolution-based.

Convolutional layers excel at extracting critical visual features from images, such as edges and shapes [53–55]. However, Convolutional Neural Networks (CNNs) and derivatives exhibit notable limitations. In addition to their susceptibility to adversarial examples, CNNs struggle to capture texture details, positional attributes, and spatial relationships between object components. The inability to model these properties renders them vulnerable to structural distortions, such as object deformations [53–55]. With effect, a CNN may misclassify an image with a nose and two eyes positioned below the mouth as a human face. To reduce the dimensionality of feature maps and consequently, the size of convolution kernels, convolutional layers are typically paired with pooling layers. Pooling layers extract the most salient features from localized regions of the feature map generated by a convolution and propagate these features to subsequent layers. However, this process comes at the expense of reduced capacity to learn pose and object arrangement as the relative spatial information between features is lost across the network [53–55]. Although several defenses against adversarial examples have been proposed [56–61], these techniques address pixel-level perturbations and fail to mitigate vulnerabilities to structural transformations, such as object deformations.

This vulnerability of CNNs motivated the development of a novel class of deep learning architectures known as Capsule Networks (CapsNets) [53]. The core unit of a CapsNet is a capsule, which consists of a group of neurons structured as vectors. These vectors encapsulate both posture information (e.g., position and orientation) and other instantiation parameters (e.g., width and height), collectively representing an object or its constituent parts [53]. With effect, each vector dimension represents a different object parameter and the length of the vector represents the probability of the image containing the object. A key feature of CapsNets is their reliance on spatial agreement among capsules in lower layers to identify objects. For an object to be detected, capsules must align in their spatial configuration, thereby capturing the relationships and hierarchies between object parts more effectively than CNNs [53].

Although CapsNets have been promoted as a groundbreaking advancement in ML for image processing tasks [54], CapsNets are still computationally intensive and impose high memory requirements [62–68]. In an era where ML is shifting to the deep edge, the feasibility of CapsNets remains mainly confined to high-performance cloud servers. While integer-precision computation has already proven to be a viable solution for accelerating ANNs and CNNs, the feasibility of deploying CapsNets under these constraints remains unexplored. Unlocking the deployment of CapsNets in resource-constrained MCUs would enable TinyML systems to overcome the object deformation vulnerability of CNNs, thereby unlocking new opportunities for TinyML in mission-critical applications requiring advanced image processing.

1.2 Contributions

1.2.1 Research Challenges

RC1. Need for solutions to protect the intellectual property of ML models in MCUs. The shift of ML to edge devices poses significant challenges to protecting the intellectual property of ML models, particularly in untrusted user devices. TEEs on COTS processors have emerged as a potential solution for protecting the intellectual property of ML models. Still, current TEE-based approaches focus on high-performance processors such as Arm Cortex-A [9, 11, 12, 15, 17, 36, 39–41] and x86 [13, 16, 37, 38] architectures. With MCUs dominating the IoT market [32] and the increasing interest in TinyML systems powered by ultra-low-power MCUs [30, 31], namely Arm Cortex-M, rises the need for solutions to protect the intellectual property of ML models deployed in this class of devices. However, the constraints of Arm TrustZone-M [42, 43], including its fundamental differences from TrustZone-A and Intel-SGX, and the limited computational capabilities and memory of Cortex-M MCUs, which feature a few megabytes at best, make it unclear how to implement existing methods on these devices effectively. Furthermore, the absence of GPU hardware acceleration for ML workloads exacerbates performance and latency concerns.

RC2. Scattered information on QNNs vulnerability to adversarial examples. With the adoption of SIMD in Arm Cortex-M [18] and RISC-V PULP [19, 21, 22] MCUs to accelerate matrix multiplication and convolution with integer operands, quantization has become the de-facto strategy to deploy ML in TinyML systems. Although quantization has already proved to have a negligible impact on accuracy, existing studies on the adversarial robustness of QNNs offer inconsistent and fragmented insights. Some argue that QNNs exhibit greater resilience to adversarial attacks [51], while others report increased vulnerability [50]. These contradictions are a consequence of the diverse strategies/policies available to quantize neural networks, which differ greatly, but not only, with the computational power of the device they target. Quantization is used not only to speed up inference on GPUs and save memory on APUs but also to enable ML in MCUs. A comprehensive evaluation of the adversarial robustness of QNNs tailored for TinyML systems powered by Arm Cortex-M MCUs and following the quantization policy compatible with the state-of-the-art APIs developed or backed by Arm (CMSIS-NN and TensorFlow Lite Micro) remains an open challenge. Furthermore, it is yet to be explored whether existing defense mechanisms developed for full-precision ANNs can be adapted to improve the resilience of QNNs in TinyML systems or whether novel defenses must be proposed.

RC3. Make TinyML robust against object deformation. CNNs are widely used for image-based tasks in TinyML due to their proficiency in extracting visual features, but face significant limitations. They are not only vulnerable to adversarial examples but also fail to capture essential spatial relationships, positional attributes, and texture details, making them susceptible to structural transformations like object deformations [53–55]. This vulnerability severely restricts their applicability in mission-critical image

processing tasks, where robustness to such transformations is essential. CapsNets were introduced to address these issues by representing objects and their parts as vectors that encode both posture and instantiation parameters (e.g., position, orientation). CapsNets rely on spatial agreement between capsules in lower layers to effectively capture hierarchical relationships, improving robustness to object deformation [53–55]. Despite their promise, the high computational power and memory required by CapsNets have limited their deployment to high-performance systems. While fixed-point computation and quantization have been used to accelerate traditional neural networks on TinyML systems, the feasibility of adapting CapsNets to Arm Cortex-M or RISC-V PULP MCUs remains unexplored.

1.2.2 Research Questions

In this PhD thesis, we address key challenges concerning the privacy of ML models and the integrity of ML decisions in TinyML systems powered by ultra-low-power MCUs. More specifically, we address three critical challenges hindering the development of secure and reliable TinyML systems: (i) the lack of frameworks to protect the intellectual property of third-party QNNs in ultra-low-power MCUs; (ii) the lack of understanding regarding the vulnerability of QNNs tailored for ultra-low-power MCUs to adversarial examples; and (iii) the vulnerability of TinyML systems applied in image processing tasks to object deformation. From these challenges, we formulated five pivotal research questions that define the scope of the research work performed in this thesis:

RQ1. *Can TEEs be leveraged to protect the privacy of QNNs on Arm Cortex-M MCUs?*

RQ2. *How does quantization affect a neural network defense against adversarial examples?*

RQ3. *Do defenses for full-precision ANNs still work for QNNs deployed on resource-constrained MCUs?*

RQ4. *Can CapsNets be quantized with negligible impact on accuracy?*

RQ5. *Is it feasible to deploy CapsNets on resource-constrained Arm Cortex-M and RISC-V PULP MCUs?*

To address **RQ1**, we conducted a comprehensive literature review of existing TEE-based approaches to protect the privacy of ML models in Arm Cortex-A and x86 processors. Building on this analysis, we proposed, designed, and implemented SecureQNN - the first publicly available open-source framework aimed at increasing the protection of the intellectual property of QNNs deployed in Arm Cortex-M MCUs. SecureQNN introduces an innovative method for identifying privacy-critical layers within a QNN, effectively reducing the computational and memory overhead within the TEE.

To answer **RQ2**, we performed the most comprehensive empirical evaluation of adversarial robustness of QNNs targeting TinyML applications. This evaluation assessed the robustness of QNNs - without applying any defense - against ten widely recognized adversarial attacks and compared their performance against full-precision ANNs. As part of this investigation, we analyzed QNNs under the int-8 and int-16

quantization settings used by CMSIS-NN and TensorFlow Lite Micro. To answer **RQ3**, we tested six popular defense mechanisms against the attacks that had the greatest impact on QNN accuracy in our initial robustness testing.

Finally, to tackle **RQ4**, we developed the first open-source framework for quantizing CapsNets from floating-point precision to int-8, ensuring compatibility with the quantization policies of Arm CMSIS-NN and RISC-V PULP-NN libraries. Alongside this framework, we developed the first open-source library enabling the deployment of CapsNets on Arm Cortex-M and RISC-V (RV32IMCXpulp) MCUs, answering **RQ5**.

1.2.3 Main Contributions

The diagram in Figure 1 links all the scientific contributions made during this PhD thesis to the research questions and research challenges that each work aims to address. There are three primary contributions [69–71], which are highlighted in dark green, and transcribed in Chapter 5, Chapter 6, and Chapter 7, respectively. Secondary contributions are highlighted in yellow and include papers that were authored [72] and co-authored [73] but excluded from this document. We will next discuss the primary contributions of this PhD thesis, summarizing their objectives, individual contributions, and key findings and insights.

SecureQNN: Shielding the Intellectual Property of QNNs in TinyML Systems. Given the lack of solutions to protect the intellectual property of ML models in TinyML systems, namely on those powered

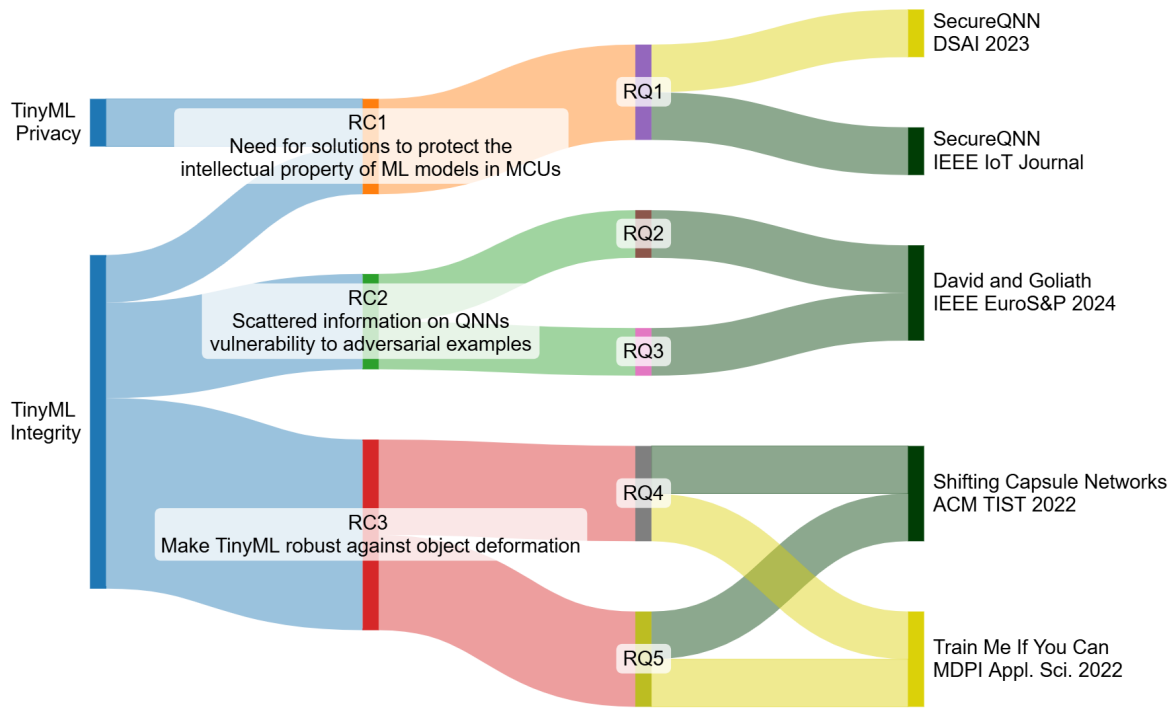


Figure 1: Relationship between research challenges, research questions, and scientific contributions.

by Arm Cortex-M MCUs, we propose SecureQNN. SecureQNN is an open-source framework that simulates a powerful attack scenario to determine the most privacy-critical layers of a QNN. Our framework considers that a potential attacker has access to everything deployed outside the TEE, including the Operating System (OS), and is aware of the architecture of the private layers. The attacker does not have access to the training dataset, but can spoof sensors or network connections to collect inference data and apply deep generative ML to build a surrogate dataset. In this scenario, attackers may freeze public layers and train a substitute QNN until its accuracy matches or surpasses that of the original model. SecureQNN assumes that the intellectual property of a QNN is sufficiently secure when the training effort of the attacker - measured in training epochs - is at least as high as the effort required to train the original model from scratch. Ultimately, SecureQNN aims at reducing the cost-benefit and likelihood of successful model replication by third parties. For this purpose, SecureQNN isolates the most privacy-critical layers of the target QNN in the secure-world of Arm TrustZone-M. For enhanced security, SecureQNN leverages Trusted Firmware-M (TF-M). TF-M provides secure boot functionality, which prevents tampering with the firmware image and, therefore, with the QNN itself (private and public layers), and cryptographic capabilities, which protect privacy-critical layers from highly sophisticated flash probing attacks. Experiments demonstrate that SecureQNN can undermine the cost-benefit of unauthorized QNN replication by isolating 51%-65% of the model size while adding only 0.06% to the overall decision latency. An earlier version of this paper was published in the form of a conference perspective paper at the 2023 Data Science and Artificial Intelligence [72]. The final and full version is submitted to the IEEE Internet of Things Journal [69] and is a collaboration with Tiago Gomes and Sandro Pinto.

David and Goliath: An Empirical Evaluation of Attacks and Defenses for QNNs at the Deep Edge. Although SecureQNN protects the integrity of QNNs, by preventing tampering with internal parameters, the decision integrity can still be subverted through adversarial examples. Given the scattered information on the effects of quantization against adversarial examples, we conducted the most comprehensive empirical evaluation on the adversarial robustness of QNNs targeting TinyML applications. In particular, we assessed how well adversarial attacks and defenses, originally developed for full-precision ANNs, perform on QNNs tailored for TinyML environments. Our evaluation suite includes three QNNs, ten attacks, and six defenses. With this study, we draw a set of interesting findings. First, the average point distance to the decision boundary increases as the neural network precision decreases. Second, quantization makes it more difficult for adversarial attacks to optimize over the loss surface of neural networks, often resulting in the explosion or vanishing of the estimated gradient. Third, we observe poor transferability of adversarial examples from full-precision ANNs to QNNs, a consequence of two key factors: (i) gradient misalignment and (ii) quantization-shift. Our investigation into defense mechanisms revealed that input pre-processing defenses are particularly effective at mitigating small perturbations but struggle to handle larger adversarial attacks. Train-based defenses, on the other hand, generally increase the average point distance to the decision boundary, a behavior that persists after quantization. Train-based defenses

are especially appealing for TinyML applications because they impose no additional computational demands during inference, making them highly portable for resource-constrained systems. We envision that developing train-based defenses capable of smoothing the effects of quantization-shift and gradient misalignment is a promising approach to improve the robustness of QNNs in TinyML systems. This work was performed with the support of Sandro Pinto and resulted in a paper [70] published and presented at the 2024 European Symposium on Security and Privacy (EuroS&P). The source code is available open-source.

Shifting Capsule Networks from the Cloud to the Deep Edge. The integrity of ML-based decisions in image applications can be compromised at the pixel-level, via adversarial examples, but also via object deformation. CapsNets were introduced to address this vulnerability; however, their feasibility on TinyML systems remains unknown. To address this challenge, we propose new software kernels extending the CMSIS-NN and PULP-NN APIs to support the deployment of CapsNets on Arm Cortex-M and RISC-V (RV32IMCXpulp) MCUs, respectively. The extended libraries feature new core functions to support all capsule operations in CapsNets with Dynamic Routing, using 8-bit integers as operands. Additionally, we propose a framework to perform post-training quantization of CapsNets built in TensorFlow. Experimental results show that our quantization framework reduces the memory footprint of CapsNets by nearly 75%, with negligible impact on accuracy - showing a maximum accuracy loss of 0.18%. For Arm Cortex-M MCUs, results extracted for the STM32L4R5ZIT6U board (Arm Cortex-M4 @ 120 MHz) suggest that our API enables the execution of primary capsule and capsule layers with medium-sized kernels in just 119.94 and 90.60 milliseconds, respectively. For RISC-V MCUs, results extracted for the Gapuino v1 board (octa-core RV32IMCXpulp @ 170 MHz) show even lower latency results for the same medium-sized primary capsule and capsule layers: 7.02 and 38.03 milliseconds, respectively. The quantization framework and the software kernels extending CMSIS-NN and PULP-NN APIs are available open-source. This work was performed in collaboration with Diogo Costa and received the support of Tiago Gomes and Sandro Pinto. It was published in 2022 in the ACM Transactions on Intelligent Systems and Technology journal.

1.2.4 Other Contributions

During this PhD program, we had the opportunity to collaborate with a master's student on a project related to the core of our research. Below, we summarize the objectives and key findings of this project.

SecureQNN: Introducing a Privacy-Preserving Framework for QNNs at the Deep Edge. Before developing a framework to protect the intellectual property of QNNs in resource-constrained MCUs (SecureQNN), we performed an extensive review of existing works. While no prior work was identified that specifically targets MCUs, some solutions have been proposed for more capable platforms, such as those based on Arm Cortex-A and x86 architectures. In this paper [72], we provide a comprehensive analysis of existing approaches, highlighting the challenges and limitations associated with adapting or porting them to MCUs. Building on insights from that study, we proposed an initial architectural design for SecureQNN

and established baseline reference values for inference latency. This work, conducted in collaboration with Tiago Gomes, Jorge Cabral, João Monteiro, Adriano Tavares, and Sandro Pinto, was published and presented at the 2023 Data Science and Artificial Intelligence conference [72].

Train Me If You Can: Decentralized Learning on the Deep Edge. While the inference pass of ANNs is already executed successfully on Arm Cortex-M MCUs, the training remains confined to the cloud. This scenario not only imposes the transaction of high volumes of private data over a network but may also lead to unpredictable delays when ML applications attempt to adapt to adversarial environments. In this co-authored paper [73], we explore the feasibility of performing ANN training directly on Arm Cortex-M MCUs for the first time. To this end, we present L-Stochastic Gradient Descent (SGD), a streamlined adaptation of SGD, designed to optimize speed and minimize memory usage on resource-constrained Cortex-M MCUs. Two variations of L-SGD were developed: one utilizing 32-bit floating-point weights (L-SGD float-32) and the other employing quantized 8-bit weights with 16-bit gradients (L-SGD int-8). Results show that for a fully-connected ANN, L-SGD (float-32) is 4.20× faster than SGD while requiring only 2.80% of the memory with negligible accuracy loss. On the other hand, L-SGD (int-8) proves unsuitable for training an ANN from scratch due to the high memory peaks and increased overhead to mitigate overflow issues. Nevertheless, L-SGD (int-8) is a viable solution for minor model updates. This paper, co-authored with Diogo Costa and Sandro Pinto, was published in 2022 in the Applied Sciences journal.

1.3 Thesis Outline

This thesis is structured into two primary parts. The first part, comprising Chapters 2, 3, and 4, provides a comprehensive overview of the research topics addressed in this work. Chapter 2 introduces foundational concepts necessary for understanding the subsequent contributions of the thesis. Specifically, it discusses topics related to MCU-based systems (Section 2.1) and TEEs (Section 2.2). Section 2.3 discusses quantization in the context of ML and Section 2.4 discusses adversarial attacks and defenses. Chapter 2 ends with a review of the most relevant CapsNets architectures proposed to date (Section 2.5). Chapter 3 reviews state-of-the-art works in areas relevant to this thesis, including the privacy of ML models (Section 3.1), the robustness of QNNs against adversarial examples (Section 3.2), and hardware and software solutions for accelerating the computation of CapsNets (Section 3.3). Chapter 4 concludes the first part of the thesis by synthesizing key insights and identifying potential directions for future research.

The second part of the thesis, encompassing Chapters 5, 6, and 7, presents the main contributions of this PhD, detailed by full transcription of the publications that form the core of this work. Chapter 5 describes the development of SecureQNN, a TEE-based framework designed to protect the intellectual property of QNNs deployed on Arm Cortex-M MCUs. Chapter 6 presents an empirical evaluation of the adversarial robustness of QNNs and possible defense mechanisms, while Chapter 7 introduces a framework enabling the deployment of CapsNets on Arm Cortex-M and RISC-V PULP MCUs.

2 Background

This chapter provides a comprehensive overview of the concepts and technologies essential for understanding the work presented in this thesis. It begins with an introduction to MCU systems (Section 2.1), focusing on Arm Cortex-M and RISC-V RV32IMCXpulp in the context of ML and their respective state-of-the-art ML APIs. This is followed by a discussion of TEEs, including an analysis of Arm TrustZone technology and its progression to Arm TrustZone-M (Section 2.2). Section 2.3 delves into quantization as a key mechanism for TinyML systems. Subsequently, in Section 2.4 we review a wide selection of adversarial attacks and defenses with great impact and citation in the research community and whose implementation is available open-source. Lastly, Section 2.5 delves into the theoretical foundations of CapsNets and explores their main variations.

2.1 Microcontroller Systems

What is an MCU? An MCU is an extremely small computer, self-contained in a single chip, and tailored for specific tasks within embedded systems [74, 75]. Its architecture is optimized for real-time operations, prioritizing reduced cost, minimal silicon area, and low power consumption. MCUs are typically characterized by their simplicity, incorporating a single Central Processing Unit (CPU) with a shallow pipeline structure and two types of memory: a flash memory for firmware storage, and a Static Random Access Memory (SRAM) [74, 75]. Both memory types are highly constrained, with capacities ranging from some hundred kilobytes to a few megabytes, in stark contrast to the gigabyte-scale memories commonly found in APUs, as depicted in Figure 2 [76]. MCUs execute instructions directly from physical memory and do not feature a Memory Management Unit (MMU) for virtual memory support, making them incapable of running Unix-like OSs [74, 75]. Additionally, MCUs typically feature a wide range of input/output peripherals such as Analog-to-Digital Converters (ADCs), timers, and communication interfaces like Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI), and Universal Asynchronous Receiver-Transmitter (UART) [74, 75]. However, they do not support GPU acceleration. The most advanced MCUs gained support for TEEs, allowing developers to establish two execution environments: secure and normal [42, 43].

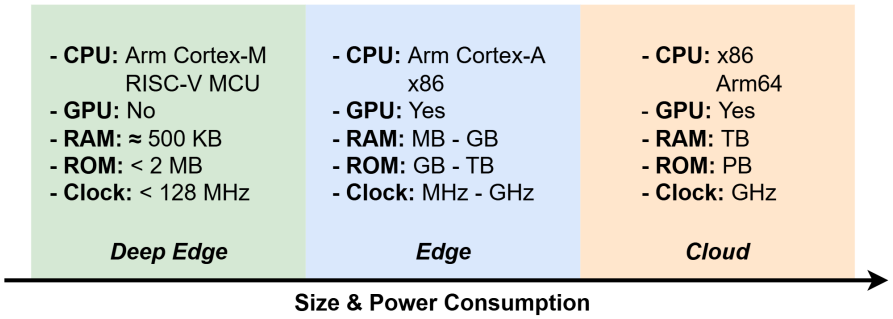


Figure 2: Comparison of deep edge, edge, and cloud computation [76].

MCUs in the Market. In 2022, approximately 250 billion MCUs were estimated to be in use globally [77], with projections from Arm suggesting this figure could surpass 1 trillion across various market segments by 2035 [18]. According to Grand View Research [78], the global MCU market was valued at approximately USD 32.37 billion in 2023 and is expected to grow at a Compound Annual Growth Rate (CAGR) of 11.7% from 2024 to 2030. The most relevant MCUs operate at 8-bit, 16-bit, 32-bit, and 64-bit. Nevertheless, 32-bit MCUs accounted for over 50% of the market share in 2023 and are projected to grow at a CAGR of 12.3% until 2030 [78]. Among the market leaders by MCU architecture, which includes Arm Cortex-M, Microchip PIC, Atmel AVR, Intel 8051, and Infineon TriCore, Arm was the leading player in 2023, capturing a revenue share of over 47.0% [78].

Applications of MCU-based ML (TinyML). MCUs are a key technology in several application domains and are now ubiquitous in areas such as healthcare, smart homes, agriculture, transportation, and maintenance systems [25, 27–29]. The recent integration of ML into these devices, leading to the emergence of the TinyML concept [24], has further expanded MCU's scope and market relevance. In the healthcare sector, the combination of low-power MCUs, such as Arm Cortex-M, with highly pruned, low-bit-width neural networks has enabled the development of innovative devices for the detection and recovery of freezing of gait in patients with Parkinson's disease [79], epilepsy treatment [80], diagnosis of focal liver lesions [81], among others [27]. In the domain of smart homes, MCU-based TinyML systems have been utilized in applications such as keyword spotting [82] and gas leakage monitoring [83]. Within transportation systems, TinyML has facilitated adaptive traffic control [84] and driver fatigue monitoring [85]. Furthermore, TinyML has found extensive use in Unmanned Aerial Vehicles (UAVs) for tasks such as infrastructure inspection [86] and in smart farming [87, 88], where applications include soil moisture prediction, optimal water usage, and plant disease detection.

2.1.1 Arm Cortex-M

The Arm Cortex-M family dominates the MCU market and Arm has developed a wide range of ISAs - Armv6-M to Armv8.1-M - to meet diverse requirements. Among these, the most recent architectures - Armv7E-M

(Cortex-M4 and Cortex-M7) [89], Armv8-M (Cortex-M33, and Cortex-M35P) [90], and Armv8.1-M (Cortex-M55 and Cortex-M85) [91] - feature mechanisms to accelerate the computation of ML. With effect, Arm Cortex-M4 MCUs [92] were the first in the series to feature a Digital Signal Processing (DSP) extension with support for SIMD instructions to accelerate integer precision ANNs, also referred to as QNNs.

The DSP extension introduced SIMD instructions for integer-precision Multiply-and-Accumulate (MAC) operations, extensively used in the computation of fully-connected and convolutional layers. These instructions support 16-bit data operands, enabling the acceleration of QNNs with int-16 weights and activations through single-cycle MAC instructions [89]. The acceleration also extends to int-8 QNNs as Armv7E-M ISA features seamless instructions (*SXTB16, PKHTB, PKHBT*) to expand 8-bit data into 16-bit [18, 89]. Table 1 summarizes some of the most relevant DSP instructions available in the Armv7E-M and newer ISAs for accelerating the computation of int-16 and int-8 QNNs [18, 89, 90].

The Armv8.1-M architecture introduced a significant advancement in ML computation through Arm Helium technology, which employs the M-Profile Vector Extension (MVE) instructions [91]. Helium enables SIMD vector processing to enhance parallelism, allowing for up to 4x16-bit and 8x8-bit MAC operations per clock-cycle [93]. Despite these advancements, this PhD thesis focuses on the Armv7E-M and Armv8-M architectures, as these remain significantly more prevalent at the time of this research.

Instruction Type	**Functionality**	**Example Instructions**
Saturation Arithmetic	Adds/Subtracts two integers and saturates the result to avoid overflow	QADD QSUB
Parallel Arithmetic	Adds/Subtracts 8-bit or 16-bit integers in parallel	UHADD8; UHSUB8 SHADD8; SHSUB8
MAC	Performs two 16x16-bit multiplications and accumulates the result	SMLAD; SMLALD; SMLABB SMLABT; SMLATB; SMLATT
Packing and Unpacking	Data preparation and alignment for arithmetic and MAC operations	SXTB16; PKHBT; PKHTB SSAT; USAT
Load/Store with Alignment	Vectorized memory access and dual-load/store	VLDM; VSTM LDRD; STRD

Table 1: SIMD of Armv7E-M and Armv8-M for int-16 and int-8 QNNs.

2.1.1.1 CMSIS-NN

The CMSIS-NN library [18] is a series of open-source software kernels developed by Arm to facilitate the deployment of QNNs on Arm Cortex-M MCUs. These software kernels are built upon the CMSIS abstraction layer and are tuned for Cortex-M MCUs featuring DSP extension and/or Helium technology. In MCUs featuring these technologies, CMSIS-NN abstracts away the extensive set of SIMD instructions required to execute a QNN. In the less capable Arm MCUs, namely on Cortex-M0 (Armv6-M) and Cortex-M3 (Armv7-M), the computation runs as pure *C* code. CMSIS-NN is structured into two primary components: *NNFunctions* and *NNSupportFunctions*. The *NNFunctions* module implements the standard neural network layers, namely convolutional and fully-connected layers. The *NNSupportFunctions* module offers

auxiliary functionality for tasks such as data precision conversion and activation functions. CMSIS-NN is also the backend of TensorFlow Lite Micro, an API introduced by Google for Arm Cortex-M MCUs.

CMSIS-NN accommodates software kernels to support both int-8 and int-16 data formats. The quantization policy relies on a power-of-two scaling, meaning that a given floating-point value is represented as $A * 2^n$, where A is the integer value and n is the number of bits representing the fractional part. Additionally, the quantization policy enforced by CMSIS-NN is uniform, symmetric, static, and implies a fixed bit-width for the whole QNN. Regarding granularity, the original ANN must be quantized layer-by-layer. All these constraints combined enable CMSIS-NN to simplify the scaling of a layer output, which is implemented as a simple bitwise shift operation.

For convolutional layers, input data is assumed to be in Height-Width-Channel (HWC) format, i.e., channel first. Regarding activation functions, CMSIS-NN supports Rectified Linear Unit (ReLU), softmax, sigmoid, and hyperbolic tangent functions [18]. ReLU is implemented as a straightforward clipping operation that sets negative values to zero while retaining positive values. Meanwhile, the sigmoid and hyperbolic tangent functions are implemented as Lookup Tables (LUTs) for efficient computation.

2.1.2 RISC-V RV32IMCXpulp

RISC-V is an open-source ISA "used to develop custom processors for a variety of applications, from embedded designs to supercomputers" [94]. The RV32IMCXpulp is a specific RISC-V extension tailored to accelerate the computation of QNNs while maintaining low power consumption [19]. RV32IMCXpulp builds on top of *RV32I* - the base 32-bit version of the RISC-V ISA. RV32IMC incorporates the *M* standard extension for integer multiplication and division, and the *C* standard extension for compressed 16-bit instructions, i.e., intermix 16-bit instructions with the 32-bit base instructions [95]. The *Xpulp* extension is based on the PULP cluster [96] and adds a DSP featuring SIMD vectorial instructions for QNN acceleration on RISC-V-based MCUs [19].

The *Xpulp* extension supports two vectorial data types: *v4s* and *v2s*. The former allows to fill a 32-bit register with four int-8 data values in two clock cycles using the *pack4* instruction. The latter allows the filling of a 32-bit register in a single clock-cycle with two int-16 data values. The MAC operation is done via two main instructions, *sdopt4* and *sdopt2*, both executed in a single clock-cycle. *sdopt4* takes four int-8 operands, while *sdopt2* takes two int-16 operands. Table 2 summarizes the most relevant SIMD vectorial instructions of *Xpulp* for QNN computation.

2.1.2.1 PULP-NN

PULP-NN [19] is an open-source API to optimize the deployment of QNNs on RISC-V RV32IMCXpulp-based MCUs. Drawing inspiration from CMSIS-NN, it is divided into two primary modules: (i) *NNKernels*, which implements the standard fully-connected and convolutional layers, and (ii) *NNUtils*, which provides auxiliary functions for tasks like data precision conversion and batch normalization. In contrast to CMSIS-NN, which supports int-16 and int-8 data, PULP-NN supports byte (int-8) and sub-bye data types, namely

Instruction Type	Functionality	Example Instructions
Saturation	Clips unsigned/signed values to a given number of bits	clipu_r; clip_r
Statistics	Calculates the maximum/average of 4 int-8 values Calculates the maximum/average of 2 int-16 values	maxs4; maxu4; avgs4; avgu4 maxs2; maxu2; avgs2; avgu2
MAC	Performs four 8x8-bit multiplications and accumulates the result Performs two 16x16-bit multiplications and accumulates the result	sdotsp4; sdotusp4 sdotsp2; sdotusp2
Packing and Unpacking	Data preparation and alignment for arithmetic and MAC operations	pack4 bextract; bextractu

Table 2: SIMD of RISC-V RV32IMCXpulp for int-16 and int-8 QNNs.

int-4, int-2, and int-1. Nevertheless, it must be considered that the SIMD vectorial instructions of RISC-V RV32IMCXpulp are still not able to accelerate the computation of QNNs with precisions lower than int-8. To take the most of the vectorial SIMD instructions of RV32IMCXpulp (Table 2) convolutional layers are computed using a $2x2$ sized matrix multiplication kernel. By its side, fully-connected layers rely on a $2x1$ sized kernel. For int-8 QNNs, PULP-NN adopts the same weight quantization strategy as CMSIS-NN but employs a distinct methodology for activation quantization. Specifically, it limits activations to non-negative values, thereby requiring an asymmetric quantization approach.

Similar to CMSIS-NN, PULP-NN considers data to be organized in the HWC format. PULP-NN was the first ML API to efficiently exploit parallel MCU architectures, as those available on GAP8 [21] and GAP9 [22] MCUs. In convolutional layers, computations can be distributed along the height, width, or channel dimensions of the output feature map. In fully-connected layers, the workload is split among output neurons.

2.2 Trusted Execution Environments

A TEE is a secure and trusted area within the CPU that ensures the confidentiality and integrity of data and applications. TEEs operate in isolation from the primary execution environment, being this isolation enforced via hardware integrated into the CPU architecture. Therefore, even if the primary execution environment is compromised, data and applications within the TEE are not. A typical TEE-based system incorporates two distinct domains: a trusted domain and an untrusted domain. The trusted domain includes a minimal TCB dedicated to executing critical applications. In contrast, the untrusted domain encompasses the feature-rich OS and general-purpose applications, which are executed without access to the memory and peripherals allocated to the trusted domain.

2.2.1 Arm TrustZone-(M)

Arm TrustZone, introduced in 2004, is a hardware-based security extension designed for Arm APUs, specifically the Arm Cortex-A series [42]. This technology adopts a system-wide approach to security, implementing hardware-enforced protection mechanisms at both the CPU and System on Chip (SoC)

levels. Initially implemented within the Armv6K architecture, TrustZone significantly advanced processor design by dividing the system into two isolated execution environments: the secure-world and the normal-world [42, 43]. The secure-world hosts privacy- and integrity-critical software components, while the normal-world is reserved for non-sensitive applications. The processor can only operate within one of these domains at a time, with its current state dictated by the Non-Secure (NS) bit located in the Secure Configuration Register (SCR) [42, 43]. The context switch between these two worlds is handled by a privileged software mechanism - Secure Monitor - which saves the state (registers, stack pointers, etc.) of the current world before switching to the other [42, 43]. TrustZone also extended the Cortex-A architecture with two optional entities for memory management [42]. The TrustZone Address Space Controller (TZASC) is configurable and sets a memory region on the on-chip memory as secure or non-secure. The TrustZone Memory Adapter (TZMA) implements a similar functionality but for off-chip memory. Applications within secure-world can access non-secure memory regions, but the opposite is not allowed.

Arm TrustZone-M. In 2016, TrustZone was adapted to the new generation of Arm MCUs, namely those based on Armv8-M and Armv8.1-M ISAs [43]. This newer technology, referred to as Arm TrustZone-M, retains the foundational dual-world architecture, which ensures that non-secure software cannot directly access secure data and applications. At the low level, there are significant architectural adjustments to align with the characteristics of the Arm Cortex-M MCUs. Unlike the Arm Cortex-A APUs, the security state in Armv8-M and Armv8.1-M does not rely on a dedicated security bit. Instead, the separation between secure and non-secure worlds is memory-based: the MCU operates in a secure state when executing from secure memory and in a normal state when running from non-secure memory [42, 43]. To enforce this security model, TrustZone-M introduces two key components: the Security Attribution Unit (SAU) and the Implementation-Defined Attribution Unit (IDAU) [42, 43, 45]. The SAU is programmable and enables dynamic partitioning of memory regions, while the IDAU defines the static partitions specified by the chip manufacturer. The security status of an address is determined through a *logical OR* operation combining the outputs of the SAU and IDAU. In contrast to Cortex-A, the Armv8-M, and Armv8.1-M ISAs obviate the need for Secure Monitor to perform the context switch between both worlds. Instead, context-switching mechanisms are integrated into the core logic, enabling significantly faster transitions, a critical requirement for real-time applications [42, 43]. Figure 3 depicts the classical dual-world architecture of a TEE assisted by TrustZone-M.

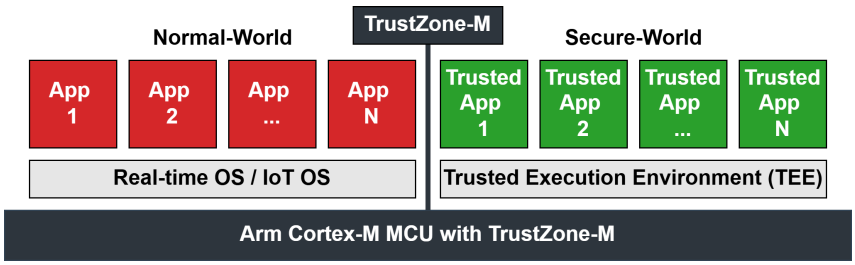


Figure 3: Classic architecture of a Trusted Execution Environment based on Arm TrustZone-M.

Trusted Firmware-M (TF-M). TF-M is an open-source reference implementation of a secure firmware for Arm Cortex-M MCUs featuring TrustZone-M. It is designed to align with the Arm Platform Security Architecture (PSA), "enabling chips, Real Time OSs and devices to become PSA Certified" [97]. In addition to the TF-M core, which handles software component isolation and communication, TF-M incorporates modules for secure boot and security services such as cryptography and protected storage.

TF-M adopts MCUboot as its reference implementation for a secure bootloader. The secure boot ensures the integrity of both secure and non-secure binaries by verifying their digital signatures. Anytime the signature is corrupted, the boot is aborted to prevent the execution of malicious code. The TF-M core enables isolation between applications in secure and normal worlds, but also between applications within the secure world. To partition the memory into secure and non-secure regions, TF-M relies on the SAU and IDAU units of TrustZone-M. To make partitions within the secure memory, i.e., divide the secure memory among different Trusted Applications (TAs), TF-M employs a proprietary secure memory protection unit.

2.3 Quantization in Machine Learning

In the context of ML, quantization is a neural network compression technique that maps floating-point parameters, namely weights, bias, and activations, to integer-precision parameters with reduced bit-width [98–102]. This process involves encoding three key components of a floating-point value into a single integer: (i) the sign, (ii) the integer part, and (iii) the fractional part. Quantized values are represented in binary in the form of $Qm.n$ notation, where m denotes the number of bits reserved for the integer part and n specifies the bits allocated for the fractional part [102].

ANNs are usually trained using float-32 or float-64 data, a precision proven not required during inference [101]. With effect, ANN quantization gained interest in the 2010s when Vanhoucke et al. [103] proved that it was possible to perform inference using int-8 data with negligible impact on accuracy. This was a turning point for ML computation, especially in MCUs, as operations with integer data require significantly less computation than those with floating-point numbers. In processor cores without a dedicated Floating-Point Unit (FPU), operations with floating-points are emulated in software, which inherently creates overhead [101, 102]. In addition, quantization significantly reduces the size of ANNs, which contributed not only to the deployment of ML in MCUs but also in mobile devices in general [98–100]. Quantization is also commonly used to speed up inference even in powerful GPUs as those operate much faster using integer data [98–100]. Over the last years, several quantization methodologies/policies have been proposed [98–102, 104]. However, the effectiveness of any quantization methodology is strongly influenced by the memory characteristics and ISA of the deployment platform. Consequently, these two aspects must guide the following design choices.

Post-Training Quantization vs. Quantization-Aware Training. In post-training quantization, the ANN is quantized strictly after training without fine-tuning. Under this quantization strategy, the clipping

ranges and the scaling factors are computed post-training using a small subset of the training data, known as the calibration data [98–102]. In contrast, quantization-aware training aims to lower the impact of quantization on accuracy by re-training the ANN. The re-training process simulates the low-precision of weights at inference in the forward and backward passes, being weights quantized after each gradient update. This strategy considers the quantization error as part of the loss returned by the loss function, which the optimization algorithm tries to minimize [98–102]. Clipping range and scaling factors are computed during the fine-tuning of the model. As expected, the advantage of post-training quantization comes at the cost of increased overhead during the training pass.

Uniform vs. Non-Uniform Quantization. In uniform quantization, the range of floating-point values is divided into intervals/steps of equal size, and each step is mapped to a different integer value [98–100, 102]. In non-uniform quantization, the quantization steps are unevenly sized (not equally spaced). Non-uniform quantization is designed to capture with higher resolution the regions of values where data is more dense with the intent of returning QNNs with higher accuracy [98–100, 102]. For this purpose, non-uniform quantization typically uses a logarithmic function to map floating-point to integer values [100], in opposition to uniform quantization, which relies on linear functions. Although non-uniform quantization reduces the quantization error, it also imposes significant computational overhead when implemented on general-purpose hardware, such as APUs, MCUs, and GPUs, occasionally resulting in inference latencies that exceed those of the original floating-point ANNs. With effect, operations with non-uniform quantization often rely on LUTs storing the operands for each operation [98–100, 102]. Consequently, uniform quantization is widely adopted as the standard approach for efficient and fast computation on resource-constrained devices [100, 102]. Equation 2.1 represents a uniform quantizer Q, where r represents the floating-point value to be quantized, S is a scaling factor, and Z is the zero-point or offset.

$$Q(r) = round(r/S) - Z \tag{2.1}$$

Symmetric vs. Asymmetric Quantization. The scaling factor and zero-point are critical components of any uniform quantizer. The scaling factor determines the quantization step size, dividing the range of floating-point values into uniformly spaced intervals [99–102]. Equation 2.2 defines the scaling factor S of a uniform quantizer [100], where $[x_{min}, x_{max}]$ represents the range of floating-point values to be quantized (clipping range), and b denotes the bit-width of the quantization, i.e., the number of bits representing the quantized values. The clipping range can either be symmetric around zero (where $|x_{min}| = |x_{max}|$) or asymmetric (where $|x_{min}| \neq |x_{max}|$) [99–102]. For asymmetric ranges, the zero-point is calculated as shown in Equation 2.3 and rounded to the nearest integer [99]. The zero-point serves as an integer offset in asymmetric quantization, ensuring that the value zero is quantized without error. Asymmetric quantization typically enables a tighter clipping range, which can reduce quantization error when the data distribution is not centered around zero [99–102]. This property makes it particularly suitable for quantizing activations, which often have non-negative values due to ReLU or softmax functions.

Nevertheless, symmetric quantization is more prevalent in practice due to the computational overhead associated with asymmetric quantization during inference [100–102]. A standard approach for determining x_{min} and x_{max} is to align them with the minimum and maximum values of the floating-point range to be quantized. However, to avoid losing quantization precision with outliers, some authors have proposed other strategies, such as the use of percentiles or the minimization of the Kullback-Leibler divergence between floating-point and quantized values [99, 100].

$$S = \frac{x_{max} - x_{min}}{2^b - 1} \tag{2.2}$$

$$z = -round(x_{min} * S) - 2^b - 1 \tag{2.3}$$

Quantization Granularity. Quantization granularity refers to the scope at which the scaling factor is computed for the weights and activations of an ANN [98–100, 102]. When determining the appropriate granularity, two metrics should be considered: (i) impact on model accuracy and (ii) computational cost [99]. At one extreme, computing a distinct scaling factor for each weight and activation has the minimum impact on model accuracy; however, this is obviously not feasible as it incurs significant computational overhead and memory requirements. At the other extreme, calculating a single scaling factor for the entire ANN minimizes computational cost but often leads to a noticeable degradation in accuracy [100, 102]. As a compromise, layer-wise or filter-wise quantization are widely adopted approaches, balancing computational efficiency with model accuracy [98, 102].

Fixed Bit-Width vs. Mixed Bit-Width. In a fixed bit-width configuration, a uniform bit-width is applied across the entire neural network [98, 100]. Conversely, a mixed bit-width setting adjusts the bit-width of individual layers or filters (depending on the granularity of quantization) to optimize (i) accuracy, (ii) speedup inference latency, and/or (iii) reduce memory footprint [98, 100]. In this scheme, layers or filters with less impact on the final output are assigned lower bit widths, while those with higher impact are allocated higher bit widths. Current techniques to find the optimal bit-width for each layer/filter can be grouped into four distinct groups [104]:

- **Gradient-Based Optimization:** Formulates the search as an optimization problem solved via variants of gradient descent.
- **Heuristic-Based Optimization:** Formulates the search as an optimization problem that computes a heuristic assessing the sensitivity/importance of the layers/filters.
- **Meta-Heuristic-Based Optimization:** Formulates the search as an optimization problem solved via differential search or evolutionary search algorithms.
- **Reinforcement Learning-Based Optimization:** Formulates the search as a reinforcement learning problem solved by maximizing the reward.

Static vs. Dynamic Quantization. In static quantization, the scaling factor and zero-point are determined offline before inference, using a small subset of the training data, and maintained static afterward. In contrast, dynamic quantization determines these parameters dynamically during the inference phase for each input sample [100, 102]. With dynamic quantization, weights and activations are stored in memory as low bit-width integers and are subsequently dequantized for neural network operations, such as matrix multiplication and convolution, which are performed using floating-point arithmetic. As the computation of a given layer finishes, weights and activations are re-quantized again to be stored in memory. Although this approach reduces the impact of quantization on model accuracy compared to static quantization, its overhead may render it unsuitable for low-end MCUs [100, 102]. Dynamic quantization is particularly advantageous in scenarios where the primary objective is to reduce the memory footprint of ANNs, such as in mobile devices.

2.4 Adversarial Examples

Definition and Cause. The susceptibility of ANNs to adversarial examples was first identified by Szegedy et al. [105] in 2014. Adversarial examples are intentionally crafted inputs intended to mislead an ANN into making an obvious misclassification. Considering image data as reference, the adversary's goal is to merge the original image with perturbations imperceptible to the human eye, but cause the model to misclassify the input (Figure 4). Over the last years, researchers have been identifying multiple causes for this vulnerability, exploiting them to propose both attacks and defenses. Among the identified causes, overfitting, inadequate regularization, and the linear behavior of ANNs in high-dimensional spaces, particularly when employing linear activation functions like ReLU, are the most exploited [46, 106–109]. With effect, adding perturbations aligned with the ANN gradient, pushing input samples towards neighboring classes is a common technique exploited by adversarial attacks.

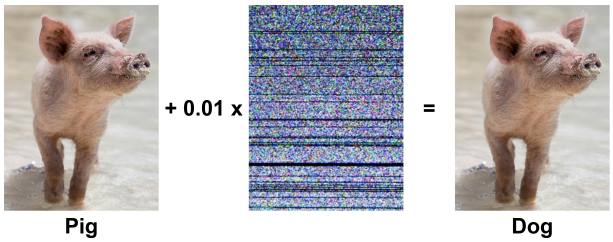


Figure 4: Adversarial example in image data.

Evasion vs. Poisoning Attacks. The integrity of ANNs can be compromised at two distinct stages, with attacks categorized into evasion and poisoning attacks [107, 108]. Evasion attacks occur during the inference phase, where the model is tricked into making incorrect predictions, while poisoning attacks target the training phase, manipulating the learning process. However, as TinyML systems predominantly

perform inference rather than training - research on ANN training at the MCU level is still at an embryonic stage - we frame the scope of this work to inference-time attacks.

Physical vs. Digital Domain. During inference, a typical ML pipeline involves (i) collecting input features from sensors or repositories, (ii) processing these features digitally, (iii) making inferences about the environment using the model, and (iv) relaying results to actuators [110]. Adversarial examples can disrupt this pipeline either in the physical domain by tampering with sensors and network infrastructure or by modifying inputs in the digital domain [46, 107, 108, 110, 111]. This work focuses exclusively on digital-domain attacks and defenses, as they are more prevalent and dominate the current literature.

Targeted vs. Non-Targeted Attacks. Adversarial attacks can be classified as targeted or non-targeted [46, 106–109, 111]. Targeted attacks aim to mislead the ANN into classifying an adversarial example into a specific, predetermined class. In contrast, non-targeted attacks only ensure that the adversarial example is misclassified into any incorrect class other than the original one. Non-targeted attacks typically enable a broader assessment of the robustness and generalization of a neural network.

Distortion Metrics. Generating adversarial examples can be described as a minimization problem (Equation 2.4). In this formulation, an input x, initially classified correctly by a machine learning model h, is perturbed with r to create an adversarial example x'. The adversarial example x' remains within the input domain of h, but is misclassified with a label l. Different metrics are used to quantify and minimize the perturbation r, leading to distinct attack strategies. Table 3 details the most common L_p norms to quantify adversarial distortion [46, 111, 112].

$$arg\min_{r} h(x+r) = l \;:\; x' = (x+r) \;\in\; D \tag{2.4}$$

Norm	Definition	Equation
L_0	Number of modified dimensions of the input	$L_0(x,x') = \sum_{i=1}^{n} 1(x_i \neq x'_i)$
L_1	The sum of the absolute distortion across all dimensions of the input	$L_1(x,x') = \sum_{i=1}^{n} \lvert x_i - x'_i \rvert$
L_2	Euclidean distance between the benign and adversarial inputs	$L_2(x,x') = \sqrt{\sum_{i=1}^{n}(x_i - x'_i)^2}$
L_∞	Maximum absolute distortion across all dimensions of the input	$L_\infty(x,x') = \max_{i\in\{1,2,\ldots,n\}} \lvert x_i - x'_i \rvert$

Table 3: Metrics to quantify the distortion of adversarial examples.

2.4.1 Attacks

According to the knowledge available to the adversary, evasion attacks can be performed in a white-, gray-, or black-box scenario. White-box attacks consider that the adversary has full access to the target ANN, including its architecture and parameters. In a gray-box scenario, the adversary does not know the architecture of the ANN or its internal parameters. Nevertheless, they have access to classification

Attack	Norm	Summary
		WHITE-BOX
FGSM [113] 2015	L_∞	Applies a predefined perturbation to each feature in the direction of the gradient's sign.
DeepFool [114] 2016	L_2	Uses iterative linearization of class boundaries to find the minimal perturbation that causes misclassification.
JSMA [115] 2016	L_0	Constructs an adversarial saliency map to iteratively perturb the most influential input features until misclassification.
C&W [116] 2017	L_0, L_2, L_∞	Formulates adversarial generation as an optimization problem balancing perturbation minimization and adversarial success, using a custom loss function.
BIM/PGD [57, 117] 2016/2018	L_∞	BIM is an iterative version of FGSM; PGD further refines this by projecting perturbed examples back into an ϵ-ball around the original input.
EAD [112] 2018	L_1 L_2	Generalizes C&W attack by incorporating an L_1 term in the optimization problem, promoting sparse perturbations.
FAB [118] 2020	L_0, L_2, L_∞	Improves over DeepFool by allowing for multiple norms and better handling of non-linear boundaries via convex combination projections.
AutoAttack [119] 2020	L_0, L_2, L_∞	Parameter-free attack that combines four attacks: two variants of PGD, FAB, and Square Attack.
		GRAY-BOX
ZOO [120] 2017	L_2	Adapts C&W attack for gray-box settings by using finite-difference methods to estimate gradients, and applies either first-order or second-order optimizers (ADAM/Newton).
Square Attack [121] 2020	L_∞	Adds random square-shaped perturbations to images and accepts them if they increase the classification error, repeating until misclassification.
		BLACK-BOX
Boundary Attack [122] 2018	L_2	Starts from a known adversarial example and iteratively reduces perturbation by projecting the adversarial example into a sphere around the original input.
GeoDA [123] 2020	L_2	Uses binary search to locate the decision boundary, approximates it with a local hyperplane, and moves the input along the normal direction until it is misclassified.

Table 4: Main characteristics of state-of-the-art adversarial attacks.

probabilities. Black-box attacks consider that the adversary can only feed input data to the target ANN and query its output. They do not have access to the class probability vector but rather to the final decision. In the following subsections, we review the attacks with the most impact and citation in the research community and whose implementation is available open-source. Table 4 summarizes the main characteristics of each attack considered in this thesis.

2.4.1.1 White-Box

Fast Gradient Sign Method (FGSM) [113] - 2015. FGSM was the first adversarial attack ever proposed. As detailed in Equation 2.5, it applies a perturbation δ to each feature/dimension in the same direction as the gradient. FGSM computes the gradient of the loss function with respect to each input feature and then scales the sign of this gradient by a predefined perturbation δ.

$$x' = x + \delta * Sign(\frac{\partial L(w, x, y)}{\partial x}) \tag{2.5}$$

DeepFool [114] - 2016. DeepFool is an iterative attack that seeks to find the minimum L_2 perturbation δ to misclassify x, moving it toward the closest boundary (Equation 2.6). At iteration t, DeepFool uses Taylor expansion to linearize the boundaries around x_t. At the end of each iteration, x_t is moved in the direction of the boundary with the lowest Euclidean distance. This process is repeated until x_t is misclassified. DeepFool has been widely used in the literature to measure the distance of input samples to the closest decision boundary.

$$min||\delta_t|| \text{ such that } L(x'_t) + \triangledown L(x'_t)^T \delta_t = 0 \tag{2.6}$$

Jacobian-based Saliency Map Approach (JSMA) [115] - 2016. An iterative attack that perturbs the most influential features of x. JSMA begins by computing the Jacobian matrix of x to derive the adversarial saliency map S, as detailed in Equation 2.7. The saliency map is then used to rank and distort features in order of their relevance to the final classification: each feature is distorted by δ until there is misclassification or the maximum number of distorted features (L_0), defined by the adversary, is reached.

$$S = \left\{ \begin{matrix} 0, \text{ if } \frac{\partial L_{y'}(x)}{\partial x_i} < 0 \text{ or } \sum_{y \neq y'} \frac{\partial L_y(x)}{\partial x_i} > 0 \\ \frac{\partial L_{y'}(x)}{\partial x_i} * \left| \sum_{y \neq y'} \frac{\partial L_y(x)}{\partial x_i} \right|, \text{ otherwise} \end{matrix} \right\} \tag{2.7}$$

Carlini and Wagner (C&W) [116] - 2017. An attack specifically designed to break defensive distillation - a defense described in Section 2.4.2. The C&W attack formulates the generation of adversarial examples as the optimization problem detailed in Equation 2.8, solved via gradient descent techniques. c is a trade-off parameter, defined by the adversary, that balances minimizing the perturbation δ and ensuring adversarial success. Instead of relying on the loss function of the target ANN, the C&W attack relies on a custom function detailed in Equation 2.9, where $Z(x')_y$ represents the probability of the target class y and $max\{Z(x')_y : y \neq y'\}$ represents the maximum probability among all non-target classes. k is a parameter, defined by the adversary, that sets the minimum confidence for the adversarial example. There are three variations of the C&W attack, each minimizing a different distortion metric (L_0, L_2, L_∞).

$$\min_{\delta} D(x, x + \delta) + c \cdot L(x + \delta) \tag{2.8}$$

$$L(x') = \max\left(\max\{Z(x')_y : y \neq y'\} - Z(x')_y - k\right) \tag{2.9}$$

Basic Iterative Method (BIM) [117] - 2016 / Projected Gradient Descent (PGD) [57] - 2018. BIM is an iterative version of FGSM: BIM executes FGSM over T iterations, applying a small perturbation α per iteration to each feature/dimension of x, till a maximum distortion of δ. PGD is a variant of BIM

that instead of clipping the maximum distortion of each dimension to δ, projects the adversarial examples created at each iteration to the δ-neighborhood of benign samples and starts the next iteration from the most adversarial point. By projecting adversarial examples into the δ-neighborhood of benign samples, PGD is better at finding the most adversarial example while minimizing the perturbation. PGD is formulated as detailed in Equation 2.10.

$$x_t = Proj_{x_t,\delta}(x_{t-1} + \alpha * Sign\frac{\partial L(w, x_{t-1}, y)}{\partial x_{t-1}}) \tag{2.10}$$

Elastic-Net Attack to DNNs (EAD) [112] - 2018. EAD generalizes the C&W attack by simultaneously minimizing the L_1 and L_2 norms. EAD was the first attack to incorporate the L_1 norm into its optimization problem, promoting sparsity in the perturbation in addition to minimizing its overall magnitude. The EAD formulation is presented in Equation 2.11, where the parameter c weights the loss function L against the distortion and β sets the relevance of L_1 norm in the optimization process.

$$min_{x'} c \cdot L(x', y') + \beta \left\|x' - x\right\|_1 + \left\|x' - x\right\|_2^2 \tag{2.11}$$

Fast Adaptive Boundary (FAB) Attack [118] - 2020. Similarly to DeepFool, the FAB attack aims to identify the smallest L_2 distance to the nearest decision boundary. However, the FAB attack extends this capability by supporting the L_0 and L_∞ norms. Additionally, the FAB attack is designed to handle with higher precision the non-linearity of class boundaries. For this purpose, after linearizing the boundaries around x_t using Taylor expansion, the FAB attack takes a convex combination of the projection of x_t and x on the estimated hyperplane to minimize the perturbation of the subsequent adversarial example x_{t+1}. This behavior is mathematically defined in Equation 2.12, where the parameter α controls how aggressively or conservatively the adversarial example x_{t+1} moves toward the original input while staying on the adversarial side of the boundary. A final extrapolation step ($\eta > 1$) ensures that x_{t+1} is adversarial.

$$x_{t+1} = Proj_c((1 - \alpha)(x_t + \eta\delta_t) + \alpha(x + \eta\delta)) \tag{2.12}$$

AutoAttack [119] - 2020. AutoAttack is a parameter-free ensemble attack that combines four methods: two variants of Auto-PGD, the FAB attack, and the Square Attack (SA). Auto-PGD extends PGD by dynamically adjusting the perturbation step size based on the optimization trend. Specifically, if the loss increases in at least 75% of recent iterations, the step size remains unchanged; otherwise, it is reduced by half. The two Auto-PGD variants differ in their loss functions: one employs cross-entropy, while the other utilizes the difference of logits ratio. The SA is a gray-box attack, described in Section 2.4.1.2.

2.4.1.2 Gray-Box

Zeroth-Order Optimization (ZOO) [120] - 2017. ZOO is an adaption of the C&W-$L2$ attack to the gray-box scenario. Unlike the original C&W attack, which uses softmax logits in its loss function, ZOO modifies the loss function to depend directly on the actual output of the ANN (Equation 2.13). To estimate the gradient of the loss, ZOO relies on a finite difference numerical method. ZOO estimates the gradient of the loss function by querying the model with slightly perturbed inputs and observes the changes in its output. For a given input dimension, the gradient is approximated as detailed in Equation 2.14, where h is a small step size and e_i is a basis vector with only the i^{th} component set to 1. To estimate the second-order derivatives (Hessian estimate), ZOO relies on a slightly different equation (Equation 2.15). To make the computation faster, ZOO employs stochastic coordinate descent over small batches of input dimensions per iteration, avoiding the calculus of derivatives with respect to all input dimensions. Two versions of the ZOO attack were proposed: (i) ZOO-ADAM and (ii) ZOO-Newton. The former uses the first-order gradient approximation and the ADAM optimizer to calculate the perturbation δ. The latter uses the second-order gradient approximation and the Newton's method to calculate the perturbation δ.

$$L(x') = \max\left\{\max\log[F(x)]_y : y \neq y' \ - log[F(x)]_{y'} - k\right\} \tag{2.13}$$

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h} \tag{2.14}$$

$$\frac{\partial^2 f(x)}{\partial x_{ii}^2} \approx \frac{f(x + he_i) - 2f(x) + f(x - he_i)}{h^2} \tag{2.15}$$

Square Attack (SA) [121] - 2020. An iterative attack that samples a random square-shaped perturbation per iteration and adds it to image x if it increases the classifier's error. There are two versions of SA, one tailored for the L_∞ norm and another for the L_2 norm. The SA-L_∞ initializes the perturbation as vertical stripes of width one, being the color of each stripe per color channel sampled from a uniform random distribution $[-\epsilon, \epsilon]$, where ϵ is the step size of the perturbation. At the end of each iteration, the perturbation is clipped to fit within the maximum perturbation δ. The attack stops as soon as it finds an adversarial example. The width and distortion of the squares decrease according to a fixed schedule. The SA-L_2 perform similarly but initializes the perturbation as $5x5$ squares uniformly distributed over the input image. The pixel values of each square are sampled from a uniform distribution $[-\delta, \delta]$.

2.4.1.3 Black-Box

Boundary Attack (BA) [122] - 2018. The attack starts by finding an initial adversarial example, where the value of each pixel is drawn from a uniform distribution. Then, the attack iteratively reduces the perturbation magnitude while ensuring the example remains adversarial. For this purpose, the attack

considers a sphere or neighbor-ball around the original sample x with a radius of δ, the maximum perturbation defined by the adversary. At each iteration, the attack samples a new direction from a Gaussian distribution for the current adversarial example x'_t and then projects it into the sphere around x. Each iteration ends by moving x'_t towards the original sample x by a step-size ϵ. The x'_t with the lowest L_2 perturbation respecting the adversarial condition is selected as the final adversarial example. The basic intuition behind BA is represented in Figure 5, where x is color-coded in green and x'_t in red.

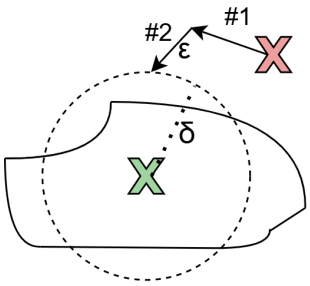


Figure 5: Basic principle of Boundary Attack.

Geometric Decision-Based Attack (GeoDA) [123] - 2020. GeoDA is based on the observation that state-of-the-art ANNs have a very low mean curvature in the neighborhood of data samples. Building on this insight, Rahmati et al. [123] employed a binary search algorithm to identify a point on the nearest decision boundary with relation to the input sample x. This point is then used to construct a hyperplane that approximates the boundary surface of the ANN near x. After defining the hyperplane, GeoDA uses an iterative method to find a vector normal to the boundary. More specifically, GeoDA samples a series of points in the vicinity of the decision boundary and based on the classifications of these points, GeoDA approximates the vector normal to the hyperplane. The original sample x is then moved along this normal vector until misclassification.

2.4.2 Defenses

Defenses against adversarial examples can be categorized based on the actuation time into (i) train-based defenses and (ii) input pre-processing defenses. Train-based defenses involve retraining the model to enhance its ability to correctly classify potential adversarial examples. In contrast, input pre-processing defenses are applied before the classifier during the inference phase. These defenses either clean input samples to eliminate potential adversarial noise or discard inputs when the detected distortion exceeds a predefined threshold, preventing them from being processed by the classifier. Table 5 summarizes the main characteristics of each defense considered in this thesis.

2.4.2.1 Train-Based Defenses

Defensive Distillation [56] - 2016. Defensive distillation divides the retraining of the target ANN into two stages. First, a teacher model is trained with the original dataset using the hard-coded (ground

Defense	Summary
TRAIN-BASED DEFENSES	
Defensive Distillation [56] - 2016	Uses a teacher-student training pipeline with softened outputs (softmax with high temperature) to reduce model sensitivity to input variations.
PGD Adversarial Training [57] - 2018	Augments the training dataset using adversarial examples generated by the PGD attack.
Ensemble Adversarial Training [58] - 2018	Augments the training dataset with adversarial examples crafted for different pre-trained models to increase robustness against adversarial example transferability.
Sinkhorn Adversarial Training [59] - 2021	Extends PGD adversarial training with a custom loss function to reduce adversarial perturbation. The new loss function combines cross-entropy and Sinkhorn Divergence.
INPUT PRE-PROCESSING DEFENSES	
Feature Squeezing [60] - 2017	Reduces input dimensionality via color depth reduction and spatial smoothing. Flags inputs as adversarial if the classification of squeezed samples differs largely.
PixelDefend [61] - 2018	Uses a generative model (e.g., PixelCNN) to purify inputs, projecting them back into the natural data distribution of the training dataset.

Table 5: Main characteristics of state-of-the-art adversarial defenses.

truth) labels. The teacher model is then used to perform inference on the training dataset to get the soft labels, i.e., the probability distribution of each training sample, resulting from the softmax activation of the last layer. After associating each training sample with the corresponding soft labels, the resulting dataset is used to train the student model. Defensive Distillation uses high distillation temperatures in softmax activation in both training stages. High distillation temperatures cause the softmax to return smoother probabilities, reducing the sensitivity of the ANN to input variations. These smoother probability distributions encourage the student model to generalize better by focusing on the relationships among classes rather than overfitting to specific input patterns. By smoothing the decision boundaries, defensive distillation forces attacks to create more distorted adversarial examples to cross these softened boundaries. Defensive distillation was originally tested and succeeded against the JSMA attack. Equation 2.16 details the softmax function with the temperature parameter T, where z_i is the input of the softmax function and N is the number of classes.

$$\text{Softmax}(z_i) = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_{l=0}^{N-1} \exp\left(\frac{z_j}{T}\right)} \tag{2.16}$$

PGD Adversarial Training [57] - 2018. Adversarial training retrains an ANN by injecting adversarial examples into the training dataset. The first work on adversarial training [113] proposed to generate adversarial examples at each training step using FGSM, mixing clean samples and adversarial examples. By adding adversarial examples with the true labels into the training set, the classifier will associate the adversarial examples with the correct class. This work was later borrowed by [57], which proposed adversarial training with PGD. Considering that PGD can be seen as a heuristic method that aims to find

the most adversarial example, adversarial training with PGD tends to make ANNs more robust on the regions that they are most likely to be fooled. PGD adversarial training was tested and showed robustness against: (i) FGSM, (ii) BIM, (iii) DeepFool, (iv) JSMA, and (v) the C&W attack. Since its proposal in 2018, PGD adversarial training remains one of the strongest defenses against adversarial examples.

Ensemble Adversarial Training [58] - 2018. This defense was proposed as an alternative to PGD adversarial training, which has shown to be time-consuming and difficult to scale to large datasets. Ensemble adversarial training leverages the concept of adversarial example transferability to decouple the generation of adversarial examples from the training process. Adversarial example transferability refers to the ability of an adversarial example to simultaneously evade the prediction on two or more ANNs performing similar tasks (trained on similar or equal datasets). Ensemble adversarial training augments the original training dataset with adversarial examples crafted for other pre-trained ANNs. Empirical results suggest that this approach is indeed effective in mitigating the effects of adversarial example transferability.

Sinkhorn Adversarial Training (SAT) [59] - 2021. SAT is a recent iteration of PGD Adversarial Training. SAT augments the training dataset with adversarial examples generated through the PGD attack; however, SAT adopts a combination of cross-entropy and Sinkhorn Divergence as the loss function (Equation 2.17). The Sinkhorn Divergence (Equation 2.18) is a regularized version of Wasserstein distance, which measures the cost of transporting one probability distribution (α) to another (β). In the context of SAT, it is used to calculate and reduce the Euclidean distance (L_2) between representations of original images and their adversarial counterparts. The SAT loss function is detailed in Equation 2.17, where L_{CE} is the cross entropy of adversarial examples and S_σ is the Sinkhorn Divergence between adversarial and original samples. SAT was originally tested against the PGD attack, exhibiting state-of-the-art robustness.

$$L_{SAT\sigma}(x, x', y, f) = L_{CE}(x', y, f) + S_\sigma(f(x), f(x')) \tag{2.17}$$

$$S_\sigma(\alpha, \beta) = W_{2,\sigma}(\alpha, \beta) - \frac{1}{2}W_{2,\sigma}(\alpha, \alpha) - \frac{1}{2}W_{2,\sigma}(\beta, \beta) \tag{2.18}$$

2.4.2.2 Input Pre-Processing Defenses

Feature Squeezing [60] - 2017. Feature Squeezing builds on the principle that the dimensionality of input samples is often unnecessarily large, increasing the attack surface. Consequently, Feature Squeezing proposes comparing the probability distribution of squeezed and unsqueezed samples. If their classification differs significantly, exceeding a pre-defined L_1 threshold, the sample is classified as adversarial. Feature Squeezing incorporates two dimensionality reduction techniques: (i) color bit-depth reduction and (ii) spatial smoothing. The color bit-depth of squeezed samples ranges from 1 to 8 bits, while spatial smoothing can be applied using local or non-local filters. The first empirical evaluation of Feature Squeezing demonstrated significant robustness against (i) FGSM, (ii) BIM, and (iii) PGD attacks.

PixelDefend [61] - 2018. PixelDefend uses a generative model before the classifier to project potential adversarial examples into the same data distribution as clean data, effectively denoising them. The generative network is trained on the same dataset as the classifier to learn the intrinsic data distribution. To ensure that the purified image looks similar to the original input, PixelDefend enforces a constraint whereby the maximum change in pixel values does not exceed a predefined threshold: $||x' - x||_\infty \leq \epsilon$. In its primary version, PixelDefend used PixelCNN as the generative network. This defense was initially evaluated against (i) FGSM, (ii) BIM, (iii) DeepFool, and (iv) the C&W attacks, demonstrating effectiveness in denoising adversarial examples.

2.5 Capsule Networks

The first concept of CapsNets in the context of ML was introduced in 2011 [124]. Capsules were conceptualized as groups of neurons structured into vectors, where each capsule represents a specific object or object part within an image. This representation marks a departure from the scalar-based input/output operations characteristic of CNNs and ANNs, as capsules work directly with vectors. These vectors encode information about posture, such as position and orientation, alongside object properties, such as width and height, which collectively describe an object or part of it.

CapsNets are designed to adhere to a part-whole hierarchy [53, 55, 124–128]. Each layer in a CapsNet comprises multiple capsules, and each capsule at one layer is connected to every capsule of the next layer via a transformation matrix, applied to the capsule outputs. The transformation matrix is used to encode the spatial relationship and other features that relate the object represented by a lower-layer capsule to the whole object represented by the capsule of the next layer [53, 55, 124–128]. The result of the multiplication between a capsule output and a transformation matrix is commonly referred to as a prediction vector. It represents what the lower-layer capsule predicts for the output of the capsule in the next layer. For an object to be recognized, the spatial relationships among lower-layer capsules must converge, ensuring consistent agreement [53, 55, 124–128], as depicted in Figure 6. This characteristic

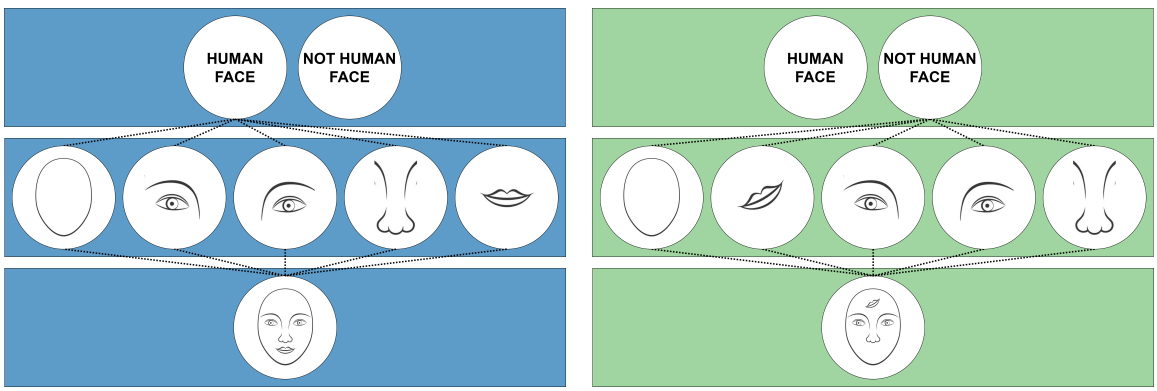


Figure 6: Basic principle of Capsule Networks: spatial agreement of object parts across lower-layer capsules is mandatory to recognize an object.

makes CapsNets a type of neural network closely related to the concept of inverse graphics, as one of its goals is to understand and represent the underlying structure of a visual scene.

Since the introduction of the capsule concept in 2011, various CapsNets architectures have been developed to address challenges across diverse domains. Image-based applications of CapsNets [126, 127] include healthcare, autonomous driving, and object detection, among others. More recently, CapsNets have been extended [128] to video and motion analysis, generative ML, and natural language processing. The following subsections provide a comprehensive review of the most significant CapsNets architectures [53, 124, 125], which served as foundational models for subsequent developments in this field.

2.5.1 Transforming Auto-Encoders

Transforming Auto-Encoders (TAEs) were the pioneering architecture behind CapsNets [124]. Unlike conventional auto-encoders, which focus on reconstructing input data, TAEs were specifically designed to encode an image and reconstruct it in its original pose. Each capsule in a TAE contains two key components: recognition units and generative units [55, 124, 126]. The recognition units are responsible for computing the pose parameters and estimating the probability that the object represented by the capsule exists in the input image. The output vector of a recognition unit is multiplied by the transformation matrix, and the resulting prediction vector is used as input for the corresponding generative unit. The generative unit recreates the object in the same pose as in the input image. More specifically, TAEs can take an image as input along with specified Δ_x and Δ_y shifts and generate the shifted image by combining the outputs of the generative units of each capsule [55, 124, 126].

In this first CapsNet implementation, the probability of the object represented by a given capsule being present in the input image is represented by a parameter in the output vector [55, 124, 126] - the magnitude of the output vector has no meaning. Despite their novelty, transformation matrices in TAEs have to be supplied externally. This limitation restricts their flexibility and makes them less autonomous than more recent CapsNets architectures.

2.5.2 Capsule Networks with Dynamic Routing

CapsNets with dynamic routing was the first CapsNet architecture tailored for classificaton tasks [53]. In this newer architecture, the probability of the object encoded in a capsule being present in the input image is represented by the magnitude of the capsule's output vector. However, the main improvement resides in the ability to infer autonomously the transformation matrices - they do not have to be provided externally as in TAEs. As shown in Figure 7, this new architecture starts with (i) a convolutional layer, followed by a (ii) primary capsule layer, and ends with (iii) one or more capsule layers. The convolutional layer extracts the most relevant features from the input image. Its output is sent to every capsule composing the primary capsule layer [53, 55, 126–128]. Primary capsules feature a convolutional kernel and are designed to refine the regions of interest detected by the preceding convolutional layer while extracting instantiation

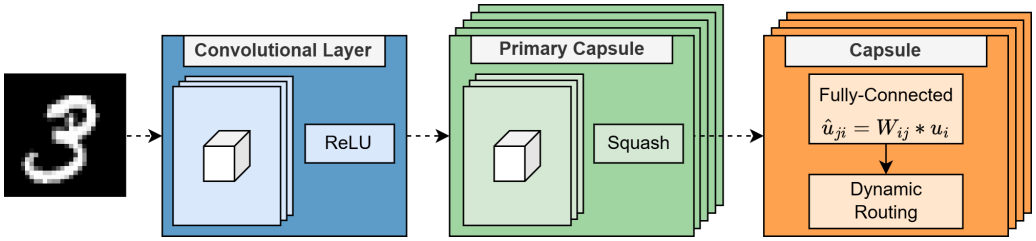


Figure 7: General architecture of a Capsule Network with dynamic routing between capsules.

parameters, including spatial and pose attributes [53, 55, 126–128]. Each primary capsule represents the lowest-level objects within the input image.

Every capsule i in layer L produces an output vector u_i encoding the instantiation parameters of the object represented by capsule i. This vector is sent to every capsule in layer $L+1$. A capsule j in layer $L+1$ receives the vector u_i and multiplies it by the corresponding transformation matrix W_{ij}, as detailed in Equation 2.19 [53, 55, 126–128]. The resulting vector $\hat{u}_{ij}$, known as the prediction vector, represents the prediction of capsule i in layer L for the object represented by capsule j in layer $L+1$.

$$\hat{u}_{ji} = W_{ij} * u_i \tag{2.19}$$

The prediction vector $\hat{u}_{ij}$ is then submitted to the dynamic routing between capsule i in layer L and capsule j in layer $L+1$. The dynamic routing iteratively determines the coupling coefficients c_{ij} between these two capsules, which represent the probability of the object encoded by capsule i in layer L being part of the object encoded by capsule j in layer $L+1$ [53, 55, 126–128]. If the agreement is high, the coupling coefficients increase, otherwise, they are reduced. This process repeats for all capsules of layer L connected to capsule j in layer $L+1$. As detailed in Equation 2.20, the output vector s_j of capsule j is a weighted sum of the prediction vectors from layer L, multiplied by the respective coupling coefficients [53, 55, 126–128]. The output vector of every capsule, including primary capsules, is subjected to a squash activation (Equation 2.21) to normalize the length of the vector to the range $[0, 1]$ [53, 55, 126–128].

$$s_j = \sum_i c_{ij} * \hat{u}_{ji} \tag{2.20}$$

$$v_j = \frac{||s_j||^2}{1 + ||s_j||^2} \frac{s_j}{||s_j||} \tag{2.21}$$

Algorithm 1 outlines the dynamic routing between capsules [53, 55, 126, 127]. In the first iteration, the prediction vector of a capsule i in layer L is routed equally to all capsules in layer $L+1$, as the logits b_{ij} of the coupling coefficients are initialized to zero. In subsequent iterations, the coupling coefficients c_{ij} are refined by evaluating the agreement between the prediction vector $\hat{u}_{ij}$ of capsule i and the squashed output v_j of capsule j. This agreement is calculated as the scalar product of $\hat{u}_{ij}$ and v_j, and the result is added to the logits b_{ij} of the corresponding coupling coefficients. To ensure the coupling coefficients

Algorithm 1: Dynamic routing in CapsNets.

1: for all capsule i in layer L and capsule j in layer $L+1$: $b_{ij} = 0$
2: **for** r iterations **do**
3: for all capsule i in layer L: $c_i = softmax(b_i)$
4: for all capsule j in layer $L+1$: $s_j = \sum_i c_{ij} * \hat{u}_{ji}$
5: for all capsule j in layer $L+1$: $v_j = squash(s_j)$
6: for all capsule i in layer L and capsule j in layer $L+1$: $b_{ij} = b_{ij} + \hat{u}_{ji} * v_j$
7: **end for**
8: **return** v_j

for a capsule in layer L sum to 1 across all capsules in layer $L+1$, the logits b_{ij} are normalized using a softmax function. The total number of iterations of the dynamic routing process is a hyper-parameter.

2.5.3 Matrix Capsules with Expectation-Maximization Routing

In 2018, Hinton et al. [125] introduced an enhancement to CapsNets by substituting the vector-based input and output representations of capsules with matrices and employing an Expectation-Maximization (EM) algorithm for dynamic routing. This modification reduced the size of the transformation matrices between capsules from n^2 to n, thereby improving computational efficiency [55, 125, 126, 128]. The new architecture encodes the probability of the object represented by a capsule being part of the input image in a separate activation parameter. Consequently, this innovation obviates the need for the squash activation function (Equation 2.21), which has been criticized for its lack of objectivity and sensitivity. With effect, the standard squash activation has been found to make gradients vanish when the length of the vector becomes very close to one [129, 130]. Furthermore, since the squash function changes a vector's length in a non-linear form, it can distort the relevance of a capsule in the given image and misguide the dynamic routing process between capsules [129, 130].

In this newer CapsNet architecture, the squash activation of primary capsules is replaced by a sigmoid activation and the kernel of the following capsules is convolutional instead of fully-connected [125, 128]. The new dynamic routing considers every capsule in a higher layer represented by a Gaussian distribution, defined by a mean μ and a standard deviation σ, and the capsules in a lower layer as the data points to be mapped to the Gaussian distributions [55, 125, 126, 128]. This algorithm encompasses two phases: (i) the assignment of data points to clusters (expectation) and (ii) the update of the centroids giving the assignments (maximization). Algorithm 2 outlines the EM algorithm. The process begins by calculating the votes V_{ij} from each capsule i in layer L for the pose of all capsules j in layer $L+1$. Initially, the routing coefficients are uniformly initialized, ensuring equal contribution from all capsules in layer L to those in layer $L+1$. The algorithm then iterates over a predefined number of maximization and expectation steps.

During the maximization step, the influence of each capsule i in layer L on the capsule j in layer $L+1$ is adjusted [55, 125, 128]. The pose matrix of capsule j is computed as the mean of the pose matrix votes from all capsules i in layer L. The routing algorithm then calculates the cost of activating capsule

Algorithm 2: Expectation-Maximization in CapsNets. β_a and β_u are learnable parameters.

Compute vote for pose matrix of capsule j in layer $L+1$: $V_{ij} = M_i * W_{ij}$
Initialize routing coefficients: $R_{ij} = \frac{1}{N_j} \forall j$
for r iterations **do**
  Maximization Step
  for each capsule j in layer $L+1$ **do**
    Adjust influence of i on j: $R_{ij} = R_{ij} \cdot a_i$
    Compute mean: $\mu_j = \frac{\sum_i R_{ij} V_{ij}}{\sum_i R_{ij}}$
    Compute variance: $\sigma_j^2 = \frac{\sum_i R_{ij} (V_{ij} - \mu_j)^2}{\sum_i R_{ij}}$
    Compute total cost: $cost_j = \sum \left(\beta_u + \log(\sigma_j)\right) \cdot \sum_i R_{ij}$
    Update activation: $a_j = \text{logistic}\left(\lambda(\beta_a - cost_j)\right)$
  end for
  Expectation Step
  for each capsule i in layer L **do**
    Compute alignment probability: $p_j = GaussianPDF(V_{ij}, \mu_j, \sigma_j^2)$
    Update routing weight: $R_{ij} = \frac{a_j p_j}{\sum_k a_k p_k}$
  end for
end for

j. Specifically, if the incoming votes exhibit high variance (σ_j), it is harder for capsule j to reliably explain the inputs, resulting in a higher cost. Conversely, if the votes are consistent (low variance σ_j), the cost is smaller. The total cost is subsequently used to compute the activation probability of capsule j in layer $L+1$. The expectation step [55, 125, 128] employs a Gaussian Probability Density Function (PDF) to estimate the probability that the vote V_{ij} coming from capsule i in layer L aligns with the pose matrix of capsule j in layer $L+1$. The routing coefficients are then updated based on these computed probabilities, refining the mapping of lower-layer capsules to higher-layer Gaussian distributions.

3

State of the Art

This chapter provides a comprehensive review of state-of-the-art research relevant to the work presented in this thesis. It is structured into three main sections. Section 3.1 examines techniques designed to protect the intellectual property and privacy of ANNs deployed on untrusted devices. This is followed by an analysis of the most significant studies conducted to date on assessing the robustness of QNNs against adversarial examples (Section 3.2) and a review of software and hardware solutions to accelerate the computation of CapsNets on edge devices (Section 3.3). The chapter ends with a discussion identifying gaps in the literature and how this thesis contributes to advancing current research in TinyML (Section 3.4).

3.1 Privacy of Machine Learning Models

Over the past few years, various TEE-based methods have emerged to protect the intellectual property and privacy of ML models deployed on untrusted devices. Many of these approaches [9, 12, 13, 15] rely on encrypting entire ANNs and storing them in non-secure memory. To reduce the TCB, these solutions sequentially transfer each ANN layer from non-secure to secure memory, where decryption occurs and activations are computed. However, while these techniques enhance privacy, they introduce performance overhead due to frequent cryptographic operations and context switches between the REE and the TEE. To mitigate these inefficiencies, alternative strategies have been explored, including (i) performing model loading in parallel with decryption and inference [36]; (ii) extending TEE capabilities [17, 37, 131]; (iii) re-structuring ANNs into bident structures [41]; (iv) obfuscating strategic ANN weights [16]; and (v) executing only strategic ANN layers within the TEE [11, 38–40].

This section provides a comprehensive analysis of the key approaches proposed to date for preserving the confidentiality and integrity of ML models deployed on untrusted devices. The discussion is organized based on the core techniques employed by these solutions.

3.1.1 Full Artificial Neural Network on Trusted Execution Environment

Offline Model Guard (OMG) [12] is a framework proposed by Bayerl et al. [12] that leverages the SANCTUARY security architecture to enable inference on untrusted devices powered by Arm Cortex-A APUs while protecting the privacy of ANNs. OMG leverages the SANCTUARY capabilities to derive a key from the platform certificate issued by the device vendor. This key, in combination with a nonce, is then used by the model owner (SP) to encrypt the target ANN. The encrypted ANN is subsequently deployed to the edge device and stored in a non-secure memory region. During inference, the entire ANN is transferred to the secure-world in a single iteration, where it is decrypted and stored within secure memory. Input samples from the normal-world are then forwarded to the secure-world for processing. Despite its straightforward design, OMG does not scale efficiently for medium or large ANNs, as the framework requires the entire neural network to fit within the secure memory, which is typically limited in capacity.

VanNostrand et al. [9] and MLCapsule [13] address the OMG limitation by partitioning the inference process and loading the ANN into the TEE on a layer-by-layer basis. In both approaches, the whole ANN is stored as ciphertext in non-secure memory. Upon receiving an input sample, the parameters of the first layer are transferred to the TEE, where they are decrypted and stored in a secure memory buffer. Once the computations for the current layer are completed, the outputs are stored in a secure memory buffer to serve as inputs for the subsequent layer. This process continues iteratively until the final layer is processed. Despite employing a similar strategy, these works target different processor architectures. MLCapsule [13] is designed for Arm Cortex-A processors using OP-TEE, whereas VanNostrand et al. [9] focuses on x86 platforms leveraging Intel-SGX. Beyond layer-wise partitioning, VanNostrand et al. [9] introduces two additional techniques to minimize the TCB: sub-layer partitioning and branched partitioning. In sub-layer partitioning, a single layer is subdivided into smaller segments. In branched partitioning, the final layers of an ANN are restructured into independent sub-layers, each maintaining distinct connections to subsequent layers. Despite potential advantages, VanNostrand et al. [9] does not evaluate these partitioning strategies.

Nakai et al. [36] argue that layer-wise partitioning offers clear benefits for TCB minimization; however, this strategy implies one cryptography operation per ANN layer, repeated for each input sample. This overhead, combined with frequent context switches between execution environments, adversely affects inference latency. To address this issue, Nakai et al. [36] used the shared cache between the normal and secure worlds in Arm Cortex-A APUs to perform parameter loading concurrently with parameter decryption and inference.

SecDeep [15] is a framework developed to restructure Arm-NN, an API designed for ML computations on Arm Cortex-A APUs, with the goal of minimizing the TCB of ANNs fully deployed within a TEE. Through a detailed analysis of the Arm-NN codebase, Liu et al. [15] found that more than 90% of the code is

devoted to auxiliary tasks, such as tensor preparation and CPU/GPU configuration, while less than 10% pertains to the core tensor computations that handle privacy-sensitive model and user data. Leveraging this observation, the authors proposed SecDeep, which partitions the functions of Arm-NN into two distinct categories: (i) confidential functions, which are executed within the secure-world, and (ii) non-confidential functions, which are executed in the normal-world.

3.1.2 Extension of Trusted Execution Environments

TEEm [131] is an extension of OP-TEE for Arm Cortex-A APUs, specifically designed to support ML workloads. OP-TEE restricts each TA to a maximum virtual address space of 1 gigabyte, a constraint arising from using a single L2 page table per TA. To overcome this limitation, TEEm extends OP-TEE by incorporating Single-to-Multiple (S2M) memory mapping policy and Parameter-based Memory Allocation (PbMA) mechanism. The S2M memory mapping policy enables multiple L2 page tables per TA, thereby significantly increasing the virtual address space available to individual applications. In parallel, the PbMA mechanism facilitates dynamic memory allocation based on the specific runtime needs of each TA, promoting more efficient and flexible memory usage.

Smart-Zone and Tinylib [17]. Smart-Zone is an innovative MMU for Arm Cortex-A APUs that dynamically adjusts the secure memory and the memory shared between normal and secure worlds, according to the specific requirements of a given ANN. This management includes virtual memory page tables and physical memory regions. The authors also propose Tinylib, a lightweight library to optimize ML computations within a TEE while maintaining a small TCB.

Penetralium [37] follows the same line of Smart-Zone - it extends the Intel-SGX technology of x86 APUs by incorporating a custom MMU optimized for ML computation. To fully leverage the capabilities of this enhanced MMU, the authors also introduce a framework for partitioning ANNs and configuring optimal parameter blocks for each layer, thereby minimizing computational overhead within the TEE.

3.1.3 Model Refactoring

MirrorNet [41] is a framework designed to restructure an ANN into two distinct ANNs, with one operating in the normal-world and the other in the secure-world. While both ANNs maintain the architecture of the original model, the secure-world ANN utilizes smaller kernels and plays a pivotal role in the final decision-making process. Conversely, the normal-world ANN retains the kernel sizes of the original model; however, its parameters are intentionally perturbed to significantly degrade accuracy, thereby preventing adversaries from reconstructing the original ANN. The outputs of each layer in the normal-world ANN are forwarded to the next layer within the normal-world and the corresponding layer in the secure-world ANN. In contrast, there is no feedback from the secure-world ANN to the normal-world ANN. The authors evaluated MirrorNet on an Arm Cortex-A processor leveraging Arm TrustZone.

3.1.4 Weight Obfuscation

Hou et al. [16] proposed to fully deploy ANNs in the REE by obfuscating a strategic percentage of weights in descending order of their magnitude, i.e., contribution to the final decision. The crafted noise changes the predicted label of every input sample, bringing the accuracy of the ANN below the value of random choice. Within this technique, the inference process is mostly performed in the REE, while the TEE is responsible for denoising the outputs of each ANN layer. The intellectual property of an ANN is protected when an attacker has to pay the same machine training hours to recover the model as the model owner training from scratch. Furthermore, an attacker must not be able to identify which weights have been subverted. For this purpose, crafted weights are drawn from the same distribution as the original ones. The framework was evaluated on x86 APUs with support for Intel-SGX.

3.1.5 Selected Layers on Trusted Execution Environment

Slalom [38] and ShadowNet [39] introduced a selective layer execution technique that confines only strategic layers of the ANN within a TEE. Slalom partitions the neural network into linear and non-linear layers, assigning the execution of non-linear components to the enclave, which manages the overall inference process. When a linear layer is encountered in the pipeline, the enclave encrypts both the inputs and layer parameters before offloading the computation to a co-located GPU. Upon completion, the GPU returns the computed outputs to the enclave, where they are subsequently decrypted. Beyond preserving ANN privacy, Slalom also ensures ANN integrity. By applying Freivalds' algorithm, Slalom can verify if the outsourced computation has not been compromised by a malicious attacker, while significantly increasing the computation speed of ANNs. Slalom targets x86 processors with support for Intel-SGX. Sun et al. [39] borrowed this concept and proposed ShadowNet, a similar framework tailored for mobile devices powered by Arm Cortex-A APUs with support to OP-TEE.

SecureFrameNet [40] divides ANNs according to the relevance of each layer to the classification task. The authors rely on model pruning techniques to establish a metric for quantifying the contribution of each neural network layer to the final decision. SecureFrameNet then uses the importance score of each layer to partition the ANN into two or more subgraphs. During the segmentation process, SecureFrameNet also evaluates the Floating Point Operations (FLOPs) associated with each subgraph. The authors recommend that only 40% of the total ANN FLOPs be computed within the TEE, optimizing both privacy and computational efficiency. SecureFrameNet targets Arm Cortex-A processors with support for protected Virtual Machine (pKVM).

DarkneTZ [11] is a framework with a slightly different goal - instead of focusing on the intellectual property of the ANN, DarkneTZ focuses on protecting the privacy of the training dataset. The authors proposed a powerful white-box MIA that traverses the model layer-by-layer to determine the layers that leak more information about the training dataset. Experimental results indicate that the final layers tend to

reveal more sensitive information. Consequently, DarkneTZ partitions a given ANN into a confidential and a non-confidential block. The non-confidential block, consisting of the first l layers, is stored and executed within the normal-world. Meanwhile, the confidential block is stored as a cyphertext in non-secure memory and transferred to the secure-world as the inference proceeds. Within the TEE, the confidential layers are decrypted, and their outputs are computed. DarkneTZ supports Arm Cortex-A APUs with OP-TEE.

3.2 QNN Robustness Against Adversarial Examples

Shumailov et al. [49] were among the first to investigate the adversarial robustness of QNNs. Their study systematically evaluated neural networks quantized via the Mayo tool [132], configured to perform quantization-aware training and return QNNs with fixed-point quantization. The bit-width configurations evaluated by Shumailov et al. [49] included $Q1.2$, $Q2.5$, and $Q4.n$. For configurations with a total bit-width exceeding 8 bits, the integer part was consistently assigned 4 bits ($Q4.n$). The evaluation was conducted on two neural network architectures: LeNet-5 trained on MNIST and CifarNet trained on CIFAR-10. To assess adversarial robustness, the authors employed three adversarial attacks: (i) Iterative Fast Gradient Sign Method (IFGSM), (ii) Iterative Fast Gradient Method (IFSM), and (iii) DeepFool. The study investigated not only direct attacks on QNNs but also the transferability of adversarial examples between full-precision ANNs and QNNs and vice-versa. The key findings of the study include:

- QNNs operating in int-8 or lower bit-widths show defensive behavior, mainly because of the reduction in integer precision.
- Clipping activations change the feature space of QNNs, which could offer marginal protection against the transferability of adversarial examples originally crafted for full-precision ANNs.
- The success of adversarial example transferability from QNNs to full-precision ANNs decreases as the number of bits dedicated to the integer part decreases.

Bernhard et al. [8] were the first to investigate QNNs under non-white-box conditions by incorporating gray-box adversarial attacks into their analysis. In addition to standard white-box attacks such as FGSM, BIM, and the C&W-L_2 attack, their evaluation suite included the Simultaneous Perturbation Stochastic Approximation (SPSA) and ZOO attacks. The study focused on QNNs with bit-widths ranging from 1 to 4 bits, trained using quantization-aware methods. Two quantization configurations were examined: (i) weight-only quantization, in which weights are quantized while activations remain in floating-point; and (ii) full quantization, where both weights and activations are quantized. For weight-only QNNs with 1-bit precision, the authors employed the BinaryConnect [133] training method. For fully quantized networks with 1-bit weights and activations, the BinaryNet [134] method was used. For QNNs with 2-, 3-, and 4-bit precision, the authors relied on the DoReFa-Net [135] approach. All training methods incorporated the Straight Through Estimator (STE) to approximate gradients during backpropagation, thus enabling

effective optimization despite the non-differentiability introduced by quantization. Experiments conducted on CIFAR-10 and Street View House Numbers (SVHN) datasets revealed that:

- The loss surface of binarized models is more difficult to optimize over.
- Activation quantization can lead to gradient masking. As a result, gradient-based adversarial attacks (i.e., white-box attacks) may become less effective against QNNs. However, QNNs remain vulnerable to gradient-free attacks, typically used in gray-box and black-box scenarios.
- The poor transferability of adversarial examples from full-precision ANNs to QNNs can be a consequence of the quantization-shift phenomenon. By mapping two floating-point values to the same quantization bucket, quantization can erase the adversarial effect.
- Poor transferability can also result from gradient misalignment, wherein the direction of the gradient varies between full-precision ANNs and QNNs, and even among QNNs with different bit-widths. Empirical observations indicate that the cosine similarity between gradients of models with varying bit-widths tends to be close to zero.

Duncan et al. [51] evaluated the adversarial robustness of QNNs through the lens of relative robustness. The authors consider that a QNN is robust relative to its full-precision version if, whenever the full-precision ANN is proven robust against an adversarial attack, its quantized version remains robust against the same attack. The study focused on QNNs trained on MNIST and CIFAR-10 datasets and quantized to 8-bits for both weights and activations using TensorFlow Lite Micro, which is compatible with the CMSIS-NN API for Arm Cortex-M MCUs. Unlike the previous work, the authors adopted a post-training quantization setting rather than quantization-aware training. To craft adversarial examples, the authors relied on the Deep Learning Verified tool [136], which creates adversarial examples using scratches, changes to camera angle, and lighting conditions. Using this tool, they constructed two distinct sets of perturbed images: (i) one comprising inputs that are correctly classified by the full-precision ANN, and (ii) another consisting of inputs that successfully induce misclassification in the full-precision ANN. Their empirical findings demonstrate the following:

- QNNs exhibit high levels of relative robustness when compared to their full-precision counterparts. Specifically, the QNNs maintained robustness against 98.07%–99.69% of the adversarial examples for which the corresponding full-precision ANNs were also robust.
- Quantization introduces noticeable resilience against adversarial example transferability; however, this resilience varies with the strength of the perturbation. From the smooth adversarial examples that disrupted misclassification on ANNs, only 52.05%-60.34% could equally deceive the corresponding QNN. However, this value rises to 89.61%-98.89% for bigger/sharp perturbations.

These findings led Duncan et al. [51] to propose that, beyond model compression, quantization may also serve as a mechanism to enhance adversarial robustness.

Gorsline et al. [14] introduced a geometric model to evaluate the robustness of QNNs under white-box adversarial attacks. In particular, the authors modeled a neural network as an n-dimensional hypercube, wherein the decision boundaries are represented by n-simplices. Within this framework, the Euclidean distance of a given input sample to the nearest decision boundary (i.e., simplex) is approximated by Equation 3.1, where q denotes the number of bits used to represent the network's weights.

$$\bar{\epsilon} \approx \frac{1}{\left(2q + \frac{1}{2}n^{3/2}\right)} \tag{3.1}$$

Based on the analysis of Equation 3.1, the authors presented the following insights:

- The average distance to the decision boundary decreases as weight precision and input dimensionality increase. This implies that quantization has a tangible effect on the geometry of a neural network's decision boundaries.

- There is an attack strength at which the variation of adversarial accuracy of neural networks at different bit-widths is negligible. This threshold is referred to as the critical attack strength. The existence of this phenomenon was empirically validated using the FGSM attack on QNNs trained on the MNIST and Spiral datasets, with bit-widths varying in the range $[1, 10]$ and quantized according to the deterministic rounding scheme described in [137].

3.2.1 Defenses

Lin et al. [50] proposed a pioneering defense against adversarial examples (Defensive Quantization), specifically tailored for QNNs. The defense aims at mitigating the quantization-shift phenomenon - a condition in which small input perturbations can be nullified by quantization, but larger perturbations may be amplified within the hidden layers of the QNN, making it less robust than the full-precision counterpart.

To tackle the quantization-shift phenomenon, the authors propose a training optimization function that maintains the Lipschitz constant of the QNN below 1, making the QNN non-expansive. The Lipschitz constant measures how much the output changes when the input changes. By forcing the Lipschitz constant to be below 1, the authors ensure that even when the magnitude of the perturbation is large enough to make the input sample jump to the next quantization bucket, the noise effect is not amplified in the hidden layers of the network. The Lipschitz constant of a given layer is, by definition, equal to the magnitude of the biggest weight. Therefore, to maintain the Lipschitz constant below 1, it is mandatory to make the weight matrix row W orthogonal, which can be achieved by keeping $W^T W I \approx I$, where I is the identity matrix. This led to the optimization function detailed in Equation 3.2, where $\mathcal{L}_{CE}$ is the original cross entropy loss and β is the weight of the novel regularization term $\left\|W_l^T W_l - I\right\|$.

$$\mathcal{L} = \mathcal{L}_{CE} + \frac{\beta}{2} \sum_{W_l \in W} \left\|W_l^T W_l - I\right\|^2 \tag{3.2}$$

Defensive Quantization was empirically validated on VGG-16 and Wide ResNet architectures trained on CIFAR-10 and SVHN datasets. Experimental results demonstrated that QNNs enhanced with this defense exhibited greater robustness to large adversarial perturbations than their full-precision counterparts. Moreover, the robustness gains were more pronounced in QNNs with lower bit-widths and for larger values of the regularization parameter β. The evaluation considered bit-widths ranging from 1 to 5 and included attacks such as FGSM, Random+FGSM, and PGD.

Song et al. [52] also relied on the Lipschitz constant to measure the effect of adversarial examples in weight-quantized neural networks with 8 bits or less. Through empirical experiments on a four-layer CNN trained on MNIST and a Wide ResNet-32 trained on CIFAR-10, the authors observed that adversarial trained ANNs are more prone to accuracy loss due to quantization, making adversarial training not directly portable to QNNs. This behavior was observed in a set of different attacks, including: (i) C&W-L_2, (ii) FGSM, (iii) PGD, (iv) BIM, and (v) Momentum Iterative Method (MIM).

To address this issue, Song et al. [52] propose a retraining technique that modifies the activation mapping between layers to be non-linear. For this purpose, the authors adopted the μ-law algorithm to dampen the sensitivity of gradients to small input changes, a characteristic commonly exploited by white-box adversarial attacks. The μ-law transforms activations as detailed in Equation 3.3, where x is the input activation and y is a hyperparameter that controls the degree of nonlinearity.

$$F(x) = \text{sgn}(x)\frac{\ln(1+\mu|x|)}{\ln(1+\mu)}, \quad -1 \leq x \leq 1 \tag{3.3}$$

3.3 Capsule Networks at the Edge

3.3.1 Software-Hardware Co-Design

SH-CapsNet [65, 66] is a software–hardware co-design framework developed to accelerate and optimize the inference phase of CapsNets on GPU platforms. This framework targets the routing procedure of CapsNets, which constitutes a major performance bottleneck due to (i) excessive off-chip memory access and (ii) frequent synchronization operations required to avoid race conditions in limited on-chip memory.

Profiling results revealed that (i) spatially adjacent capsules often share highly similar routing coefficients, and (ii) many capsule-to-capsule routing coefficients converge before completing all routing iterations. Based on these observations, the software component, termed S-CapsNet, introduces a training-time optimization that classifies capsules into essential and similar groups. For similar capsules, the routing algorithm is modified such that routing coefficients are computed only for the central capsule in a 3×3 region and reused for its neighboring capsules. The hardware component, PIM-CapsNet, complements the software by offloading the routing computations to a dedicated Processing-In-Memory (PIM) unit, while the convolutional layers remain executed on the GPU. This software-hardware co-design led

to a 3.41× improvement in inference speed and achieved up to 68.72% energy savings compared to conventional GPU-only implementations.

Fast Energy-Efficient CapsNet Accelerator (FEECA) and CapsAcc [64]. FEECA is a design space exploration framework that optimizes configurable hardware accelerators tailored to the execution of CapsNets. The framework takes as input the hardware design parameters of the configurable accelerator, the architecture of the target CapsNet, and a set of analytical models that estimate inference latency, chip area, and power consumption. To calculate the set of optimal trade-offs among these competing metrics, FEECA employs a genetic algorithm[1].

CapsAcc is an actual implementation of a CapsNet hardware accelerator - Application-Specific Integrated Circuit (ASIC) - that leveraged FEECA for its design. At its core, CapsAcc consists of four primary components: (i) a Processing Element (PE) array, (ii) an accumulator, and (iii) an activation unit. The PE array is a two-dimensional grid of PEs designed to enable parallel MAC operations. Intermediate results are stored in the accumulator, which also performs additions when necessary. The activation unit supports multiple activation functions used in CapsNets, including ReLU, squash, and softmax. To accelerate performance, the squash and softmax functions are implemented using LUTs. Experimental evaluation using a CapsNet trained on the MNIST dataset demonstrated that CapsAcc achieves a 6× speedup compared to an Nvidia GTX 1070 GPU.

FastCaps [138] is a two step approach to efficiently accelerate CapsNets on Field-Programmable Gate Arrays (FPGAs). At the software level, the authors propose to prune the convolution and primary capsule layers using their Look-Ahead Kernel Pruning (LAKP) technique. Unlike traditional magnitude-based pruning, LAKP defines the score of a given kernel as a sum of the scores of each kernel weight, calculated by analyzing the effect of the weight on the current layer and the layers adjacent to it. The score of a given weight w in layer i is computed as detailed in Equation 3.4 and considers the Frobenius norms of connected weights in adjacent layers $i-1$ and $i+1$. Experiments on CapsNets trained on MNIST and Fashion-MNIST show that LAKP can achieve a compression rate up to 99.26% with an accuracy loss lower than 1%.

$$L_i(w) = |w| \cdot \|W_{i-1}[j,:]\|_F \cdot \|W_{i+1}[:,k]\|_F \tag{3.4}$$

At the hardware level, FastCaps optimizes the dynamic routing procedure. With effect, the authors argue that the squash function of dynamic routing involves multiple exponential and division operations, which are computationally expensive and hardware-unfriendly. FastCaps accelerates the squash function with a hardware module that approximates the exponential function using Taylor expansion and the division with a combination of logarithmic and exponential functions. Additonally, the FPGA accelerator contains an array of ten PEs, with support for 16-bit operands, where convolution takes place.

[1] A type of optimization algorithm inspired by the process of natural selection in biology.

3.3.2 Hardware Solutions

Park et al. [67] extended the CapsAcc accelerator [64], previously analyzed, to support CapsNets optimized for 3D point cloud segmentation in mobile environments. The proposed accelerator features two primary components: a convolution core and a dynamic routing core. The convolution core comprises four arrays of PEs, each configured to perform multiplications with 8-bit operands. Instead of traditional 1D convolutions, this core implements group convolution with channel shuffling. To increase throughput, connections between input and output channels are selectively removed. To compensate for the resulting accuracy degradation, a shuffling layer is introduced, interchanging the channels among all the groups. This design achieves up to a 1.55× improvement in throughput compared to CapsAcc [64].

The dynamic routing core is based on the empirical observation that most coupling coefficients remain nearly constant across routing iterations. Leveraging this, the routing operation is skipped when the change in coupling coefficients is below 0.005%, significantly reducing redundant computations. This optimization lowers off-chip memory accesses during the routing procedure by up to 39.1%, with minimal impact on accuracy. The squash function datapath is implemented as an L_2 norm approximation block, combined with a small LUT. The capsule outputs are fed into the L_2 norm approximation block, whose result is used as index of the LUT returning the output of the squash activation. Experimental results show that this design choice consumes 3.92x less power and occupies 5.14x less area than CapsAcc [64].

Hemmati et al. [139] proposed a hybrid solution combining the AMD Xilinx Deep Processing Units (DPUs) with a custom hardware unit to accelerate the inference pass of CapsNets in AMD Zynq Ultrascale+ FPGAs. In this design, the built-in AMD DPU is used to accelerate the initial convolutional and primary capsule layers. However, since current DPUs do not support dynamic routing, a custom hardware unit is introduced to accelerate the remaining capsule layers. This accelerator was designed to parallelize the execution of the softmax, MAC, and squash functions. To further reduce inference latency, the authors applied loop unrolling on critical parts of the dynamic routing procedure: (i) the weighted sum vector computation and (ii) the squash function. Specifically, the computation of weighted sum vectors was unrolled by a factor of 16, resulting in a 16× reduction in latency. The outer loop of the softmax function was unrolled by a factor of 8, yielding a latency reduction of over 15k clock cycles. The accelerator operates exclusively on 32-bit integer operands, ensuring compatibility with FPGA platforms. Experimental evaluation conducted on a Zynq UltraScale+ ZCU102 board, using a CapsNet trained on the MNIST dataset, demonstrated a 34× speedup compared to a CPU-only baseline running on a quad-core Arm Cortex-A53.

3.3.3 Software Solutions

Q-CapsNet [62] is an open-source quantization framework tailored for CapsNets. Q-CapsNets aims to efficiently reduce the bit-width of weights and activations, while satisfying two constraints: (i) accuracy tolerance and (ii) memory budget. Q-CapsNet encompasses three main steps:

- **Step 1:** All weights and activations are quantized to a fixed-point format with a single bit allocated for the integer part ($Q1.n$). A binary search is used to determine the minimum number of fractional bits that maintains the model's accuracy within 5% of the defined tolerance.

- **Step 2:** The bit-width of weights is reduced layer-by-layer until the memory budget is met. Later layers, typically more tolerant to quantization noise, are assigned lower bit widths. If the accuracy is higher than the accuracy tolerance, Q-CapsNet proceeds to Step 3A; otherwise, it proceeds to Step 3B.

- **Step 3A:** The bit-width of activations is reduced layer-by-layer until the accuracy of the CapsNet is just above the tolerance. Activations involved in the dynamic routing procedure usually tolerate a lower bit-width. At this step, the framework returns a quantized CapsNet meeting the accuracy tolerance and memory budget constraints.

- **Step 3B:** Starting on the model generated in Step 1, the framework reduces the bit-width of weights until the accuracy is just above the tolerance. At this step, the framework returns a quantized CapsNet meeting the accuracy tolerance and another meeting the memory budget (from Step 2).

Experimental results extracted for a CapsNet trained on the CIFAR-10 dataset have shown that Q-CapsNet reduced the memory footprint by 6.2x with only 0.15% of accuracy loss.

Kakillioglu et al. [68] proposed a weight pruning framework for 3D CapsNets. The pruning method is based on the Alternating Direction Method of Multipliers (ADMM) optimization algorithm. It aims to eliminate redundant weights while minimizing the overall loss function. The ADMM algorithm decomposes the optimization problem into two subproblems. The first subproblem (Equation 3.5) corresponds to the minimization of the original loss function with a dynamic weight decay regularization. This regularization term is solved on the second subproblem (Equation 3.6), which can be seen as a Euclidean (Π) projection onto a sparse set, where only the largest-magnitude weights in each layer are retained and the remaining are set to zero. Experiments on 3D CapsNets trained on ModelNet-10 and ModelNet-40 datasets demonstrate that the framework can return 3D CapsNets as accurate as the original models while having 4.54× to 33.48× fewer parameters.

$$\min_{\{W_l\},\{b_l\}} f\left(\{W_l\}_{l=1}^{N}, \{b_l\}_{l=1}^{N}\right) + \sum_{l=1}^{N} \frac{\rho_l}{2} \left\| W_l - Z_l^k + U_l^k \right\|_F^2 \tag{3.5}$$

$$Z_l^{k+1} = \Pi_{S_l}(W_l^{k+1} + U_l^k) \tag{3.6}$$

- $f(\cdot)$ is the original loss function,

- W_l and b_l are the weights and biases for layer l,

- Z_l^k is the sparse projection of W_l,
- U_l^k is Lagrange multiplier,
- ρ_l is the penalty parameter for layer l,
- $\|\cdot\|_F$ denotes the Frobenius norm.

3.4 Gap Analysis

3.4.1 Privacy of Machine Learning Models

Table 6 puts the framework proposed in this thesis for preserving the privacy of QNNs on Arm Cortex-M MCUs (SecureQNN - Chapter 5) in perspective against the works described in Section 3.1. As observed, previous research focuses on Arm Cortex-A and x86 APUs with TrustZone and Intel-SGX, respectively. To the best of our knowledge, SecureQNN is the first framework to account for the constraints imposed by Arm Cortex-M MCUs and TrustZone-M. Below, we analyze the constraints of porting previous works to Cortex-M MCUs.

Full Artificial Neural Network on Trusted Execution Environment. Storing a full QNN as ciphertext in non-secure memory and loading it into the secure-world at inference time, such as the OMG [12] framework, is impractical for Cortex-M MCUs. TF-M, the reference secure firmware for Cortex-M MCUs, allocates memory for the secure and normal worlds during the boot process and maintains it static afterwards. Given that Cortex-M MCUs feature only a few megabytes of memory at best, accommodating a complete ANN within the secure-world would significantly limit the execution of other applications in the normal-world.

Deploying the ANN layer-by-layer to the secure-world, like VanNostrand et al. [9], MLCapsule [13], and Nakai et al. [36], improves efficiency by requiring memory only for the largest QNN layer within the TEE. However, this approach requires a cryptographic operation for each layer every time an input sample is received. Due to the limited computational capabilities of Cortex-M MCUs, especially when compared to more powerful Arm Cortex-A or x86 processors, this layer-wise cryptographic overhead might degrade inference latency. Additionally, Cortex-M MCUs lack support for GPU acceleration and shared caches between normal and secure worlds to compensate for the additional overhead.

SecDeep [15] introduces a novel strategy by isolating the Arm-NN API functions responsible for handling user and ANN data within the secure world. This approach is particularly promising for Cortex-A APUs, as these functions constitute only 10% of the overall Arm-NN codebase. However, CMSIS-NN, the state-of-the-art API for ML on Cortex-M MCUs, is specifically optimized for resource-constrained environments. Unlike Arm-NN, it does not provide functionality for tensor preparation or hardware configuration, making SecDeep not effective on Cortex-M MCUs.

	Target Device			Trusted Execution	
	Architecture	*TEE*	*TinyML*	*Category*	*Summary*
OMG [12] 2020	Arm Cortex-A	SANCTUARY	No	Full-ANN on TEE	Saves the ANN as cyphertext in non-secure memory. Loads the model to the TEE in a single iteration.
VanNostrand et al. [9] 2019	Arm Cortex-A	OP-TEE	No	Full-ANN on TEE	Saves the ANN as cyphertext in non-secure memory. Loads the model layer-by-layer to the TEE.
MLCapsule [13] 2021	x86	Intel-SGX	No	Full-ANN on TEE	Saves the ANN as cyphertext in non-secure memory. Loads the model layer-by-layer to the TEE.
Nakai et al. [36] 2021	Arm Cortex-A	OP-TEE	No	Full-ANN on TEE	Uses the shared memory between the REE and the TEE to load the model in parallel with prediction.
SecDeep [15] 2021	Arm Cortex-A	OP-TEE	No	Full-ANN on TEE	Arrm-NN functions dealing with user and model data and are delegated to the TEE.
TEEm [131] 2024	Arm Cortex-A	OP-TEE	No	TEE Extension	Extension of OP-TEE with S2M memory mapping policy and PbMA mechanism.
Smart-Zone and TinyLib [17] 2024	Arm Cortex-A	Arm TrustZone	No	TEE Extension	Custom MMU/API for ML computation within TEE.
Penetralium [37] 2024	x86	Intel-SGX	No	TEE Extension	Custom MMU for ML computation within TEE.
MirrorNet [41] 2023	Arm Cortex-A	Arm TrustZone	No	Model Refactoring	Refactors 1 ANN in 2 ANNs. The ANN within the TEE contributes the most to the classification task.
Hou et al. [16] 2022	x86	Intel-SGX	No	Weight Obfuscation	Obfuscates strategical weights and denoises the output of each layer within the TEE.
Slalom [38] 2019	x86	Intel-SGX	No	Selected Layers on TEE	Offloads linear layers from the TEE to the GPU.
ShadowNet [39] 2023	Arm Cortex-A	OP-TEE	No	Selected Layers on TEE	Offloads linear layers from the TEE to the GPU.
SecureFrameNet [40] 2024	Arm Cortex-A	N.A.	No	Selected Layers on TEE	Uses pruning-based techniques to identify the layers that should be offloaded to the TEE.
DarkneTZ [11] 2020	Arm Cortex-A	OP-TEE	No	Selected Layers on TEE	Offloads layers that potentiate MIAs to the TEE.
SecureQNN This Thesis	**Arm Cortex-M**	**TF-M**	**Yes**	**Selected Layers on TEE**	**Offloads the layers that lessens the effort of an attacker to build a surrogate ANN to the TEE.**

Table 6: Gap analysis: privacy of ML models. N.A. means Not Applicable.

Extension of Trusted Execution Environments. Arm Cortex-M MCUs lack support for virtual memory, which makes the S2M memory mapping policy and the PbMA mechanism of TEEm [131] not adequate for this class of devices. The same applies for the TEE extension approaches adopted in SmartZone [17] and Penetralium [37], as Cortex-M MCUs lack support for MMUs. While extending the TEE for ML workloads presents an avenue for future research, existing approaches for Cortex-A [17, 131] and x86 processors [37] are not directly portable.

Model Refactoring. Refactoring a ANN into two models, one to be executed within the normal-world and the other in the secure-world, might be an interesting solution for ML systems working with large and heavy models. However, in TinyML systems, QNNs are heavily optimized for maximum accuracy at minimum memory footprint. Additionally, Arm Cortex-M MCUs are generally limited to a single core, which restricts their ability to execute two neural networks in parallel. Consequently, MirrorNet [41] is hardly portable to Arm Cortex-M MCUs, particularly given that MirrorNet requires the network of the normal-world to replicate both the architecture and kernel sizes of the original ANN.

Weight Obfuscation. The weight obfuscation technique proposed by Hou et al. [16] is among the most efficient techniques proposed to date for reducing the TCB in TEE-based ML systems. However, it implies two context switches per ANN layer to denoise activations, a process that has to be repeated for every input sample. Therefore, while optimal from the TEE memory footprint perspective, the weight obfuscation technique proposed by Hou et al. [16] might compromise the inference latency in Arm Cortex-M MCUs.

Selected Layers on Trusted Execution Environment. Slalom [38] and ShadowNet [39] depend on GPU-based acceleration to compensate for the overhead caused by the frequent cryptographic operations and context switches. Therefore, they are unsuitable for Arm Cortex-M MCUs. Nevertheless, storing and executing strategic layers within a TEE is an interesting approach to trade-off TCB and inference latency on TinyML systems powered by Arm Cortex-M MCUs. With effect, the framework we propose in this thesis (SecureQNN) adopts this approach, taking inspirations from SecureFrameNet [40] and DarkneTZ [11] frameworks. However, unlike SecureFrameNet, which determines the criticality of a layer based on its impact on output predictions, SecureQNN perceives layer criticality in terms of how it lessens the effort of a possible attacker in constructing a surrogate model with equal or higher accuracy. Additionally, SecureFrameNet relies on FLOPs to estimate the workload within the TEE. This metric is incompatible with the integer-precision nature of neural networks that dominate Arm Cortex-M MCUs. DarkneTZ focuses mainly on protecting training data privacy, rather than defending the model structure itself.

3.4.2 QNN Robustness Against Adversarial Examples

Table 7 puts the comprehensive empirical evaluation we performed in this thesis on the QNN adversarial robustness (Chaper 6) in perspective against the works described in Section 3.2. This comparison is structured around three key dimensions: the quantization policies adopted, the effect of quantization on the robustness of QNNs in the face of adversarial attacks, and the combined influence of quantization and TinyML-related constraints on widely used adversarial defenses. Below, we detail how this work contributes to and extends the current state-of-the-art.

Attacks. The main indicator used in previous works to evaluate the robustness of QNNs is the *transferability vs. quantization-shift*, which represents how adversarial examples generated for full-precision ANNs transfer to QNNs. As quantization maps more than one floating-point value to the same integer bucket, QNNs can respond differently than their full-precision counterparts to the same adversarial input. This transferability gap is the main focus of several prior works, though their findings are inconsistent.

While Lin et al. [50] states that QNNs are more vulnerable to adversarial examples, Duncan et al. [51] suggests that QNNs show increased robustness when compared to ANNs. These contradictory results likely stem from differing quantization policies. Duncan et al. [51] evaluate int-8 models quantized using TensorFlow Lite Micro, whereas Lin et al. [50] do not provide details on the quantization policy used. The conclusions from Shumailov et al. [49] sit in the middle, stating that QNNs offer increased robustness

	Work						
	Shumailov et al. 2019 [49]	Bernhard et al. 2019 [8]	Duncan et al. 2020 [51]	Gorsline et al. 2021 [14]	Lin et al. 2019 [50]	Song et al. 2021 [52]	**This Thesis**
ATTACKS							
Point distance to the decision boundary (T1)				•			•
Gradient masking (T2)		•					•
Transferability vs. quantization-shift (T3)	•	•	•	•	•	•	•
Gradient misalignment (T4)		•					•
DEFENSES							
Point distance to the decision boundary (T5)							•
Adversarial example transferability (T6)							•
Defense transferability to TinyML (T7)							•
Compare relevant pre-processing defenses (T8)							•
Distortion of adversarial examples (T9)							•
Train-Based vs. Pre-Processing defenses (T10)							•
QUANTIZATION							
Policy	Mayo Tool	Binary Connect Dorefa Net	TF Lite Micro	N.D.	N.D.	N.D.	TF Lite Micro
Bit-width	[4; 28]	[1; 4]	8	[1; 10]	[1; 5]	[1; 8]	8; 16
Quantized Parameters	Weights Activations	Weights Activations	Weights Activations	Weights	Weights Activations	Weights	Weights Activations

Table 7: Gap analysis: adversarial robustness of QNNs. Filled-circles (●) indicate that a given work empirically evaluates a given takeaway. Empty-circles (○) indicate that a given work does not empirically evaluate a given takeaway. N.D. means Non-Defined.

when its bit-width is equal to or lower than int-8. Shumailov et al. [49] consider QNNs fine-tuned via quantization-aware training, following the quantization policy of the Mayo tool [132] and with bit widths varying between int-4 and int-28. Song et al. [52] also focused their evaluation exclusively on the *transferability vs. quantization-shift* indicator; however, they focused on QNNs enhanced against adversarial examples via adversarial training. Despite their conclusions, all of these works [49–52] restricted their analysis to the transferability gap, overlooking other key factors that affect QNN robustness.

A more complete perspective is offered by Bernhard et al. [8], who were the first to identify and analyze two critical phenomena: *gradient masking* and *gradient misalignment*. Gradient masking occurs when the quantization process hides the gradient of the QNN by making it explode or become zero. Gradient misalignment refers to the difference in gradient direction between a QNN and its full-precision ANN counterpart. The effect of quantization on the *point distance to the decision boundary* was only observed and evaluated by Gorsline et al. [14]. This indicator illustrates how quantization impacts the decision boundaries of a neural network.

Given the scattered information on QNN robustness and the lack of a well-defined basis for ground comparison, this thesis systematizes and empirically evaluates all the conclusions drawn from previous works, under the quantization policy of APIs widely used in TinyML (CMSIS-NN and TensorFlow Lite Micro). Compared to previous research, our empirical evaluation is far more extensive, including ten state-of-the-art attacks with high impact and citation in the research community. Furthermore, this thesis is the first work to include a parameter-free attack (AutoAttack), which reduces the chance of bad results due to

human error in hyperparameter tuning. We also give proper attention to the black-box scenario, which is more reflective of real-world threat models. While previous studies ignored this setting entirely or relied on weak perturbation schemes, such as adding random noise to images [50, 52], our empirical evaluation includes two powerful black-box attacks.

Defenses. Previous work addressing adversarial example defenses for QNNs is limited, with Lin et al. [50] and Song et al. [52] representing the primary contributions. Both works propose a training optimization function (loss function) aimed at controlling the quantization-shift in QNNs. Despite their contributions, neither study fully discloses the target quantization policy. Based on the limited information available, it appears unlikely that either is compatible with CMSIS-NN or TensorFlow Lite Micro, the prevailing standard ML APIs for MCUs. These APIs are not compatible with the bit widths considered in these works.

This thesis adopts an alternative perspective by systematically evaluating how existing adversarial defenses developed for full-precision ANNs perform when applied to QNNs deployed on MCUs, within the context of TinyML. We advocate that before developing defenses for this scenario, it is important to understand if and how current defenses perform on it. Our empirical evaluation comprises six open-source and widely cited state-of-the-art defense mechanisms, spanning the two principal categories: training-based defenses and input pre-processing methods. To the best of our knowledge, this is the first empirical investigation to evaluate: (i) how quantization interferes with the success of defenses (T5, T7, T9); (ii) how defenses affect the adversarial example transferability from ANNs to QNNs (T6); (iii) which pre-processing defenses are preferable to be deployed on MCUs (T8); and (iv) which type of defense (train-based vs. pre-processing) is preferable and in which circumstances (T10).

Quantization and TinyML. As discussed in Section 2.3, QNNs can be deployed across a wide spectrum of hardware platforms, including ultra-low-power MCUs, mobile APUs, and GPUs. Each target introduces distinct architectural constraints and requires specific quantization policies. In particular, QNNs designed for GPUs tend to resemble full-precision ANNs more closely than those tailored for MCUs. These differences are evident in several dimensions, including quantization policy, bit-width, kernel size, and network depth. A recurring issue in prior work is the lack of clarity regarding the quantization policy [14, 50, 52]. Others [8, 49] rely on quantization techniques that produce QNNs whose gradients can be estimated by STE and Mirror Descent techniques, which are not applicable in CMSIS-NN and TensorFlow Lite Micro.

At the time of writing of this thesis, CMSIS-NN and TensorFlow Lite Micro are the state-of-the-art APIs for ML computation on Arm Cortex-M MCUs, being tailored to exploit the SIMD capability of these platforms [18, 20]. Although the work from Duncan et al. [51] meets this requirement, their focus is limited to analyzing the effect of quantization-shift on adversarial transferability (T3). To the best of our knowledge, this thesis pioneers research on adversarial example robustness of QNNs tailored for MCUs powering TinyML applications. Our evaluation includes three QNN architectures: two based on image classification tasks from the MLPerf Tiny benchmark suite [24], and one specifically designed for efficient execution on Arm Cortex-M MCUs [140].

3.4.3 Capsule Networks at the Edge

Table 8 puts the quantization framework and the APIs proposed in this thesis for the execution of CapsNets in Arm Cortex-M and RISC-V MCUs in perspective against the works described in Section 3.3. As observed, previous research focused on two major points: (i) reducing the inference latency of CapsNets and (ii) the memory footprint of CapsNets. Below, we detail how this thesis addresses these topics and how it extends the current state-of-the-art.

Latency. In terms of inference latency, most prior works address this challenge through dedicated hardware accelerators, implemented in ASIC or FPGA modules. Marchisio et al. [64] followed a disruptive approach and proposed a modular hardware accelerator (CapsAcc), whose configurable parameters can be optimized to the execution of a specific CapsNet by a software tool (FEECA). Other works [65–67, 138, 139] followed a more traditional approach, proposing accelerators to support a broader range of CapsNet architectures. All these works put special effort in accelerating the dynamic routing procedure. Rahoof et al. [138] introduced FastCaps, which accelerates dynamic routing by approximating the squash activation with a Taylor series and replacing division with logarithmic and exponential functions. Hemmati

	Target Device	Problems Addressed	Summary
SOFTWARE-HARDWARE CO-DESIGN			
SH-CapsNet [66, 108] 2021	GPU	Latency	1. Software framework to optimize the dynamic routing. At training, it identifies sets of capsules with similar coupling coefficients. At inference, it computes the coupling coefficient of the central capsule in each set. 2. PIM unit to offload the computation of the dynamic routing from the GPU.
FEECA and CapsAcc [64] 2021	ASIC	Latency Energy	1. Software to optimize the architecture of an accelerator for a given CapsNet. 2. Hardware accelerator for CapsNet inference.
FastCaps [138] 2023	FPGA	Latency Memory	1. Software framework to prune the convolution and primary capsule layers. Prunning is performed using the LAKP technique. 2. Hardware accelerator that optimizes squash and divison operations. Squash is implemented as a Taylor expansion; divison as a series of log. and exp. functions.
HARDWARE SOLUTIONS			
Park et al. [67] 2020	ASIC	Latency Energy	1. Hardware accelerator for 3D CapsNet inference. It implements group convolution with channel shuffling. The routing operation is skipped when the change in coupling coefficients is below 0.005%.
Hemmati et al. [139] 2024	FPGA	Latency	1. Combines AMD DPU with a custom hardware unit to accelerate CapsNets. The DPU accelerates the convolutional and primary capsule layers. The custom unit accelerates capsule layers. It applies loop unrolling on the dynamic routing.
SOFTWARE SOLUTIONS			
Q-CapsNet [62] 2020	GPU FPGA	Memory	1. Quantization tool that reduces the bit-width of weights and activations in a CapsNet while optimizing two metrics: accuracy tolerance and memory budget.
Kakillioglu et al. [68] 2020	GPU CPU	Memory	1. Weight pruning tool for 3D CapsNets. It uses ADMM optimization to eliminate redundant weights while minimizing the overall loss function.
This Thesis	MCU	Memory Latency	1. CapsNet quantization tool supporting fixed int-8 data. 2. API for CapsNets inference in Armv7E-M, Armv8-M MCUs. 3. API for CapsNets inference in RISC-V (RV32IMCXpulp) MCUs.

Table 8: Gap analysis: CapsNets at the edge.

et al. [139] improved dynamic routing speed through loop unrolling. In turn, the hardware accelerator from Park et al. [67] skips the routing procedure when the change in coupling coefficients between capsules is below 0.005%. In contrast to all previous works, SH-CapsNet [65, 66] targets GPU platforms, offloading the dynamic routing procedure in a PIM unit.

Despite the novelty, all prior works address the inference latency issue in CapsNets by relying on high-performance and expensive hardware, such as ASICs, FPGAs, or GPUs with PIM. In this thesis, we advance the state-of-the-art by performing CapsNet inference on commercially available, ultra-low-power MCUs with constrained computational and memory resources. Specifically, we extended the CMSIS-NN and PULP-NN APIs of Arm Cortex-M and RISC-V (RV32IMCXpulp) MCUs with software kernels tailored to the computation of CapsNets in integer precision. We put special effort on the MAC optimization within capsule layers, trading-off the number of memory accesses and the speedup offered by SIMD available on the target MCU architecture. To accelerate the squash activation function, we employ the Newton-Raphson method [141] for efficient square root approximation. Our software kernels rely exclusively on integer-precision operations, a critical design choice given that most MCUs lack hardware FPUs.

Memory Footprint. To reduce the memory footprint of CapsNets, prior research has predominantly explored pruning and quantization techniques. FastCaps [138] employs the LAKP method to prune both convolutional and primary capsule layers, while Kakillioglu et al. [68] uses ADMM optimization to prune 3D CapsNets. While pruning effectively decreases model size, the primary requirement for enabling CapsNet deployment on MCUs, namely Arm Cortex-M and RISC-V (RV32IMCXpulp), is quantization. These devices typically lack support for hardware FPUs, rendering floating-point operations computationally expensive.

Although Marchisio et al. [62] proposed a quantization framework (Q-CapsNet), it relies on a custom quantization policy that is incompatible with the state-of-the-art ML APIs for MCUs. This thesis addresses that gap by presenting a CapsNet quantization framework aligned with the quantization specifications of CMSIS-NN and PULP-NN. While Q-CapsNet offers flexibility in bit-width and accuracy trade-offs, our framework is explicitly optimized for the SIMD instruction sets of MCU platforms, thereby enhancing inference efficiency on resource-constrained embedded devices.

4

Concluding Remarks

In this chapter, we summarize the key contributions and findings of this thesis. Section 4.1 outlines the primary conclusions drawn from the work conducted for each publication and answers the Researh Questions (RQ) defined in Section 1.2.2. Section 4.2 discusses potential avenues for future research.

4.1 Conclusions

As concerns surrounding data privacy intensify and ML applications increasingly permeate safety-critical domains, there is a notable shift toward performing ML inference on edge devices rather than in the cloud. MCUs are emerging as attractive platforms for on-device ML due to their energy efficiency and their ability to support QNNs, which maintain high inference accuracy with reduced computational overhead.

Nevertheless, deploying ML models on untrusted edge hardware raises concerns about intellectual property theft and the potential for training data leakage. While TEEs have been largely explored on high-end platforms (e.g., Arm Cortex-A, x86), their use on resource-constrained MCUs is still unexplored. This thesis introduces a framework that leverages the recent Arm TrustZone-M technology to protect the intellectual property of QNNs on Arm Cortex-M MCUs, with negligible impact on inference latency.

It is important to note, however, that TEEs alone are insufficient to mitigate all threats. Adversarial attacks are known to challenge the decision integrity of neural networks. Although the vulnerability of full-precision ANNs has been widely studied, the resilience of QNNs tailored for TinyML systems remains poorly understood. To address this gap, we present a comprehensive empirical evaluation of the adversarial robustness of QNNs, including potential defense mechanisms.

Beyond adversarial perturbations, the decision integrity of neural networks can also be compromised through object deformation. CapsNets have been proposed as a robust alternative, offering improved tolerance to spatial transformations. However, the quantization of CapsNets and their deployment on resource-constrained MCUs remain open research challenges. Our contributions include a quantization framework for CapsNets and the extension of state-of-the-art MCU-compatible ML APIs to support the inference pass of quantized CapsNets.

Privacy of Machine Learning Models. To evaluate the feasibility of leveraging Arm TrustZone-M to protect the intellectual property of QNNs in Arm Cortex-M MCUs, we developed SecureQNN. SecureQNN is a framework that determines the minimum set of privacy-critical layers that must be delegated to the secure-world to protect the intellectual property of the whole model (Chapter 5). We consider the intellectual property of a QNN protected when the effort required by an adversary to build a surrogate QNN with equal or higher accuracy is equal to the effort of the model owner when training from scratch.

Results extracted for two QNNs borrowed from the reference benchmark for TinyML systems (MLPerf Tiny [24]) suggest that it is possible to undermine the cost-benefit of unauthorized QNN replication without transferring the execution of the whole model to the secure-world. In the case of a ResNet trained on the CIFAR-10 dataset, our framework can maintain the privacy of the QNN by transferring only the last 3 out of 10 layers to the secure-world, corresponding to 50.92% of the model size. For a MobileNetV1 trained on the Visual Wake Words (VWW) dataset, the intellectual property can be preserved by delegating 9 out of 28 layers - 65.27% of the model size. Deployment experiments on a very low budget platform (STM Nucleo-L552ZE-Q) running each of these two QNNs upon TF-M, demonstrate the practicality of SecureQNN in real-world scenarios. The peak memory utilization within the secure-world is registered for the Random-Access Memory (RAM) and it never goes above 38.46% of the total device memory (256 KBytes).

Regarding latency, SecureQNN only imposes two context switches between the normal and secure worlds per input sample. Although context switching typically incurs latency overhead, our measurements show that the associated CPU cycle cost accounts for merely 0.04% to 0.06% of the total inference latency. Additionally, partitioning the QNN execution can result in compiler optimizations that, in some cases, reduce overall inference latency compared to executing the full QNN in the non-secure world.

In addressing **RQ1** (Section 1.2.2), these results suggest that TEEs, namely Arm TrustZone-M, are a viable mechanism to protect the privacy of QNNs in Arm Cortex-M MCUs, at least, to the point of undermining the cost-benefit of unauthorized replication. SecureQNN demonstrates that leveraging a TEE does not entail significant latency penalties nor restrict the ability to host concurrent applications in the secure or non-secure world.

QNN Robustness Against Adversarial Examples. To determine if and how QNNs tailored for TinyML are vulnerable to adversarial examples, we conducted an empirical evaluation of three QNNs in the face of ten state-of-the-art attacks. We observe that the average point distance to the decision boundary increases as the precision of a neural network decreases, which makes int-8 QNNs generally more robust than int-16 QNNs and full-precision ANNs (**RQ2**). However, quantization *per se* is not sufficient to protect neural networks from adversarial attacks. Although quantization makes it more difficult for attacks to optimize over the loss surface (**RQ2**), leading to the explosion or vanishing of the gradient estimated by some attacks, there are still attacks that can tackle this gradient masking phenomenon. Other attacks can subvert QNNs integrity through adversarial example transferability. Although adversarial example transferability is generally poor for most attacks, due to quantization-shift and gradient misalignment between full-precision ANNs and respective QNNs, there are still attacks that can subvert these issues. With effect,

in terms of adversarial example transferability quantization can minimize the effects of small perturbations but can also amplify bigger perturbations.

This thesis also constitutes an initial exploration into defensive strategies for TinyML systems. We empirically evaluated the efficacy of six state-of-the-art adversarial defenses, originally developed for full-precision ANNs, when applied to TinyML QNNs. Our results suggest that any defense effective for a full-precision ANN retains its effectiveness when deployed on the corresponding QNN (**RQ3**). However, the reverse does not hold. Input pre-processing techniques demonstrate considerable success in mitigating low-magnitude perturbations but lose efficacy as perturbation strength increases. In contrast, training-based defenses, namely PGD and Sinkhorn Adversarial Training, consistently increase the average point distance to the decision boundary, even after quantization. Importantly, not all defenses are suitable for deployment in TinyML systems (**RQ3**). Train-based defenses are preferable, as they do not lead to computation overhead during inference. Despite this, none of the evaluated defenses effectively addressed the challenge of adversarial example transferability from full-precision to integer-precision neural networks. We envision that developing train-based defenses, explicitly designed to smooth the quantization-shift and gradient misalignment, is the way to enhance the robustness of QNNs in TinyML applications.

Capsule Networks at the Edge. To assess the feasibility of deploying CapsNets on MCUs, we extended CMSIS-NN (Arm Cortex-M) and PULP-NN (RISC-V RV32IMCXpulp), with software kernels for the execution of quantized CapsNets. To fully exploit our software kernels, we developed a quantization framework that aligns with the quantization policies of these APIs. Results demonstrate that CapsNets can be quantized under these constraints with negligible impact on accuracy (**RQ4**). With effect, experiments on CapsNets trained on three distinct datasets (MNIST, CIFAR-10, and smallNORB) showed a maximum accuracy loss of 0.18% while registering a memory saving near 75%.

To evaluate the runtime performance of our software kernels, we considered CapsNets featuring small, medium, and large kernels for the primary capsule and capsule layers. They were deployed on three Arm Cortex-M and one RISC-V MCU. It is important to note that the classification of kernel sizes as small, medium, or large is relative to the memory limitations inherent to MCUs. Our evaluation includes MCUs with RAM sizes varying between 512 kilobytes and 1 megabyte. For the least performing MCU (STM32L4R5ZIT6U - Cortex-M4 @ 120 MHz), the execution of a primary capsule layer with a small (36 864 parameters) and medium (50 176 parameters) sized kernel varies between 93 and 500 milliseconds. In contrast, the most performant device (RISC-V RV32IMCXpulp @ 170 MHz) executed the same kernels in just 1 millisecond and 7 milliseconds, respectively. For capsule layers with small (12 800 parameters) and medium (192 000 parameters) kernels, the STM32L4R5ZIT6U required 80-268 milliseconds, while the RISC-V counterpart completed them in 11–38 milliseconds.

With these results, we believe that it is indeed feasible to deploy CapsNets on Arm Cortex-M and RISC-V PULP MCUs (**RQ5**). However, careful consideration must be given to the trade-off between the kernel size of the primary capsule and capsule layers and the performance specification of the target MCU. For example, deploying a CapsNet with a large primary capsule layer (100 352 parameters) on an

STM32L4R5ZIT6U might be infeasible, as this MCU would take more than 3 seconds to execute it. At the same time, the same operation completes in just 55 milliseconds on the RISC-V RV32IMCXpulp.

4.2 Future Work

While this thesis has presented significant contributions to enhance the privacy and integrity of TinyML systems based on MCUs, we recognize there is still room for further improvements. Future research should focus on refining and extending SecureQNN with a defense against adversarial examples and MIAs.

Refine SecureQNN adressing TF-M limitations. The current memory management strategy in TF-M statically partitions RAM and flash between the secure and non-secure worlds during the system's boot phase. While this static allocation approach is sufficient in scenarios where the QNN operates continuously, it proves suboptimal in use cases where neural network inference is intermittent. In such contexts, the static reservation of secure memory may lead to inefficient utilization of resources, thereby constraining the amount of software that can be deployed in the normal-world. This limitation is particularly relevant given the low memory resources typically available on MCUs. Enhancing TF-M with dynamic memory allocation for the secure-world post-boot would be beneficial for such a use case. As detailed in Section 2.2, the SAU unit of Arm TrustZone-M supports programmable memory partitioning, which can be leveraged to implement this feature.

Extend SecureQNN with a defense against MIAs. SecureQNN currently offers protection against white-box MIAs by obscuring the internal representations of sensitive layers, effectively mitigating attacks that rely on full model transparency. However, the framework does not address the risks posed by black-box MIAs, which, despite being less accurate than their white-box counterparts, still represent a credible threat to training data privacy. To provide a more comprehensive defense, SecureQNN could be augmented with techniques specifically designed to resist black-box MIAs. Among the defense mechanisms available for mitigating MIAs [142], defenses implemented during the model's pre-training or training phases are particularly suitable for constrained environments like Arm Cortex-M MCUs. Pre-training and training defenses imply the retraining of the QNN; however, they avoid additional inference-time overhead.

Develop a defense against adversarial examples and merge it with SecureQNN. Although PGD and Sinkhorn Adversarial Training do not impose additional overhead during inference on MCU platforms and consistently increase the average point distance to the decision boundary in QNNs, they remain insufficient to effectively counter adversarial examples in TinyML QNNs. Although the transferability of adversarial examples crafted on full-precision ANNs to QNNs is generally poor, some attacks, such as DeepFool or C&W, have shown to easily subvert the quantization-shift and gradient misalignment phenomena disrupted by quantization and successfully evade QNNs. Therefore, developing a train-based defense that smooths the quantization-shift and gradient misalignment phenomena disrupted by quantization might be

the way to enhance the robustness of QNNs. Incorporating such a defense into the SecureQNN framework would enable a comprehensive solution for secure QNN inference on MCUs, addressing both privacy and integrity concerns.

Extension of SecureQNN to other ML architectures. While our results validate the applicability of SecureQNN to QNNs tailored for image processing, extending its capabilities to encompass other ML architectures, such as audio models included in the MLPerf Tiny benchmark, would be a valuable add-on for TinyML systems. Supporting such models would require extending SecureQNN with generative models suitable for audio synthesis. Given the nature of the audio datasets included in the MLPerf Tiny benchmark, generative models based on WaveGAN [143] could be a good starting point. Furthermore, within the image processing domain, incorporating support for CapsNets would constitute a valuable enhancement to the framework's versatility. Considering that the amount of layers in a CapsNet is typically very low, extending SecureQNN to CapsNets would require sub-layer partitioning techniques, i.e., techniques to partition the kernel of the primary capsule and capsule layers and identify the privacy-criticality of each sub-kernel.

Extend the CapsNet quantization framework with prunning. Although our CapsNet quantization framework achieves a 75% reduction in model size, we believe that designing a lightweight CapsNet suitable for TinyML applications remains a significant challenge. To further support the deployment of CapsNets in resource-constrained environments, incorporating weight pruning techniques, such as those introduced by Kakillioglu et al. [68], may provide a complementary approach to quantization, facilitating more efficient CapsNet design within the TinyML context. Among pruning techniques, structured pruning should be prioritized, as it leads to tangible reductions in memory usage and computational overhead. Unlike unstructured pruning, which merely sets individual weights to zero without altering the underlying architecture, structured pruning removes entire neurons or filters. This results in a true reduction in kernel size and network complexity, thereby improving both memory efficiency and hardware compatibility. In contrast, unstructured pruning poses significant implementation challenges on MCUs, as it requires additional memory management strategies to represent and process sparse matrices efficiently. These overheads may negate the potential gains in resource savings, making structured approaches more viable for TinyML deployments.

Part II

Publications

List of Publications

Publications in this Thesis

[69] **M. Costa**, T. Gomes, and S. Pinto. "SecureQNN: Shielding the Intellectual Property of QNNs in TinyML Systems". [Submitted to IEEE Internet of Things Journal]

[70] **M. Costa**, and S. Pinto. "David and Goliath: An Empirical Evaluation of Attacks and Defenses for QNNs at the Deep Edge". In *IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. 2024.

[71] **M. Costa**, D. Costa, T. Gomes, and S. Pinto. "Shifting Capsule Networks from the Cloud to the Deep Edge". In: *ACM Trans. Intell. Syst. Technol.* (2022).

Other Contributions

[72] **M. Costa** et al. "SecureQNN: Introducing a Privacy-Preserving Framework for QNNs at the Deep Edge". In *Data Science and Artificial Intelligence*. 2023.

[73] D. Costa, **M. Costa**, and S. Pinto. "Train Me If You Can: Decentralized Learning on the Deep Edge". In: *Applied Sciences* (2022).

5

SecureQNN: Shielding the Intellectual Property of QNNs in TinyML Systems

This chapter was submitted as:

Miguel Costa, Tiago Gomes, and Sandro Pinto. “SecureQNN: Shielding the Intellectual Property of QNNs in TinyML Systems”. [Submitted to IEEE Internet of Things Journal]

Abstract

Building accurate ML models requires substantial expertise and large-scale datasets typically only available in big data companies. These companies have been selling their models as MLaaS. However, concerns about data privacy and the appliance of ML in mission-critical scenarios are forcing ML computation to move from the cloud to the deep edge, near sensor data. If edge ML increases user data privacy and makes decision latency predictable, deploying proprietary ML models on untrusted edge devices may harm the intellectual property of SP. A natural response to this issue comes from TEEs, which provide hardware-based security. However, adapting ML computation to the constraints of TEEs remains an open challenge. In this article, we propose SecureQNN, a framework that leverages state-of-the-art TEE technology (TrustZone-M) available in Arm Cortex-M MCUs to increase the protection of QNNs against unauthorized replication. SecureQNN evaluates which layers lessen the effort of an attacker when building a surrogate QNN with the same accuracy. The layers making the attacker spend less training epochs than the owner training from scratch are stored and executed in the secure-world of the TEE. Experiments demonstrate that SecureQNN undermines the cost-benefit of unauthorized QNN replication by isolating 51%–65% of the model size while incurring a worst-case decision latency overhead of only 0.06%. Although SecureQNN has broader applicability, the negligible impact on decision latency and its deterministic behavior highlight the suitability of SecureQNN for mission-critical and real-time applications.

5.1 Introduction

Developing accurate and efficient ML models requires substantial expertise and large-scale datasets, typically unavailable to ML clients. Consequently, this task is predominantly undertaken by major technology companies such as Amazon and Google, which sell pre-trained models via MLaaS [13, 16]. While MLaaS enables clients to utilize third-party models for inference on private data, it shows several limitations. Traditional MLaaS follows a centralized computing paradigm, requiring clients to transmit private data to cloud servers, necessitating trust in the server and the network infrastructure. Furthermore, frequent data transfers can congest networks and cloud computation units, leading to problematic delays for real-time applications, such as autonomous systems, medical diagnostics, and forest fire detection [5, 6, 73].

These growing concerns over data privacy and mission-critical ML, aligned with advances in memory and processing resources in low-power IoT devices led to the ongoing trend to shift ML computations to the edge of the network [5, 9, 11–13, 15–17, 73]. Although deploying ML models on user devices enhances privacy by eliminating the need to transfer user data to third parties, deploying such models on untrusted devices may damage the intellectual property of the SP. Unauthorized access to these models can expose proprietary algorithms by reverse engineering [33] or leak private training data [33–35]. Therefore, researchers from academia and industry have been investigating mechanisms to transfer a proprietary ML model to the edge while providing privacy guarantees to the model owner (SP).

The most prominent solutions to date rely on TEEs available on COTS APUs. TEEs offer built-in hardware primitives to enforce memory isolation between applications running in a REE and critical applications [43]. However, applications within a TEE must be as small as possible to minimize the TCB, requiring the redesign of ML systems.

In recent years, several TEE-based approaches have been proposed to increase the intellectual property protection of ML models in untrusted devices. Most works [9, 12, 13, 15] encrypt entire ANNs and store them in unprotected memory. To minimize the TCB, the ANN is loaded layer-by-layer from non-secure to secure memory. Within the TEE, the layer is decrypted and the activations are calculated. Despite their novelty, these works incur increased latency due to frequent cryptography operations and context switches between normal and secure worlds. More efficient works (i) perform model loading and prediction in parallel [36]; (ii) extend TEEs [17, 37], (iii) execute only strategic layers in the TEE [11, 38–40]; (iv) obfuscate strategic ANN weights [16]; or (v) refactor ANNs in a bident structure [41].

Despite these advancements, existing solutions primarily target x86 and Arm Cortex-A processors, neglecting Arm Cortex-M MCUs, projected to reach one trillion units by 2035 [18] and already powering most IoT applications. Recent support for SIMD for the execution of QNNs [18] and the availability of developer-friendly ML frameworks (CMSIS-NN) [18, 20] increased Cortex-M's relevance in ML research [24, 70, 144]. The introduction of TrustZone-M in Armv8-M further raises the question of whether TEEs can effectively protect or increase the protection of the intellectual property of QNNs on these resource-constrained devices.

Arm Cortex-M MCUs only hold a few megabytes of memory and deliver substantially lower throughput than Cortex-A and x86 processors. Therefore, porting works that perform full ANN decryption within the TEE [9, 12, 13, 15, 36] is inefficient due to the overhead of frequent cryptography operations. Cortex-M MCUs do not support GPU acceleration to compensate for this overhead. Refactoring an ANN in a bident structure [41] is also problematic, as Arm Cortex-M devices only feature a single core, not allowing the parallel execution of two different ANNs. Extending TEEs with custom MMUs [17, 37] is infeasible too, as Cortex-M lacks MMU support. Weight obfuscation [16] is more efficient but still imposes considerable overhead to denoise the activations of each ANN layer.

A more viable approach is executing only strategic layers within the secure-world while running the remaining layers in the normal-world. However, existing solutions in this category are also unsuitable for Cortex-M: Slalom [38] and ShadowNet [39] require GPUs, DarkneTZ [11] protects training data rather than the ANN itself, and SecureFrameNet [40] relies on FLOPs to balance the TEE workload, whereas Cortex-M typically runs ANNs in integer precision (QNNs). These limitations lead to a pivotal question: *can TEEs be leveraged to protect the privacy of QNNs on resource-constrained Arm Cortex-M MCUs?*

To answer this research question, we propose SecureQNN: a framework to determine the most critical layers of a QNN and protect their privacy using Arm TrustZone-M. Our framework traverses a given QNN layer-by-layer and determines which set of consecutive layers is privacy-critical and must be delegated to the secure-world. Our framework simulates the most challenging threat model, where the attacker has access to everything in the normal-world and can use the available information to recover the full model.

The attacker is also aware of the architecture of the private layers. In this scenario, the attacker can freeze the public layers and train a substitute model till it gets an accuracy equal to or higher than the accuracy of the original model. SecureQNN is designed to undermine the cost-benefit of unauthorized QNN replication. We consider a QNN to be sufficiently secure when the effort required by the attacker to train a substitute model is, at least, equal to the effort needed for the SP to train the original model from scratch. We measure the effort in training epochs. Additionally, SecureQNN leverages TF-M - a reference implementation of secure firmware for Cortex-M MCUs - to protect the integrity of QNNs and prevent the success of highly advanced attacks probing the flash memory.

We evaluated SecureQNN using QNNs and datasets from the MLPerf Tiny benchmark [24], which reflects real-world TinyML applications identified by over 30 academic and industrial organizations. Our evaluation focuses on two key metrics: (i) TEE memory footprint and (ii) inference latency. Results show that the cost-benefit of unauthorized QNN replication can be undermined by delegating only a small fraction of the QNN layers to the secure-world. Regarding latency, measurements conducted on an STM Nucleo-L552ZE-Q board, featuring an Arm Cortex-M33 at 110 MHz, indicate that SecureQNN introduces negligible overhead, maintaining deterministic and predictable inference performance suitable for real-time edge applications. All artifacts are open-sourced to enable independent validation of results and encourage further exploration of techniques to protect the intellectual property and privacy of QNNs deployed in edge devices[1].

In short, this work makes the following contributions:

1. The first public and open-source framework aimed at increasing the protection of the intellectual property of QNNs deployed in Arm Cortex-M MCUs;

2. A novel strategy to determine the set of privacy-critical layers in an ANN/QNN;

3. The concept of default privacy setting, whereby the intellectual property of an ANN/QNN is considered sufficiently secure when attackers have to pay the same effort as the model owner training from scratch to get a surrogate ANN/QNN with equal or higher accuracy, thus reducing the cost-benefit and likelihood of successful model replication by third parties.

Our framework (SecureQNN) was first introduced as a conference perspective paper [72]. This earlier publication [72] focused on (i) analyzing existing literature, (ii) highlighting the importance of the topic, (iii) defining the threat model, and (iv) providing a high-level overview of SecureQNN. In this new publication, we expand on the initial work significantly. In addition to the extension and refinement of previous topics, the current paper:

1. Describes the strategy to simulate the creation of a surrogate dataset by a powerful attacker (Section 5.4.1) and presents a comprehensive qualitative and quantitative evaluation of the surrogate datasets (Section 5.5.1);

[1] `https://github.com/MiguelCosta94/SecureQNN.git`

2. Extends the model partitioning strategy with new functionality to achieve a finer trade-off between TEE memory footprint and privacy (Section 5.4.2);

3. Restructures the Trusted Execution module to address initial design flaws and introduces protection against flash-probing attacks (Section 5.4.3);

4. Evaluates the TEE memory footprint across different privacy levels and different development boards (Section 5.5.2);

5. Analyzes the decision latency (Section 5.5.3), including the latency of the context switches and the overhead resulting from the protection against flash-probing attacks;

6. Discusses the current limitations of the framework (SecureQNN) and potential directions for future work (Section 5.6).

5.2 Methodology

The methodology guiding the design and evaluation of SecureQNN is grounded in two core aspects: the framework scope and the threat model. The scope defines the target hardware computing platform and software libraries leveraged by our framework, while the threat model formalizes the adversarial assumptions and security goals.

5.2.1 Scope

This work targets ultra-low-power IoT devices powered by Arm Cortex-M MCUs following the latest Armv8-M and Armv8.1-M architectures and supported by TF-M. These devices typically feature a single core, alongside flash and SRAM. Both flash and SRAM sizes vary between hundreds of kilobytes and a few megabytes. These MCUs execute instructions directly from physical memory and do not feature an MMU for virtual memory support, making them incapable of running Unix-like OSs. While these MCUs offer various peripherals, they do not support GPU acceleration. The most advanced Armv8-M and Armv8.1-M MCUs feature TrustZone-M technology, allowing developers to establish two execution environments: secure and normal.

CMSIS-NN and TensorFlow Lite Micro, built on top of CMSIS-NN, are state-of-the-art libraries for running QNNs on Arm Cortex-M MCUs. CMSIS-NN is accelerated by SIMD for matrix and convolution operations with 8-bit and 16-bit operands. In our framework, we use CMSIS-NN instead of TensorFlow Lite Micro to reduce the memory footprint of the secure-world. CMSIS-NN supports int-8 and int-16 QNNs with fixed bit-width and following a (i) uniform, (ii) static, and (iii) symmetric quantization. For more details on the CMSIS-NN library, we refer readers to work [18].

5.2.2 Threat Model

Figure 8 summarizes the threat model of SecureQNN. It encompasses two agents: (i) the SP or model owner and (ii) an honest but curious client.

Service Provider (SP). The SP uses its capacity to store and process data to train a proprietary ML model. The trained model constitutes the intellectual property of the SP and the SP is interested in maintaining it secret. In addition, the SP is also concerned with the privacy of the training data. The training data could be generated by the SP itself, or provided by a third party under a given confidentiality agreement. Therefore, leaking training data would not only undermine users' trust in the SP but could also put the SP in legal action raised by the data owner. Considering that recent studies have shown that ML models are vulnerable to the so-called MIAs - attacks specifically designed to understand if a given data point belongs to the training dataset - the SP is also interested in reducing the success rate of this type of attack to keep the training data private. Storing and running the entire QNN in the secure-world is not feasible. Therefore, the SP deploys the ANN in clients' devices as a gray-box, where part of it is executed within the secure-world and another in the normal-world. The SP attests to the privacy of the model when the effort (number of epochs) required by the attacker to construct a surrogate model with an accuracy equal to or higher than the accuracy of the original model is, at least, equal to the effort of the SP to train the original model from scratch.

Client. The client gets the QNN owned by the SP on its local device powered by an Arm Cortex-M MCU as long as it supports the Arm TrustZone-M and TF-M technologies. The client is an honest but curious adversary: they follow the protocol specified by the SP but are interested in learning about the QNN. Regarding capability and knowledge, we assume a worst-case scenario. The adversary can manipulate everything that runs in the normal-world, including the parameters of public QNN layers and the OS. Also, the client is aware of the architecture of private layers and the training hyper-parameters. Except for the last QNN private layer, which is typically a fully-connected layer with softmax activation, in a real-world scenario,

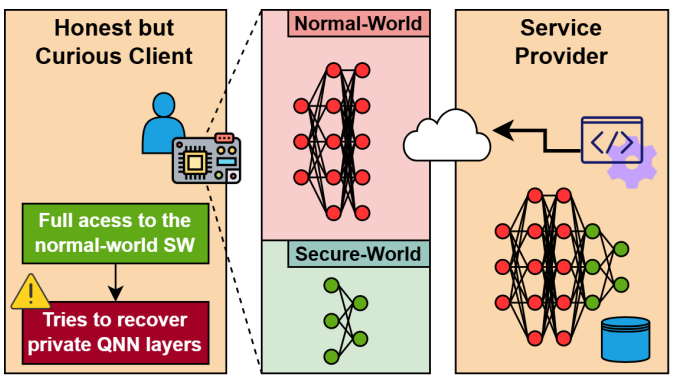


Figure 8: SecureQNN threat model: the client of the SP is honest but curious, trying to recover the private QNN layers.

it is hard to access the architecture of private layers and training hyper-parameters. Nevertheless, TEEs are not bulletproof [145], and, therefore, a worst-case scenario increases the level of privacy protection.

An advanced adversary can also probe the flash memory to retrieve the content stored in secure memory, and therefore, the content of private QNN layers. However, this type of attack is extremely difficult to perform in modern SoCs and requires very expensive equipment. Other physical attacks are out of the scope of our work. The client has access to the original testing dataset but not the training data. Considering that data is the most valuable asset of any ML model, if the attacker already has access to the training data, there is no interest in stealing the model.

5.3 Background and Related Work

This section summarizes the concepts and key technologies that support the development of SecureQNN. We begin by discussing Arm TrustZone and its adaptation for MCUs (TrustZone-M). We then explore TinyML, a growing field focused on deploying ML on ultra-low-power devices, and delve into Generative Adversarial Networks (GANs), a key ML technology to simulate the creation of a surrogate dataset by a potential attacker. We end with a comprehensive literature review on solutions addressing the topic of intellectual property/privacy of ML models on edge devices and how our work advances the state-of-the-art.

5.3.1 Arm TrustZone(-M)

Arm TrustZone was introduced in 2004 as a hardware security extension of Arm APUs (Arm Cortex-A) [42]. TrustZone follows a system-wide approach to security, providing hardware-enforced protection mechanisms at the CPU and SoC levels. Initially deployed in the Armv6K architecture, this technology imposed important architectural changes, introducing two protection domains: (i) the secure-world and (ii) the normal-world [42, 43]. The secure-world hosts privacy- and integrity-critical software components, while the normal-world runs less sensitive applications. At a given moment, the processor operates exclusively in one world. In Arm Cortex-A processors, the CPU status is defined by the NS bit of the SCR [42, 43]. The context switch is handled by a privileged software mechanism called Secure Monitor, which saves the state (registers, stack pointers, etc.) of the current world before switching to the other [42, 43].

In 2016, TrustZone was adapted to the new generation of Arm MCUs, namely those based on Armv8-M and Armv8.1-M architectures [43]. The newer technology, Arm TrustZone-M, follows the same dual-world architecture, with non-secure software blocked from directly accessing secure components. However, at the low level, there are significant changes as Cortex-M is optimized for faster context switch and low-power applications [42, 43].

In contrast to Arm Cortex-A APUs, in Armv8-M and Armv8.1-M the security state does not depend on a specific security bit. The division between the normal and secure worlds is memory-based: when the MCU operates from secure memory, it is in a secure state, and when operating from non-secure memory, it is in a normal state [42, 43]. To determine the secure state of the MCU, TrustZone-M relies on two distinct

units: (i) the SAU and (ii) the IDAU [42, 43, 45]. The former is programmable, enabling dynamic memory partitioning; the latter sets static partitions determined by the chip manufacturer. TrustZone-M determines the security status of an address by performing a logical OR on the outputs from the SAU and IDAU. In contrast to Cortex-A, Armv8-M and Armv8.1-M do not use a secure monitor software to perform the context switch [42, 43]. The bridge between both worlds is handled by a set of mechanisms implemented into core logic. For more information about Arm TrustZone-M, we refer readers to works [42, 43, 45].

5.3.2 TinyML: The new frontier in AI?

What is TinyML? TinyML is a branch of ML that focuses on hardware and software solutions that enable the deployment of ML in devices consuming 1 milliWatt or less [24], such as ultra-low-power MCUs. TinyML emerged as a response to the increasing concerns about data privacy, but also to the cloud and network limitations disrupted by the boom of IoT devices.

TinyML in the Market. Arm predicts that by 2035, MCUs could surpass 1 trillion across multiple market segments [18]. In the same line, a market study [146] projects that the MCU market size is expected to reach $992.80 million by 2030. Regarding the TinyML market, ABI Research forecasts that it will expand from 15.2 million shipments in 2020 to 2.5 billion by 2030 [31] and TinyML Software-as-a-Service could potentially become a billion-dollar market by 2030 [30].

TinyML in Academia. In academia, multiple software and hardware solutions have been proposed to enable and optimize TinyML. From the myriad of proposed solutions, TinyML has played a pivotal role in the increasing relevance of RISC-V ISA. Flamand et al. [21] proposed GAP8 - a SoC tailored to optimize the execution of QNNs - competing with the already established Arm Cortex-M MCUs. This SoC became so relevant that GreenWaves Technologies started mass production and even proposed a newer iteration (GAP9) [22]. To further support the adoption of GAP8, Garofalo et al. [19] developed PULP-NN, a software library optimized for the execution of 8-bit QNNs. AskariHemmat et al. [147] went a step further and proposed a RISC-V vector processor tailored to perform inference using sub-byte QNNs. Hoyer et al. [148] brought these innovations to the real-world, proposing a RISC-V device running QNNs to monitor and detect heart arrhythmia.

TinyML Use Cases. In terms of use cases, TinyML systems were initially designed for (i) word spotting, (ii) object recognition, (iii) object counting, and (iv) audio/voice detection [24, 30]. Recently, TinyML systems have been extended to (i) forest fire detection, (ii) health monitoring, (iii) anomaly detection, and (iv) smart pharming, among others [30]. For more information on the relevance and use cases of TinyML, we refer the readers to surveys [26, 27, 149].

5.3.3 Generative Adversarial Networks

GANs are a class of deep learning frameworks commonly used to generate new data that closely resembles real-world data. Proposed by Goodfellow et al. [150], a typical GAN comprises two neural networks, the generator and the discriminator, which are trained concurrently through an adversarial process.

The generator receives, as input, a random vector sampled from a latent space, typically defined by a normal or uniform distribution. This network transforms the input vector into data that resembles real-world data, represented by the training dataset [150–153]. The latent space, from which the random vector is drawn, is generally lower-dimensional compared to the space of the actual data. The goal of the generator is to learn how to map the different points composing the latent space to real data.

Conversely, the discriminator is a binary classifier that evaluates whether a sample is real (from the training dataset) or synthetic (crafted by the generator) [150–153]. The training is termed adversarial because the two neural networks contest each other in a zero-sum game, where the gain/success of the generator is the loss/failure of the discriminator and vice versa. Figure 9 depicts the training process of a typical GAN [150–153].

As observed in Figure 9, the generator does not access the training dataset. Instead, it learns by generating data and receiving feedback from the discriminator, which accesses original and synthetic samples. During training, both networks are optimized through backpropagation. The training ideally ends when the generator fools the discriminator half of the time, indicating that the discriminator can not reliably differentiate between real and synthetic data (50% accuracy) [150–153].

5.3.4 Related Work

In this section, we review and analyze the most significant works proposed to date for protecting the intellectual property and/or privacy of ML models deployed in untrusted devices. The discussion is structured according to the specific strategies employed by these works.

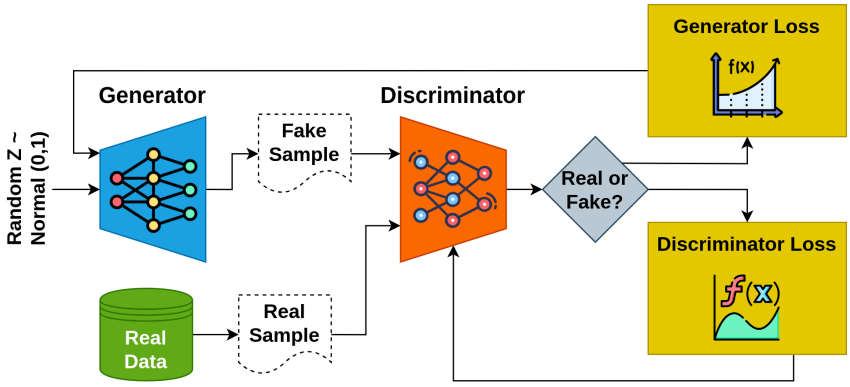


Figure 9: GAN training: the parameters of the generator and discriminator are updated until the discriminator can not distinguish fake from real samples.

Full ANN on TEE. Bayerl et al. [12] propose to deploy the whole ANN at a non-secure memory region as cyphertext. As the ML system starts, the full ANN is loaded to the secure-world, where it is decrypted and stored in secure memory. The inference runs entirely within the secure-world. The framework proposed by Bayerl et al. [12] - OMG - targets Arm Cortex-A processors supporting SANCTUARY security architecture. Despite the simplistic approach, OMG does not scale well as the whole ANN has to fit within the secure memory, which tends to be very small.

The works proposed by VanNostrand et al. [9] and Hanzlik et al. [13] reduce the TEE memory footprint by deploying the ANN to the TEE layer-by-layer. As the ML system starts, the parameters of the first layer are loaded into the TEE, where they are decrypted and stored in a secure memory buffer. As the APU calculates the outputs of the first layer, the application repeats the previous process for the following layer till it reaches the last one. This strategy significantly reduces the memory footprint of the TEE, which only has to guarantee enough memory to save the largest layer instead of the whole ANN. These works differ in the target SoC ISA - while MLCapsule [13] targets Arm Cortex-A APUs with support to OP-TEE, the work of VanNostrand et al. [9] target x86 APUs with Intel-SGX. Nakai et al. [36] improved over MLCapsule by using the shared cache memory between the normal and secure worlds to perform parameter loading in parallel with the parameter decryption and inference, thereby reducing the decision latency.

Liu et al. [15] followed a disruptive approach and proposed a framework (SecDeep) to reformulate Arm-NN, an API for ML workloads in Arm Cortex-A processors. Their analysis revealed that over 90% of the Arm-NN code is dedicated to tasks such as tensor preparation and performance optimization, while less than 10% of the code is involved in tensor computations that handle private/sensitive model or user data. Building on this insight, SecDeep partitions the Arm-NN API into two categories: confidential and non-confidential functions. Confidential functions are executed in the secure-world, while non-confidential functions are assigned to the normal-world.

TEE Extension. Xie et al. [17] also target Arm Cortex-A APUs, but instead of adapting ML models or libraries to fit within the constraints imposed by TEEs, they enhanced memory management within TEEs to support ANN inference more efficiently. Xie et al. [17] introduce a novelMMU (Smart-Zone) that dynamically adjusts the secure memory size (number of page tables in virtual memory and the secure memory region in physical memory) required by a given ANN. In addition, the authors propose Tinylib, a lightweight library that optimizes ML operations within the secure-world, while maintaining a small TCB.

Within the same line, Yang et al. [37] extended the Intel-SGX technology of x86 processors with an MMU tailored for ML computation. In addition, the authors propose a framework to decompose an ANN and set optimal parameter blocks for each layer, minimizing the workload within the TEE.

Model Refactoring. Liu et al. [41] introduced MirrorNet, a framework to refactor an ANN into two ANNs, one executed within the normal-world and another within the secure-world. The refactored ANNs follow the architecture of the original model, but the version running within the secure-world has smaller

kernels while contributing the most to the final decision. By its side, the model deployed in the normal-world must not be accurate enough to enable an attacker to reconstruct the original model. The authors tested MirrorNet on an Arm Cortex-A processor featuring Arm TrustZone.

Weight Obfuscation. Hou et al. [16] propose to obfuscate a given percentage of weights in descent order of their magnitude, i.e. contribution to the final decision. With this design approach, the inference process runs almost entirely outside the TEE, being the enclave responsible to denoise the activations of each layer. For any input sample, the crafted noise must be enough to change the predicted label. The authors consider the ANN intellectual property as properly protected when a possible attacker has to pay the same machine training hours to recover the original model as the model owner training from scratch. Hou et al. [16] tested the concept on an x86 APU with Intel-SGX.

Selected Layers on TEE. Slalom [38] is among the first works to delegate only a restricted set of layers to the TEE. For this purpose, Slalom splits an ANN between linear and non-linear layers. The inference is managed from the enclave, where the non-linear layers are executed. For linear layers, Slalom encrypts the input data and layer parameters, outsourcing the computations to a co-located GPU. Once the GPU completes the computation of a linear layer, the activations are decrypted within the enclave, and their integrity is verified using Freivalds' algorithm. Slalom is designed for x86 processors equipped with Intel-SGX. Building on this concept, Sun et al. [39] proposed ShadowNet, a similar framework optimized for Arm Cortex-A processors that support OP-TEE.

The SecureFrameNet framework [40] builds upon model pruning techniques to establish a metric for quantifying the contribution of each neural network layer to the final decision. The importance score of each layer is then used to partition the ANN into two or more subgraphs. During the segmentation process, SecureFrameNet also evaluates the FLOPs associated with each subgraph. SecureFrameNet targets Arm Cortex-A processors with support to pKVM.

Mo et al. [11] proposed a framework with a slightly off goal but still related to the intellectual property of ML models. Instead of preserving the privacy of ANN parameters like previous works, DarkneTZ seeks to protect the privacy of training data by offering increased robustness against MIAs. DarkneTZ traverses the model layer-by-layer, starting from the last layer, to determine which layers contribute the most to the success of a powerful white-box MIA. This set of layers is delegated to the secure-world. DarkneTZ targets Arm Cortex-A processors with support to OP-TEE.

5.3.5 Gap Analysis

Table 9 puts our work in perspective with previous works. As observed, previous research focuses on Arm Cortex-A APUs with TrustZone and x86 APUs supporting Intel-SGX. To the best of our knowledge, our framework is the first to account for the constraints of Arm Cortex-M MCUs and TrustZone-M.

	Target Device			Trusted Execution	
	Architecture	*TEE*	*TinyML*	*Category*	*Summary*
VanNostrand et al. [9] 2019	Arm Cortex-A	OP-TEE	No	Full-ANN	Saves the ANN as cyphertext in non-secure memory. Loads the model layer-by-layer to the TEE.
OMG [12] 2020	Arm Cortex-A	SANCTUARY	No	Full-ANN	Saves the ANN as cyphertext in non-secure memory. Loads the model to the TEE in a single iteration.
MLCapsule [13] 2021	x86	Intel-SGX	No	Full-ANN	Saves the ANN as cyphertext in non-secure memory. Loads the model layer-by-layer to the TEE.
Nakai et al. [36] 2021	Arm Cortex-A	OP-TEE	No	Full-ANN	Uses the shared cache between the REE and the TEE to load the model in parallel with prediction.
SecDeep [15] 2021	Arm Cortex-A	OP-TEE	No	Full-ANN Conf. Functions	Arrm-NN functions dealing with user and model data and are delegated to the TEE.
Xie et al. [17] 2024	Arm Cortex-A	Arm TrustZone	No	Full-ANN TEE Extension	Custom MMU/API for ML computation within TEE.
Yang et al. [37] 2024	x86	Intel-SGX	No	Full-ANN TEE Extension	Custom MMU for ML computation within TEE.
MirrorNet [41] 2023	Arm Cortex-A	Arm TrustZone	No	Model Refactoring	Refactors 1 ANN in 2 ANNs. The ANN within the TEE contributes the most to the classification task.
Hou et al. [16] 2022	x86	Intel-SGX	No	Weight Obfuscation	Obfuscates strategical weights and denoises the output of each layer within the TEE.
Slalom [38] 2019	x86	Intel-SGX	No	Selected Layers	Offloads the linear layers from the TEE to the GPU.
DarkneTZ [11] 2020	Arm Cortex-A	OP-TEE	No	Selected Layers	Offloads layers that potentiate MIAs to the TEE.
ShadowNet [39] 2023	Arm Cortex-A	OP-TEE	No	Selected Layers	Offloads the linear layers from the TEE to the GPU.
SecureFrameNet [40] 2024	Arm Cortex-A	N.A.	No	Selected Layers	Uses pruning-based techniques to identify the layers that should be offloaded to the TEE.
Costa et al. 2024	**Arm Cortex-M**	**TF-M**	**Yes**	**Selected Layers**	**Offloads layers that lessen the effort of an attacker to build a surrogate QNN to the TEE.**

Table 9: Gap analysis: privacy of machine learning models.

Full ANN on TEE. Deploying the ANN layer-by-layer to the secure-world [9, 13, 36] reduces the TEE memory footprint by requiring memory allocation only for the largest layer. However, this approach requires a cryptographic operation for each layer upon receiving an input sample. Arm Cortex-M MCUs not only feature much lower computing power than Cortex-A and x86 APUs but also lack GPU acceleration and shared caches between normal and secure worlds to compensate for the additional overhead.

SecDeep (designed for Arm Cortex-A) follows a disruptive approach, by delegating only to the secure-world the functions that deal with user or ANN data - which accounts for only 10% of the Arm-NN API. However, CMSIS-NN is already fully optimized and does not include functions for tensor preparation or hardware configuration like Arm-NN, making SecDeep not appropriate for Cortex-M MCUs.

TEE Extension. Extending the TEE via a custom MMU for ML computation [17, 37] is infeasible on Cortex-M MCUs, as these devices lack MMUs (Section 5.2.1). While extending the TEE for ML workloads presents an avenue for future research, existing approaches for Cortex-A [17] and x86 processors [37] are not directly portable.

Model Refactoring. The model refactoring strategy proposed by Liu et al. [41] is hardly portable to devices powered by Arm Cortex-M MCUs as they are short in memory and do not feature more than one core to be able to store and run two ANNs in parallel.

Weight Obfuscation. The weight obfuscation technique proposed by Hou et al. [16] is among the most efficient to date. However, it still introduces computational overhead for denoising the activations/output of each ANN layer.

Selected Layers on TEE. Slalom [38] and ShadowNet [39] are not portable to Arm Cortex-M MCUs due to reliance on GPU acceleration. Nevertheless, storing and executing strategic layers on the TEE is a viable approach to increasing intellectual property protection of QNNs in Cortex-M MCUs without incurring computation overhead. To trade off TEE memory footprint and decision latency, our framework (SecureQNN) follows a strategy similar to SecureFrameNet [40] and DarkneTZ [11]. Nevertheless, while SecureFrameNet perceives the criticality of each layer based on the relevance to the final decision, our framework perceives layer criticality in terms of how it lessens the effort of a possible attacker in constructing a surrogate model with equal or higher accuracy. Additionally, SecureFrameNet relies on the number of FLOPs to manage the TEE computational load, a metric incompatible with the integer-precision execution typical of Cortex-M MCUs. DarkneTZ, in contrast, primarily addresses the privacy of the training data (MIAs) rather than ANN privacy.

5.4 SecureQNN

Figure 10 details the high-level architecture of SecureQNN, composed of three main modules executed in the following order: (i) Surrogate Dataset Generator, (ii) Model Partitioning, and (iii) Trusted Execution. The Surrogate Dataset Generator relies on deep generative models to simulate the creation of a surrogate dataset by a possible attacker. This dataset is then used as input for the Model Partitioning module, which simulates a series of training sessions performed by a powerful attacker to determine the privacy-critical

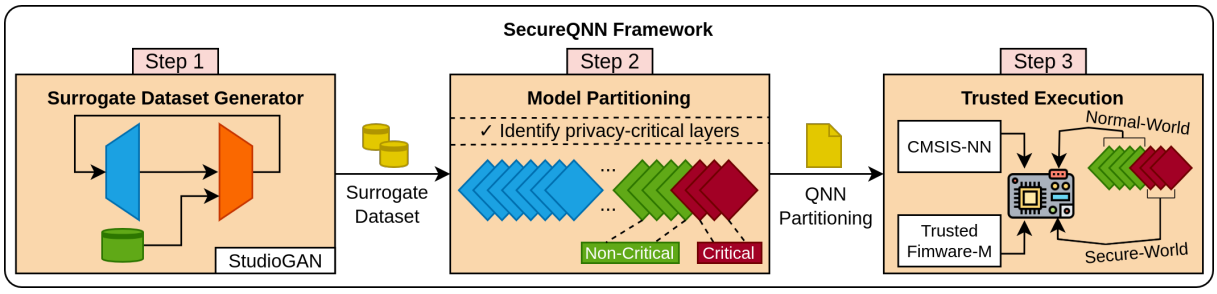


Figure 10: High-level architecture of SecureQNN. The Surrogate Dataset Generator simulates the generation of a training dataset by a potential attacker. The Model Partitioning divides the QNN into critical and non-critical layers. The Trusted Execution deploys the C code on the target MCU.

layers. The resulting QNN partitioning setting is then used as input for the Trusted Execution module, which generates the C code that provisions an edge device powered by an Arm Cortex-M MCU with SecureQNN.

5.4.1 Surrogate Dataset Generator

A potential attacker lacks access to the original training dataset. Nevertheless, they can spoof sensors or network connections to steal inference data and apply data augmentation techniques to build a surrogate training dataset. We simulate this scenario using deep generative models for image generation upon the original test dataset.

Deep Generative Model Architecture. To determine the most suitable model architecture for our use case, we referenced a study conducted in 2022 [154], which benchmarked: (i) Generative Adversarial Networks (GANs), (ii) Energy-Based Models (EBMs), (iii) Variational Autoencoders, (iv) Autoregressive Models, and (v) Normalizing Flows. These models were benchmarked on the CIFAR-10 dataset, which we also use to evaluate our framework. Among others, the study reports the Fréchet Inception Distance (FID), a widely supported and adopted metric for the quantitative evaluation of deep generative models for image generation [153, 155].

Results indicate that GANs and EBMs perform notably better than the other architectures. To further refine our choice, we evaluated the availability of these models through a unified programming framework. Ultimately, we chose GANs because of the availability of StudioGAN [153], a Python library that supports experimentation with state-of-the-art GAN architectures using both default and custom datasets. The authors of StudioGAN also provide a comprehensive benchmark for different GAN architectures and configurations on different datasets and even propose a training protocol based on empirical experimentation. To minimize the engineering effort in the development of the Surrogate Dataset Generator, we relied on the benchmark tables provided by the authors of StudioGAN and followed their training protocol.

GAN Evaluation Backbone. The evaluation backbone is a pre-trained ANN that processes real images (from the training dataset) and synthetic images (generated by the GAN). Its purpose is to extract a feature set used to compute metrics that quantify the quality of the generated images, such as the FID. StudioGAN supports and benchmarks three evaluation backbones: (i) TensorFlow-InceptionV3, (ii) PyTorch-SwAV, and (iii) PyTorch-Swin-T. To identify the most suitable evaluation backbone for our use case, we focused our analysis on the FID scores obtained for the CIFAR-10 and ImageNet datasets. StudioGAN's benchmark indicates that PyTorch-SwAV holds the best results.

GAN Architecture - CIFAR-10. For CIFAR-10, we have to select a technique for data-efficient training because the attacker only has access to a small dataset (the CIFAR-10 test dataset only contains 10k samples). Without data-efficient training, the discriminator memorizes the training dataset, making the training collapse. From the GAN architectures benchmarked in StudioGAN, we only considered options

with data augmentation or LeCam regularization - the two data-efficient training techniques supported by StudioGAN. The last design parameter to consider is the training backbone, i.e. the architecture of the generator and discriminator (GAN architecture). For CIFAR-10, StudioGAN benchmarks two architectures coupled with data-efficient training: (i) StyleGAN and (ii) BigGAN. To maximize the diversity of the surrogate dataset, we selected one GAN from each training backbone. The results detailed in Section 5.5.1.1 led us to integrate a StyleGAN2 and a standard BigGAN in our framework, both coupled with Differential Augmentation. The former returned an FID of 0.91 and the later returned an FID of 1.59.

GAN Architecture - VWW. Since StudioGAN does not benchmark GANs tailored for the VWW dataset or its resolution (96x96), we adapted configurations from other datasets and applied an image resizing filter. Among the image resolutions supported by StudioGAN, 128x128 is the closest to the original VWW resolution, being 1.33x higher. This is the resolution of the ImageNet dataset, whose GAN configurations were used as a basis for our implementation on VWW. For image resizing, we applied a bicubic filter, as advised in the training protocol of StudioGAN. StudioGAN benchmarks two GAN architectures on ImageNet: (i) BigGAN and (ii) StyleGAN. Nevertheless, we ended up including two variations of StyleGAN (StyleGAN2 and StyleGAN3-t) in our framework as we were not able to get good quality images from BigGAN models (Section 5.5.1.2). StyleGAN2 returned an FID of 4.17, while StyleGAN3-t returned an FID of 7.05.

5.4.2 Model Partitioning

The layers of an ANN do not contribute equally to the classification task [156–158]. While early layers contribute the most to decomposing the input data, later layers contribute the most to the final prediction. Building on this insight, we design a method to partition a given QNN into secure and normal worlds, aiming to achieve a good trade-off between TEE memory footprint and inference latency. For this purpose, we propose to divide the QNN into two blocks only - one to be stored and executed within the normal-world and another to be stored and executed within the secure-world.

Saving the whole QNN in non-secure memory, using cryptography to protect the privacy of critical layers, similarly to previous works, is the best strategy to reduce the memory footprint of the secure-world. Under this strategy, the secure-world only has to feature enough memory for the parameters of one layer at a time. However, this strategy does not port well to Arm Cortex-M MCUs as cryptography operations are too computationally expensive compared to Arm Cortex-A and x86 processors. This aligned with the increased context switches jeopardizes the inference latency. Nevertheless, by giving preference to deploy later layers to the secure-world instead of early ones, we are still able to significantly reduce the TEE memory footprint while also protecting the most critical layers to the classification task [156–158]. Later layers tend to feature significantly smaller activation vectors, which inherently reduces the size of intermediate buffers between layers. Figure 11 depicts the workflow of the Model Partitioning module. This module is developed upon the TensorFlow v2 API as it enables a seamless conversion of floating-point ANNs to integer precision QNNs compatible with TensorFlow Lite Micro and CMSIS-NN - the state-of-the-art

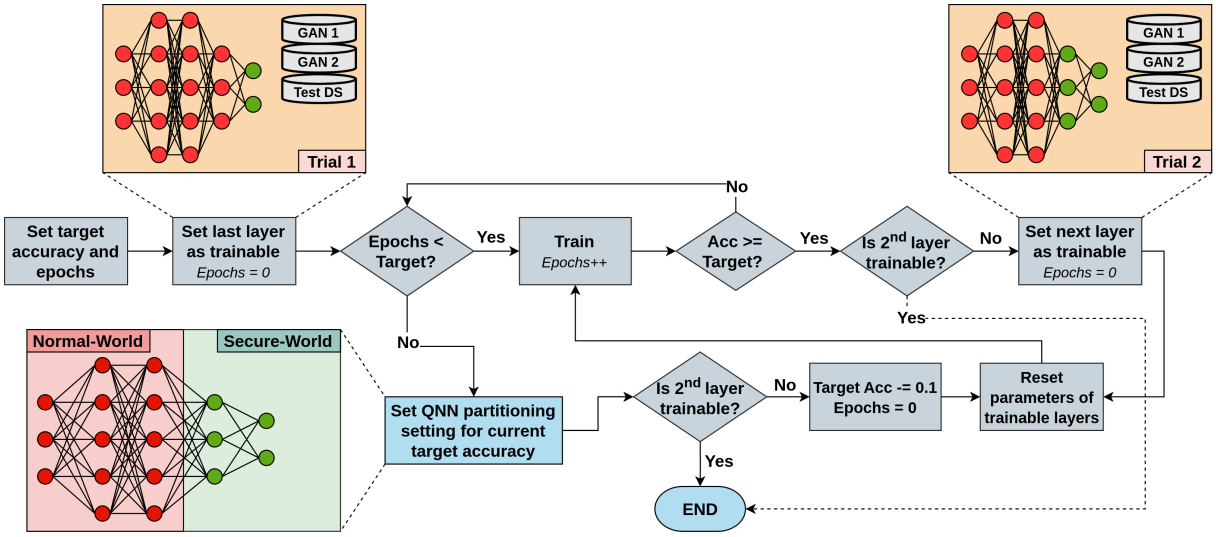


Figure 11: Workflow of the Model Partitioning module. It stops adding layers to the secure-world when the accuracy of the surrogate QNN fails to reach the target within the same training epochs as the original QNN. Red highlights layers for the normal-world, while green highlights layers for the secure-world.

APIs for ML computation on Arm Cortex-M.

Our framework analyzes the QNN layer-by-layer, starting from the final layer and moving backward. For each new layer, our framework performs a training process where the parameters of the new layer and the forwarding ones are set as trainable and updated at each epoch, while the parameters of the other layers are frozen (non-trainable). For training the surrogate QNN, our framework employs a dataset merging the original test dataset with the synthetic datasets produced by the two GANs. The training ends under one of two conditions: (i) when the accuracy of the surrogate QNN reaches or exceeds that of the target QNN, or (ii) when the number of training epochs matches that of the target QNN, but the accuracy remains lower. In the first scenario, the framework sets one more layer as trainable and repeats the training process. Every time the framework sets one more layer as trainable, the parameters of all trainable layers are reinitialized to a truncated normal distribution to avoid overfitting. The second scenario means that the framework has found a QNN partitioning setting that requires the attacker to pay the same effort to reconstruct the QNN as the SP training from scratch. The final set of trainable layers are the ones that must be delegated to the secure-world. We assume a worst-case scenario where the adversary knows the architecture of the whole QNN, including the private layers, and training hyperparameters.

The scenario where SecureQNN delegates to the secure-world the minimum set of layers to undermine the cost-benefit of unauthorized QNN replication is termed as the default privacy setting. Under this scenario, our framework sets the accuracy of the surrogate QNN to just below the accuracy of the original QNN. After determining the set of layers to comply with the default privacy setting, our framework reduces the target accuracy of the surrogate QNN in buckets (10% by default), repeating the process previously described (Figure 11). SecureQNN keeps reducing the target accuracy of the surrogate QNN until all layers but the first one is considered known to the attacker. This enables the SP to trade-off between increased privacy and memory allocated to the secure-world.

5.4.3 Trusted Execution

Figure 12 depicts the architecture of an ML system provisioned by SecureQNN. As illustrated, featuring Arm TrustZone-M and supporting TF-M are the minimum requirements for an MCU to support our framework. TrustZone-M establishes two separate execution environments, with each environment supporting privileged and non-privileged execution levels. The privileged-level blocks are colored in gray, while the non-privileged blocks are color-coded as green for the secure-world and red for the normal-world. TF-M is used as the reference implementation of the secure firmware.

Secure Boot. Within the privileged level of the secure-world, Secure Boot is mandatory as it verifies the digital signature of the firmware image during the device startup. Anytime the signature is corrupted, the boot is halted, preventing the execution of malicious code. Therefore, Secure Boot is essential to detect any tampering attempt of the QNN along its private and public layers. As a reference for the implementation of a secure boot, TF-M uses MCUboot.

Firmware Update. Also in the privileged level of the secure-world, Firmware Update is an optional service of TF-M useful if the SP updates the QNN over-the-air. This service ensures that whenever new firmware is downloaded to the device, it is only installed if its digital signature is valid. Additionally, Firmware Update includes an anti-rollback mechanism to prevent attackers from installing an older, potentially vulnerable version of the firmware and QNN.

Secure-World SW. In the non-privileged level of the secure-world, SecureQNN relies on two primary TAs. Crypto is an optional service of TF-M and provides cryptographic functionalities, including symmetric/asymmetric decryption, which enables our framework to tackle flash probing attacks tailored at stealing the parameters of private layers. If turned on, the parameters (weights and bias) of private layers are saved in a secure region of the flash memory as ciphertext. During inference, the Crypto service decrypts the parameters layer-by-layer as required. The decrypted parameters are stored in a single SRAM buffer, which is erased as the parameters are no longer needed for inference on the current sample. We store the parameters of private layers in secure memory even when TF-M Crypto is active to reduce the

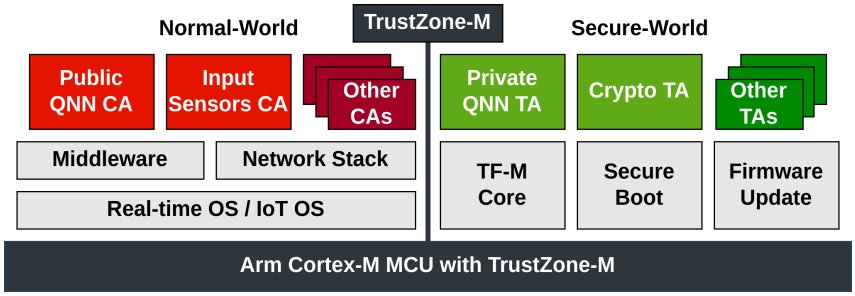


Figure 12: Architecture of an ML system provisioned by SecureQNN. Gray highlights privileged-level software. Green/red highlights non-privileged software within secure/normal worlds.

number of context switches and not jeopardize the inference latency. The Private QNN TA computes the output of the private layers. Whenever this TA is called, it loads the parameters directly from the flash, if they are stored in plain text, or interacts with the TF-M Crypto service to decrypt them, if in ciphertext. The final output is sent in plain text to the Public QNN Client Application (CA).

Normal-World SW. In the normal-world, most software components are optional, except for the Input Sensors and Public QNN CAs. The former collects and processes the raw data from sensors, so it can be used as input for the QNN. The latter initiates and manages the inference process until a private layer is found. Whenever a private layer is found, the inference process is transferred to the Private QNN TA.

5.5 Evaluation

To evaluate our framework under conditions that resemble a realistic scenario, we relied on the MLPerf Tiny benchmark [24]. This benchmark features QNNs and datasets that reflect the use cases described by more than 30 entities from industry and academia for TinyML systems. Such use cases include scenarios demanding real-time decisions and others where this requirement can be relaxed. Our evaluation focused exclusively on image applications, as images offer a practical means to assess the quality of the surrogate dataset generated by an attacker. Regarding image processing applications, MLPerf Tiny comprises two QNNs: (i) ResNet trained on CIFAR-10 and (ii) MobileNetV1 trained on VWW. CIFAR-10 contains 32x32 images distributed over 10 classes, each representing a different object. VWW is a binary dataset of 96x96 images that represent the use case of detecting whether a person is present in an image. The weights of each QNN are quantized to int-8 precision, while the bias is to int-32.

5.5.1 Surrogate Dataset

Table 10 reports the FID scores of the GANs we trained for the CIFAR-10 and VWW datasets. The FID metric quantifies the disparity between two sets of images - real and synthetic - by comparing the distributions of features extracted from an intermediate layer of a pre-trained network (evaluation backbone). Both real and synthetic images are modeled as Gaussian distributions, characterized by their respective means and covariance matrices. The FID score is computed based on the difference between these distributions. FID scores range from 0 to $+\infty$, where a lower score corresponds to a higher image similarity - more realistic synthetic images.

The GAN configurations selected to be part of our framework are highlighted in green in Table 10. For these configurations, we also measured the accuracy of the target QNN when tested on synthetic images and compared it to the baseline test accuracy, i.e., the accuracy of the original QNN measured on the original test dataset.

Dataset	GAN	Input Dim	FID	Accuracy	
			$[0, +\infty]$	*Surrogate*	*Original*
CIFAR-10	BigGAN+LeCam	32x32	2.26	-	86.53%
	BigGAN+DiffAug	32x32	1.59	88.38%	
	StyleGAN2+DiffAug	32x32	0.91	85.00%	
VWW	ReACGAN-256	128x128	11.17	-	85.75%
	BigGAN-256	128x128	13.68	-	
	StyleGAN2	128x128	4.17	81.77%	
	StyleGAN3-t	128x128	7.05	83.21%	

Table 10: Evaluation of GANs for surrogate dataset generation. GANs integrating SecureQNN are highlighted in green. FID measures the similarity between surrogate and real datasets.

5.5.1.1 CIFAR-10

To maximize the diversity of the surrogate dataset, we evaluated variations of two distinct GAN architectures: (i) StyleGAN and (ii) BigGAN (Section 5.4.1). Regarding the StyleGAN architecture, StyleGAN2 with Differential Augmentation (StyleGAN2 + DiffAug) benchmarks the best/lowest FID score within StudioGAN and we were able to get good quality images from it (FID=0.91). Regarding the BigGAN architecture, BigGAN benchmarks the best FID score in StudioGAN when coupled with LeCam regularization (BigGAN + LeCam). Unfortunately, for our use case, LeCam regularization was insufficient to avoid overfitting. As shown in Figure 13, the lowest FID registered in 105k steps is 2.26 (vs. 0.45 on benchmark) and remained the lowest value after 50k steps. Nevertheless, replacing LeCam regularization with Differential Augmentation (BigGAN + DiffAug) was sufficient to get good quality images (FID=1.59).

Figure 14 puts images crafted by the selected GANs in perspective against the original training data. Although we were not able to reach FID scores as low as the ones benchmarked in StudioGAN - we train GANs on the original test dataset to simulate a realistic attacking scenario - we could still get high quality images. For synthesis purposes, we only show images of five out of the ten classes composing the CIFAR-10 dataset. As observed, figures of airplanes, cars, birds, cats, and deer are completely discernible in the synthetic images. Furthermore, we also observe that the selected GANs were able to generate these figures

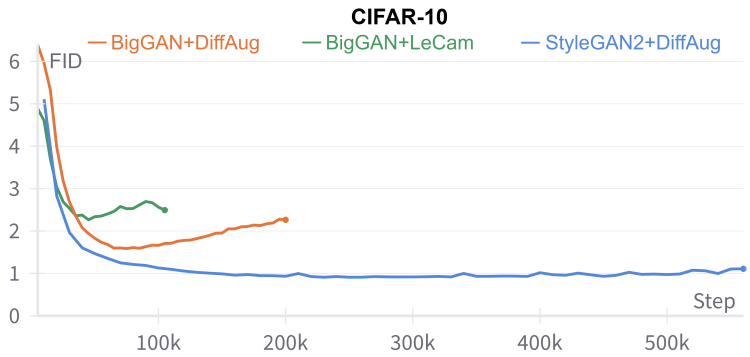


Figure 13: FID evolution over training steps for CIFAR-10 GANs.

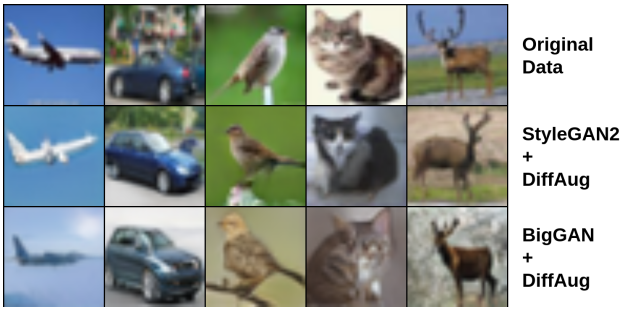


Figure 14: CIFAR-10 samples (line 1) vs. synthetic GAN samples (lines 2-3).

with different backgrounds, increasing the diversity of the dataset. The high quality of synthetic/surrogate datasets is reflected in the accuracy of the target QNN when tested on these synthetic data. The maximum accuracy loss is registered for the StyleGAN2 architecture; however, the loss remains low at 1.53%.

5.5.1.2 Visual Wake Words (VWW)

For the VWW dataset, we also evaluated variations of different GAN architectures (Section 5.4.1). Regarding BigGAN, the ReACGAN-256 and the BigGAN-256 configurations benchmark the best/lowest FID scores on ImageNet (the dataset with the closest image resolution to VWW). However, when applied to the VWW dataset, both configurations result in training collapse, with the FID scores stagnating at 11.17 and 13.68, respectively. As illustrated in Figure 15, these FID scores are observed in the initial training steps - step 30k for ReACGAN-256 and step 8k for BigGAN-256 - after which the FID increases rapidly without ever converging to lower values.

In contrast, the StyleGAN architecture, namely the StyleGAN2 and StyleGAN3-t configurations (the only ones benchmarked in StudioGAN) yielded good results on the VWW dataset. As shown in Figure 15, these configurations do not make the training collapse. The StyleGAN2 and StyleGAN3-t configurations return an FID score of 4.17 and 7.05, respectively, and we could get good quality images from them as shown in Figure 16. For synthesis purposes, we only show images representing the scenario/class where a human is present in the image. Although there is room for improvement, human-like figures are

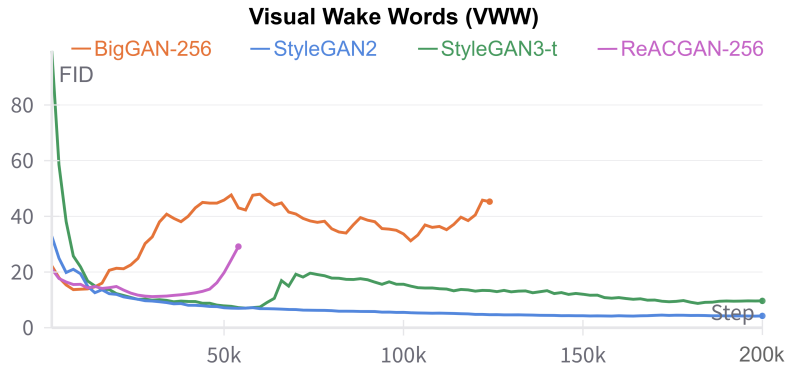


Figure 15: FID evolution over training steps for VWW GANs.



Figure 16: VWW samples (line 1) vs. synthetic GAN samples (lines 2-3).

discernible in the synthetic images, which is reflected in the accuracy of the target QNN when tested on this synthetic data. As detailed in Table 10, the accuracy is comparable to the obtained with the original test data, with a maximum accuracy loss of 4.11% registered for the StyleGAN2 configuration.

As illustrated in Figure 15, training was halted at 200k steps after no further improvement in FID score was observed for over 50k consecutive steps. Due to hardware constraints, extending the training of VWW GANs for marginal FID improvements, as was done for the CIFAR-10 dataset (Figure 13), was not feasible. Unlike CIFAR-10 GANs, which process 32×32 images, VWW GANs operate on 128×128 images, significantly increasing resource demands during training.

5.5.2 Privacy vs. Memory Trade-Off

Table 11 reports the minimum memory required to store QNN parameters within the Private QNN TA (Figure 11) as a function of the maximum accuracy that an adversary might achieve when training a surrogate QNN. The table reflects the worst-case scenario out of four trials, specifically reporting the trial in which the highest number of layers needed to be transferred to the secure-world.

As detailed in Table 11, it is feasible to preserve the intellectual property of a QNN, to the point of undermining the cost-benefit of unauthorized replication, without transferring the execution of the whole

QNN	**Max. Accuracy Surrogate QNN**	**# Private Layers**	**TEE Memory Footprint** *% QNN size in TEE*
ResNet CIFAR-10	86.53%	3 / 10	39.16 KB / 76.90 KB (50.92%)
	76.53%	5 / 10	58.04 KB / 76.90 KB (75.47%)
	66.53%	7 / 10	71.79 KB / 76.90 KB (93.36%)
MobileNetV1 VWW	86.12%	9 / 28	137.38 KB / 213.93 KB (64.22%)
	76.12%	23 / 28	212.57 KB / 213.93 KB (99.36%)

Table 11: Minimum TEE memory footprint vs. maximum accuracy of surrogate QNN.

Board	Memory Size (KBytes)		CIFAR-10		VWW		CIFAR-10 (Crypto)		VWW (Crypto)	
	SRAM	*Flash*	*SRAM* %	*Flash* %	*SRAM* %	*Flash* %	*SRAM* %	*Flash* %	*SRAM* %	*Flash* %
Musca-B1	512	2048+8192	5.36%	0.78%	5.15%	1.91%	26.04%	1.76%	31.44%	2.89%
Musca-S1	512	2048+32768	5.36%	0.23%	5.15%	0.56%	26.03%	0.52%	31.44%	0.85%
nRF5340 DK	512	1024	5.32%	8.15%	5.10%	20.71%	26.17%	18.20%	31.57%	29.20%
nRF9160 DK	256	1024	10.64%	9.70%	10.21%	19.14%	52.34%	19.75%	63.14%	29.19%
LPCXpresso55S69	320	640	8.62%	11.53%	8.28%	29.58%	42.93%	26.94%	51.57%	45.00%
B-U585I-IOT02A	786+8192	2048+65536	0.31%	0.13%	0.30%	0.31%	1.50%	0.28%	1.81%	0.45%
STM32L562E-DK	256	512+65536	10.67%	0.12%	10.24%	0.30%	52.39%	0.27%	63.20%	0.44%
NUCLEO-L552ZE-Q	256	512	10.67%	15.90%	10.24%	38.46%	52.39%	34.92%	63.20%	57.50%

Table 12: TEE memory footprint of a minimalist application built upon TF-M v1.7. The application within the secure-world only stores and executes the set of private layers complying with the default privacy setting. The application was compiled to reduce the memory size of the firmware (*-Os*).

model to the secure-world. The privacy of the CIFAR-10 ResNet can be maintained by transferring only the last 3 out of 10 layers to the secure-world, which corresponds to 50.92% of the total model size. In the case of MobileNetV1 trained on the VWW dataset, this requirement increases to 9 out of 28 layers, representing 65.27% of the total model size.

After determining the minimum set of layers to comply with the default privacy setting - the minimum set of layers that prevent an attacker from reconstructing a surrogate QNN with the same accuracy as the original model without expending more effort than the SP - SecureQNN starts to decrease the target accuracy achievable by an attacker in buckets of 10%. This enables the SP to trade off increased privacy and TEE memory footprint. As detailed in Table 11, for the CIFAR-10 ResNet, it is possible to decrease the target accuracy by, at least, 20% before being necessary to save and execute the whole QNN on the secure-world. For the VWW MobileNetV1, this value reduces to 10%.

The memory values reported in Table 11 account only for the internal parameters of the QNN (weights and bias). In addition, the MCU where the QNN is deployed must feature enough flash and SRAM to execute the CMSIS-NN library and the TF-M firmware, along with any necessary services. To better understand the memory footprint of the TEE in a real-world scenario, we developed an application based on TF-M v1.7 to perform inference using the CIFAR-10 and VWW QNNs. Results were obtained under the default privacy settings for QNNs and for configurations where the TF-M Crypto service was both enabled and disabled. Table 12 contextualizes the TEE memory footprint of this minimalist application relative to the total SRAM and flash memory capacities of several development boards supported by TF-M v1.7. With TF-M Crypto disabled, the use of SecureQNN is feasible even on very low-budget platforms such as the STM Nucleo-L552ZE-Q. On this development board, the highest TEE memory usage is observed in SRAM when running the VWW QNN; however, the usage remains relatively low at 38.46%. When TF-M Crypto is enabled, it may be more appropriate to consider more expensive and resource-rich platforms, such as the STM B-U585I-IOT02A or the Nordic NRF5340-DK, particularly when working with larger QNNs like the VWW MobileNetV1.

5.5.3 Latency

We compared the decision latency per input sample imposed by SecureQNN against the baseline reference scenario where the full QNN is executed within the normal-world. The experiments were carried out considering the default privacy setting of SecureQNN and using TF-M v1.7 on an STM Nucleo-L552ZE-Q, which features an Arm Cortex-M33 at 110 MHz. Results are detailed in Table 13.

SecureQNN imposes two context switches between the normal and secure worlds per sample, which might suggest an increased decision latency compared to the baseline reference. However, the results reported in Table 13 show that the time spent on context switches, including the transfer of the activations of the last public layer to the secure-world, is residual. When TF-M crypto is off, the two context switches account only for 0.04% and 0.06% of the total decision latency (in terms of clock cycles) in CIFAR-10 and VWW QNNs, respectively. If TF-M crypto is on, context switches represent 0.04% of the decision latency in both neural networks.

When we compare the decision latency imposed by SecureQNN without TF-M crypto against the baseline reference, we observe a slight reduction of clock cycles per inference despite the additional clock cycles spent on the two context switches. We investigated this surprising result, and we concluded that this is related to compiler optimizations gained by splitting the execution of both QNNs into two parts. We can not guarantee that this clock cycle reduction will extend to other QNNs as compiler optimizations are pretty dependent on the target application. As memory is a critical resource in our testing platform, our tests were compiled using the *Os* optimization level. Although it reduces the firmware size, this optimization level might result in unaligned memory accesses which inherently increases the latency.

QNN	**Setting**	**# Private Layers**	**Inference Latency/Sample**	
			CPU Cycles *Seconds*	*Context Switch* *NW->SW->NW*
ResNet CIFAR-10	*Normal World*	3 / 10	111.45M 1.01s	N.A.
	NW+SW Crypto OFF	3 / 10	111.44M 1.01s	44.86K 0.41ms
	NW+SW Crypto ON	3 / 10	120.58M 1.10s	
MobileNetV1 VWW	*Normal World*	9 / 28	88.16M 0.80s	N.A.
	NW+SW Crypto OFF	9 / 28	88.04M 0.80s	47.91K 0.44ms
	NW+SW Crypto ON	9 / 28	120.80M 1.10s	

Table 13: Decision latency per sample for different privacy settings. “NW” stands for normal-world, “SW” for secure-world, and “N.A.” for not applicable.

5.6 Discussion

Although shifting ML computation to the deep edge is an ongoing trend, preserving the intellectual property of ML models remains an open research topic. In this paper, we proposed the first public and open-source framework to increase intellectual property protection of QNNs in devices powered by Arm Cortex-M MCUs and proved its feasibility. However, there is still room for improvement.

Trusted Firmware-M (TF-M) Limitations. In TF-M, memory (SRAM and flash) for the secure and normal worlds is allocated during the boot process and remains statically partitioned thereafter. This approach presents no significant drawbacks in applications where the QNN is consistently or almost always active. However, in scenarios where QNNs are not required to run continuously, static memory allocation imposes unnecessary limits on the amount of software that can be deployed in the normal-world. Enhancing TF-M with dynamic memory allocation for the secure-world post-boot would be beneficial for such a use case. As detailed in Section 5.3.1, the SAU unit of Arm TrustZone-M allows dynamic memory partitioning and is programmable.

Enhanced Defense Against MIAs. By protecting the privacy of critical layers, SecureQNN serves as a defense against white-box MIAs, designed to determine whether a specific sample is part of the training dataset. Although white-box MIAs are significantly more accurate than black-box MIAs, the latter still pose a threat that our framework does not mitigate. Integrating SecureQNN with established defenses against black-box MIAs could offer enhanced security guarantees for the SP. Among the defense mechanisms available for mitigating MIAs [142], defenses applied in the pre-training and training phases are more adequate than defenses in the inference phase. Pre-training and training defenses imply the retraining of the QNN; however, these defenses do not introduce overhead during inference - a key advantage given the resource-constrained nature of Arm Cortex-M MCUs.

Extension to Other ML Architectures. We evaluated our framework on ML models borrowed from the state-of-the-art benchmark for TinyML. The QNNs and datasets included in MLPerf Tiny are representative of applications identified by over 30 entities from both industry and academia. Notably, in addition to two image classification models, the benchmark also includes two models for audio classification, namely for keyword spotting and anomaly detection. This suggests that extending SecureQNN to support audio-based ML represents a meaningful direction for future research. The audio models in MLPerf Tiny - a Depthwise Separable Convolutional Neural Network trained on Google Speech Commands and a Fully-Connected Autoencoder trained on ToyADMOS - are based on convolutional and fully-connected layers. Therefore, supporting such models within SecureQNN would primarily involve enhancing the Surrogate Dataset Generator to enable the synthesis of short audio clips. Generative models based on WaveGAN [143], which has demonstrated the capability to produce word-like audio and is adaptable to other domains, could be a good starting point. However, it is important to note that the QNNs in MLPerf Tiny do

not operate on raw audio but on audio-derived features such as log-mel spectrograms or mel-frequency cepstral coefficients. Therefore, any audio data generation pipeline would need to include a suitable feature extraction component to bridge this gap.

Additionally, within the image processing domain, it would be worthwhile to explore other model architectures already implemented on TinyML hardware, such as CapsNets [71]. Supporting CapsNets would require extending the Model Partitioning module to train capsule layers and updating the Trusted Execution module with the API proposed by Costa et al. [71].

5.7 Conclusion

In this paper, we present SecureQNN, the first public framework to increase the intellectual property protection of ML models in TinyML systems powered by Arm Cortex-M MCUs featuring TrustZone-M technology. Results extracted for QNNs trained on CIFAR-10 and VWW datasets suggest that it is possible to undermine unauthorized QNN replication without delegating the execution of the whole model to the secure-world. Specifically, the default privacy setting was achieved by delegating the execution of only 3 out of 10 layers and 9 out of 28 layers to the secure-world, respectively. This corresponds to 51% to 65% of the total QNN size. Regarding latency, our framework only requires two context switches between normal and secure worlds per input sample, which corresponds only to 0.06% of the decision latency in the worst-case scenario. We are opening the artifacts in the hope of enabling independent validation of the results and encouraging further research on the privacy of QNNs in TinyML systems.

6

David and Goliath: An Empirical Evaluation of Attacks and Defenses for QNNs at the Deep Edge

.

This chapter was published as:
Miguel Costa and Sandro Pinto, "David and Goliath: An Empirical Evaluation of Attacks and Defenses for QNNs at the Deep Edge". In *IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. 2024. doi: 10.1109/EuroSP60621.2024.00035.

Abstract

ML is shifting from the cloud to the edge. Edge computing reduces the surface exposing private data and enables reliable throughput guarantees in real-time applications. Of the panoply of devices deployed at the edge, resource-constrained MCUs, e.g., Arm Cortex-M, are more prevalent, orders of magnitude cheaper, and less power-hungry than APUs or GPUs. Thus, enabling intelligence at the so-called deep/extreme edge is the zeitgeist, with researchers focusing on unveiling novel approaches to deploy ANNs on these constrained devices. Quantization is a well-established technique that has proved effective, i.e., negligible impact on accuracy, in enabling the deployment of neural networks on MCUs; however, it is still an open question to understand the robustness of QNNs in the face of well-known adversarial examples.

To fill this gap, we empirically evaluate the effectiveness of attacks and defenses from (full-precision) ANNs on (constrained) QNNs. Our evaluation suite includes three QNNs targeting TinyML applications, ten attacks, and six defenses. With this study, we draw a set of interesting findings. First, quantization increases the point distance to the decision boundary and leads the gradient estimated by some attacks to explode or vanish. Second, quantization can act as a noise attenuator or amplifier, depending on the noise magnitude, and causes gradient misalignment. Regarding adversarial defenses, we conclude that input pre-processing defenses show impressive results on small perturbations; however, they fall short as the perturbation increases. At the same time, train-based defenses increase adversarial robustness by increasing the average point distance to the decision boundary, which holds even after quantization. However, we argue that train-based defenses still need to smooth the quantization-shift and gradient misalignment phenomenons to counteract adversarial example transferability to QNNs. All artifacts are open-sourced to enable independent validation of results and encourage further exploration of the robustness of QNNs.

6.1 Introduction

The availability of large volumes of data and tremendous computing power turned ML into a critical technology to drive innovation and assist humans in tasks ranging from healthcare [1] and finance [2] to mobility [4]. As ML workloads tend to be computationally intensive, most ML systems developed to date followed a cloud-centered design, sending the data collected at the edge to the cloud [5, 6, 73]. The cloud performs the ML computation and returns the final decision to the edge responsible for actuating over the environment. However, recent concerns about privacy, latency, power consumption, and cost are forcing ML to move from the cloud to the edge near the data source [8, 14].

Shifting ML workloads to the edge reduces the exposure of private data, as there is no data transmission over the network [8, 14]. Besides reducing the chances of data stealing, this also increases the users' confidence in applications and services as it lessens the need to provide personal data to a third party. Furthermore, edge computing enables ML systems to be more predictable as the latency is not

dependent on the available network bandwidth [6, 8, 14]. Therefore, edge computing enables developers to provide reliable throughput guarantees, mandatory in real-time or mission-critical scenarios [110].

Edge devices are now the new frontier for ML deployment [6, 8, 14, 71]. Of the panoply of devices typically deployed at the edge of the network, MCUs are getting special attention, as newer architectures aligned with friendly developer tools (e.g., CMSIS-NN [18], PULP-NN [19], and TensorFlow Lite Micro [20]) are enabling the deployment of ML systems in devices that are several orders of magnitude cheaper and less power-hungry than APUs or GPUs. Despite the resource-constrained nature of these devices, ANN compression techniques such as quantization have proved to have negligible impact on accuracy while tremendously improving latency [18, 19].

In parallel with this development, recent research has shown that full-precision ANNs are vulnerable to the so-called adversarial examples [46–48], i.e., inputs carefully crafted to fool the neural network into making an obvious misclassification. In this context, it is essential to ask if QNNs are susceptible to this class of attacks and, if so, how QNNs could be protected. Previous research found that QNNs are still vulnerable to gradient-free attacks and adversarial example transferability, even to those crafted on full-precision ANNs using weak attacks such as FGSM [8, 14, 49–51]. Lin et al. [50] pioneered in adversarial defenses for QNNs by proposing a novel training algorithm that reduces the quantization-shift phenomenon; however, the authors do not provide a clear picture of the scope of their work. As GPUs operate better with integer precision, QNNs are deployed not only on ultra-low-power MCUs but also on powerful GPUs. As GPUs have far more memory and computing power than ultra-low-power MCUs, QNNs deployed on these devices show significant differences in the quantization policy, kernel filter size, and network depth. In this context, providing a clear picture of the quantization policy and the target computing devices when working with QNNs is fundamental. To the best of our knowledge, no work proposes possible defense mechanisms or evaluates the feasibility of adapting the existing defenses for full-precision ANNs on QNNs tailored for the TinyML scenario.

To fill this gap, we performed a comprehensive empirical evaluation to understand how attacks and defenses targeting full-precision ANNs perform in the presence of QNNs. Despite the few studies on the adversarial robustness of QNNs, the information tends to be scattered and needs a well-defined basis for ground comparison. Furthermore, if adversarial examples also affect QNNs, as previously stated, it is paramount to paint a clear picture of QNNs defenses. This paper tests the robustness of three QNNs targeting TinyML applications on ten attacks and six possible defenses. This empirical evaluation aims to answer the following research questions:

- *Q1 - Can quantization alone protect neural networks from adversarial attacks?*
- *Q2 - How does the bit-width of quantization affect a QNN's defense against adversarial examples?*
- *Q3 - Do defenses for full-precision ANNs still work for QNNs?*
- *Q4 - Can existing defenses for full-precision ANNs be used on resource-constrained MCUs (deep-/extreme edge)?*

To answer *Q1* (Section 6.3), we evaluated the robustness of QNNs without any defense in place. We measured the adversarial accuracy and the distortion when directly attacking QNNs and how adversarial examples crafted on full-precision ANNs transfer to QNNs. We observe that quantization increases the average point distance to the decision boundary and makes it more difficult for attacks to optimize over the loss surface, leading to the explosion or vanishing of the estimated gradient. Additionally, we observe poor transferability of adversarial examples crafted on full-precision ANNs to QNNs, resulting from gradient misalignment and quantization-shift. Overall, quantization minimizes the effects of small perturbations but can amplify bigger perturbations.

To answer *Q2* (Section 6.3), we replicated all selected attacks on 8-bit and 16-bit QNNs. We evaluated how the bit-width affects the adversarial example transferability and the success of direct attacks. We observe that the average point distance to the decision boundary increases as the neural network precision decreases. This explains why int-8 models are generally more robust than int-16 and float-32 models, with int-16 being more robust than float-32. Furthermore, int-16 models are more affected than int-8 models by the adversarial examples transferred from float-32 ANNs. This occurs as int-8 QNNs map more float-32 values in the same quantization bucket than int-16 QNNs, which enhances the effect of quantization-shift and gradient misalignment.

To answer *Q3* and *Q4* (Section 6.4), we tested six popular defense mechanisms against the attacks that impact the most the adversarial accuracy of QNNs before any defense. We observe that input pre-processing defenses show impressive results in denoising small perturbations but fall short when the perturbation increases. Train-based defenses generally increase the average point distance to the decision boundary, and this behavior holds even after quantization. However, not all defenses can be fully applied in ultra-low-power ML applications. Train-based defenses are easily portable, as they do not require additional computing power at inference. However, they are less robust than pre-processing defenses for smaller, fine-grained perturbations.

In short, this work makes the following contributions:

1. *Presents the most comprehensive empirical evaluation of adversarial robustness of QNNs targeting TinyML applications (Section 6.3);*

2. *Evaluates the impact of bit-width on adversarial example robustness, testing 8-bit and 16-bit quantization setting of TensorFlow Lite Micro (Section 6.3);*

3. *Delves into the uncharted territory of defenses against adversarial examples in the context of QNNs and empirically assesses if defenses designed for full-precision networks still work when applied to QNNs (Section 6.4).*

6.2 Motivation and Methodology

6.2.1 TinyML: The New Wave in AI

TinyML is a field of ML dedicated to the design and development of hardware and software solutions to enable the execution of ML inference under a milliWatt scale [24]. By enabling on-device and near-sensor data analysis, TinyML eliminates the imperative reliance on the cloud infrastructure for inference results [8, 14]. This paradigm shift not only enables the dissemination of ML in regions without reliable network, but also ensures greater responsiveness and bolstered privacy.

Under this umbrella, ML at the extreme edge (powered by MCUs) is one of the hottest topics in the ML field due to the proliferation of these devices in our infrastructures/society. In effect, ARM predicts that MCUs are expected to reach 1 trillion across various market segments by 2035 [18]. A recent report [30] from ABI Research dating to 2022 emphasized that TinyML Software-as-a-Service was expected to exceed US$220 million in revenue in 2022 and have the potential to become a billion-dollar market by 2030. This same company also forecasts that the TinyML market will grow from 15.2 million shipments in 2020 to 2.5 billion in 2030 [31].

In academia, this trend has fueled the development of multiple software and hardware solutions to enable and optimize the execution of TinyML and has fuelled the rise of RISC-V ISA. Among other relevant works, Flamand et al. [21] proposed a custom SoC based on the emerging RISC-V ISA [21], while Garofalo et al. [19] proposed a software library that optimizes neural network functions on this same SoC. Scherer et al. [23] proposed a RISC-V hardware accelerator able to deliver 3200 inferences/second while only consuming 12.2 milliWatts.

In terms of use cases, TinyML has been widely deployed for (i) word spotting, (ii) object recognition, (iii) object counting, and (iv) audio/voice detection [24, 30]. More recently, its application has been extended to the detection of (i) forest fire, (ii) shape, and (iii) seizure [30].

6.2.2 Scope

This work targets ultra-low-power ML applications (TinyML), commonly powered by Arm Cortex-M (Armv7-M or Armv8-M) and RISC-V MCUs (e.g., PULP [159]). These devices typically have a single core and feature flash and SRAM. They execute instructions in place from physical memory and lack an MMU for virtual memory support. Consequently, they are unable to run Unix-like OSs.

TensorFlow Lite Micro, which operates on top of the ARM-developed CMSIS-NN library, represents the state-of-the-art solution for executing QNNs on these MCUs. TensorFlow Lite Micro is accelerated by ARM's SIMD; however, it requires a quantization policy that has been mostly neglected in previous works. We highlight that quantization can be used (i) to deploy QNNs in MCUs, (ii) to speed up inference on GPUs or TPUs, or (iii) to simply reduce the size of neural networks in mobile devices (e.g., smartphones). However, the quantization policy for GPUs/TPUs and APUs is significantly different from MCUs - thus, QNNs have completely different architectures and parameters, leading to different findings.

To help standardize the throughput of ultra-low-power hardware on ML applications, Banbury et al. [24] developed the MLPerf Tiny benchmark, composed of four QNNs targeting image and sound applications. Our empirical evaluation targets QNNs developed under the exact specification as MLPerf Tiny: QNNs using fixed-point quantization with 8 or 16 bits for weights and activations and quantized according to the TensorFlow Lite Micro policy. Within this policy, the number of bits representing the integer and fractional parts is not fixed, varying for weights and activations within the same layer.

6.2.3 Adversary Model

We define the adversary model of our research along three dimensions, which concern the (i) goal, (ii) capability, and (iii) knowledge of the attacker.

Adversary Goal. The adversary's goal is to compromise the decision integrity of a supervised model, disrupting severe accuracy drops. Previous research shows that the attacker can achieve this goal by interfering with the training or inference phase. However, research on training at the deep edge is still at an embryonary stage [73], and this class of devices is still only used for inference purposes. The most relevant part of the training of the models deployed on deep-edge devices is still performed in cloud-like environments. Therefore, we limit the scope of our research to the inference phase. Attacks to craft adversarial examples can either be targeted or non-targeted. Targeted attacks misguide an ANN by making it classify an adversarial sample as a specific class. In contrast, in non-targeted attacks, the class returned by the model is arbitrary except for the original one. Our research encompasses only non-targeted attacks, as they enable a broader assessment of the robustness and generalization of a neural network.

Adversary Capability. Capability refers to the attacker's capacity to control or manipulate the input data. At inference, the generic data processing pipeline of an ML system considers that the (i) input features are collected from sensors or data repositories and (ii) processed in the digital domain, (iii) used by the model to infer about the surrounding environment, (iv) which communicates the result to the appropriate actuator [110]. Considering this pipeline, adversarial examples can be generated either in the physical domain, by tampering with sensors, or in the digital domain. This work only considers attacks and defenses in the digital domain, which dominates the available literature.

Adversary Knowledge. Datasets are, at least, divided into two parts: one for training and another for testing. The training subset is exclusively used for training and the testing subset is used to evaluate the real accuracy of the model. In our threat model, we assume that the attacker has full access to the testing data and can use this data as input to a given attack to craft an adversarial dataset, on which the accuracy of the attacked model is evaluated. Regarding the knowledge of the target model, we define three testing scenarios. For every testing scenario, the adversary is aware of the quantization status of the network, i.e., the adversary knows that the neural network under attack is quantized.

- *White-box:* The adversary has full access to the attacked model, including its architecture and parameters. When considering QNNs, the adversary also has access to the scaling and zero-point factors used to quantize the input, output, and internal parameters.
- *Gray-box:* The adversary does not know the architecture of the target model or its internal parameters. Nevertheless, they have access to class probabilities. The scaling and zero-point factors used in quantization remain secret.
- *Black-box:* The adversary can only feed input data to the target model and query its output. They cannot access the class probability vector but the final decision.

6.2.4 Selected Datasets and Models

To benchmark the throughput of the hardware used in TinyML, Banbury et al. [24] proposed the MLPerf Tiny benchmark. This benchmark results from the collaborative effort of more than 30 organizations from industry and academia and reflects the community's needs by simulating real-world applications (see Section 6.2.1). As the datasets and models used in these applications are the companies' intellectual property, researchers selected representative and open-sourced models and datasets.

We borrowed the models and datasets from the MLPerf Tiny benchmark but limited the empirical evaluation to image processing tasks, as images provide a convenient means to observe the impact of adversarial attacks and defenses. Table 14 summarizes the benchmarking suite.

In terms of image processing, the MLPerf Tiny benchmark comprises two datasets: (i) CIFAR-10 and (ii) VWW. CIFAR-10 consists of 60000 32x32 Red-Green-Blue (RGB) images distributed over ten classes, each representing a different object. There are 50000 images for training and 10000 images for testing. VWW is a binary dataset with 115000 96x96 RGB images representing a vision use-case of identifying whether a person is present in the image or not. The training subset contains 98568 images, and the testing subset contains 10961.

We included a third dataset because evaluating attacks and defenses on only two datasets may fall short. For this purpose, we borrowed the work [144], which lists and reviews ML implementations for Arm Cortex-M MCUs. After analyzing the reproducibility of each listed work, we replicated the work [140]. This work addressed the development of an ML system devoted to detecting coffee plant diseases. The dataset is public and contains 1747 images for training and testing distributed over four classes, each corresponding to a different disease. All QNNs included in our study are quantized post-training.

Use Case	**Dataset**	**Model**
Image classification	CIFAR-10	ResNet
Human face detection	Visual Wake Words (VWW)	MobileNetV1
Coffee plant monitoring	Coffee Dataset	DS-CNN

Table 14: Benchmarking suite for QNNs adversarial robustness.

6.2.5 Selected Attacks and Defenses

We prioritize attacks/defenses with great impact and citation in the research community and whose implementation is open-sourced. Furthermore, since we target QNNs following the TensorFlow Lite Micro quantization policy, we prioritize attacks and defenses implemented in TensorFlow 2 or Keras (the back-end engine of TensorFlow 2). TensorFlow 2 provides straightforward mechanisms to operate with models in TensorFlow Lite format. We also prioritize attacks/defenses available in ART [160], Cleverhans [161], or Foolbox [162, 163] frameworks, as these frameworks provide a straightforward mechanism to benchmark different attacks and defenses upon a common interface. Our evaluation suite includes six white-box, two gray-box, and two black-box attacks, described in Table 15.

White-box Attacks. White-box attacks are used to evaluate the robustness of QNNs against adversarial transferability (Section 6.3.2). Transferability is a well-known vulnerability of ANNs and refers to the ability of an adversarial example to simultaneously evade the prediction on two or more ML models performing similar tasks (trained on similar or equal datasets). In our threat model, the adversary uses the white-box attacks listed in Table 15 to craft adversarial examples on a full-precision ANN, which are then quantized and applied against the corresponding int-16 and int-8 QNNs. The full-precision ANN can be seen as a surrogate model mimicking the QNN on which the adversary performs transfer-based attacks.

We do not use white-box attacks to directly attack QNNs as white-box attacks require an exact gradient calculation, which is impossible due to the quantization policy. More specifically, this results from the dynamic bit-width used to represent the fractional and integer parts throughout the network. Despite every activation and model parameter using a static bit-width of 8 or 16 bits, the number of bits dedicated to the fractional and integer parts is dynamic. Furthermore, the fractional bit-width in the same layer differs for the input, output, and model parameters. In this scenario, the only feasible way to calculate a gradient is to convert these parameters to a common format, preferably with higher precision (float-32), and calculate the gradient afterward. This process corresponds to dequantizing the QNN back to full-precision and applying the attacks afterward. The resulting adversarial examples are then quantized (converted from float-32 to int-16 or int-8) and transferred to the respective QNN. However, this process is, in reality, evaluating the transferability of adversarial examples.

Gray- and Black-Box Attacks. The gray- and black-box attacks listed in Table 15 are used to evaluate the adversarial transferability, but also to directly attack each ANN and QNN, i.e., craft adversarial examples tailored for a given neural network, and compare how the bit-width of a neural network affects its adversarial robustness. We do not include white-box attacks targeting QNNs as existing attacks only apply to QNNs whose gradients can be approximated by STE or Mirror Descent techniques. This feature is absent in QNNs compliant with the TensorFlow Lite Micro quantization policy. Nevertheless, we include a series of gray- and black-box attacks (Square Attack, Boundary Attack, and GeoDA) developed to tackle gradient obfuscation, a phenomenon disrupted by QNNs that gives a false sense of security.

Attack	Year	Norm	Description	Relevant Formulas
			WHITE-BOX	
DeepFool [114]	2016	L_2	An iterative attack that seeks to find the minimum perturbation δ to misclassify x, moving it toward the closest boundary. At iteration t, DeepFool uses Taylor expansion to linearize the boundary around x_t, moving x_t outward. This process is repeated until x_t is misclassified.	$min\|\|\delta_t\|\|$ such that $L(x'_t) + \nabla L(x'_t)^T \delta_t = 0$
JSMA [115]	2016	L_0	The Jacobian-based Saliency Map Approach (JSMA) iteratively perturbs the most relevant features of x. It first calculates the Jacobian matrix of x and then the adversarial saliency map S. JSMA distorts features in order of relevance: each feature is distorted by δ until x is misclassified or the maximum number of features is distorted.	$S = \begin{cases} 0, \text{ if } \frac{\partial L_{y'}(x)}{\partial x_i} < 0 \text{ or } \sum_{y \neq y'} \frac{\partial L_y(x)}{\partial x_i} > 0 \\ \frac{\partial L_{y'}(x)}{\partial x_i} * \left\lvert \sum_{y \neq y'} \frac{\partial L_y(x)}{\partial x_i} \right\rvert, \text{ otherwise} \end{cases}$
C&W [116]	2017	L_0 L_2 L_∞	For any L_p norm, the Carlini and Wagner (C&W) attack follows the same optimization problem to find the minimum distortion (min_δ). c is a constant weighting the relative importance of distance and loss, and $Z(x')_y$ denotes the probability of the target class y. k sets the minimum confidence for the adversarial example.	$min_\delta\, D(x, x+\delta) + c.L(x+\delta)$ $L(x') = max(max\{Z(x')_y : y \neq y'\} - Z(x')_y - k)$ $\delta_i = \frac{1}{2}(tanh(\sigma_i) + 1) - x_i$
PGD [57]	2018	L_∞	Projected Gradient Descent (PGD) is an iterative attack that applies a small perturbation α to each feature of x in the same direction as the gradient till a maximum distortion δ. At the beginning of each iteration, PGD adds random noise to x and restarts the iteration from the most adversarial point.	$x_t = Clip_{x_t,\delta}(x_{t-1} + \alpha * Sign \frac{\partial L(w, x_{t-1}, y)}{\partial x_{t-1}})$
EAD [112]	2018	L_1 L_2	Elastic-Net Attack to DNNs (EAD) is a variant of the C&W attack that simultaneously minimizes L_1 and L_2 norms. c weights the loss function L against the distortion and β sets the relevance of L_1 norm.	$min_{x'} c * L(x', y') + \beta \left\|x' - x\right\|_1 + \left\|x' - x\right\|_2^2$
AutoAttack [119]	2020	L_1 L_2 L_∞	A parameter-free attack, grouping four attacks: two Auto-PGD (APGD) versions, Fast Adaptive Boundary Attack (FAB), and SA. APGD differs from PGD as the step size is adjusted dynamically based on the optimization trend. If the loss increases in 75% or more of the past iterations, the step size is maintained; otherwise, it is halved. The two versions of APGD differ in their loss function: one uses cross-entropy, and the other uses the difference of logits ratio. FAB is an optimized version of DeepFool.	-
			GRAY-BOX	
ZOO [120]	2017	L_2	Zeroth-Order Optimization (ZOO) is a gray-box version of C&W$_{L2}$ attack. ZOO uses an alternative formula to estimate the gradient, where h is a small constant and e_i is a standard basis vector with only the i^{th} component set to 1. To be efficient, ZOO uses stochastic coordinate descent methods.	$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h}$
SA [121]	2020	L_2 L_∞	Square Attack (SA) is an iterative attack that samples a random square-shaped perturbation per iteration and adds it to x if it increases the classifier's error. The attack stops as soon as it finds an adversarial example. SA decreases the width and distortion of squares according to a fixed schedule.	-
			BLACK-BOX	
BA [122]	2018	L_2	Boundary Attack (BA) samples a first adversarial example x' from a uniform distribution. Then it performs a random walk along the decision boundary such that x' remains adversarial and the distance towards x decreases. BA samples a new direction at each iteration and projects x'_i on the δ neighbor-ball of x. Then x'_i is moved towards x. The x'_i closest to x respecting the adversarial condition is selected.	-
GeoDA [123]	2020	L_2	Geometric Decision-Based Attack (GeoDA) uses binary search to find a point belonging to the nearest decision boundary. This point is used to define a hyperplane that approximates the boundary surface. After defining the hyperplane, GeoDA uses an iterative method to find a vector normal to the boundary and uses that vector to perturb x until misclassification.	-

Table 15: Adversarial example attacks.

Defense	Year	Description	Tested Against
TRAIN-BASED DEFENSES			
Defensive Distillation [56]	2016	Distillation works by firstly training the teacher model with the original dataset using the hard-coded labels. Then, the soft labels returned by the teacher model are used to train the student model. Defensive Distillation uses high distillation temperatures in softmax activation, as high temperatures cause the softmax to return smoother probabilities, reducing the sensitivity of the ANN to input variations.	JSMA
PGD Adversarial Training [57]	2018	Adversarial training consists in retraining an ANN by injecting adversarial examples into the training dataset. The first work on adversarial training [113] proposed to generate adversarial examples using FGSM in each training iteration, mixing clean and adversarial samples. This work was later borrowed by [57], which proposed adversarial training with PGD.	FGSM BIM DeepFool JSMA C&W Attack
Ensemble Adversarial Training [58]	2018	Ensemble Adversarial Training improves the training dataset by incorporating adversarial examples from other classifiers. This approach separates the generation of adversarial examples from the training process, enabling it to scale more efficiently to larger datasets than PGD Adversarial Training.	FGSM BIM PGD
Sinkhorn Adversarial Training [59]	2021	Sinkhorn Adversarial Training (SAT) is a recent iteration of PGD Adversarial Training that adopts Sinkhorn Divergence as the loss function. The Sinkhorn Divergence exploits the optimal transport theory to calculate and reduce the ground distance between representations of original images and their adversarial counterparts. Adversarial images are calculated at each training epoch using the PGD attack.	PGD
INPUT PRE-PROCESSING DEFENSES			
Feature Squeezing [60]	2017	Feature Squeezing decreases the color bit-depth and performs spatial smoothing of the input sample. If the classification of squeezed and unsqueezed samples differ by a large margin, i.e., exceeds a given L_1 distance, Feature Squeezing classifies the sample as adversarial.	Transferability
Pixel Defend [61]	2018	Pixel Defend uses a generative network to denoise adversarial examples. The generative network is trained on the same data as the classifier to learn the inherent data distribution. This model is then used to reconstruct adversarial examples, bringing them back to the same distribution of training data. Pixel Defend uses PixelCNN as the generative network.	FGSM BIM DeepFool C&W Attack

Table 16: Adversarial example defenses. Basic Iterative Method (BIM) is an iterative version of FGSM.

Defenses. We consider six defense mechanisms (Table 16) distributed across the two types of defense proposed by the research community. Our empirical evaluation includes four train-based defenses and the other two based on input pre-processing. Although there might be more recent defenses in the literature than those we selected, PGD Adversarial Training, proposed in 2018 and included in our work, remains the state-of-the-art defense against adversarial attacks according to recent studies [119, 164, 165]. Nevertheless, we still include a recent iteration of this defense, entitled SAT (2021), which promises to reduce the ground distance between original images and their adversarial representations.

6.2.6 Empirical Evaluation

To answer the research question *Q1*, we evaluate how QNNs, without any defense in place, perform compared to the respective full-precision ANNs in the face of adversarial examples. The evaluation focuses on adversarial accuracy, i.e., the percentage of adversarial examples correctly classified, and on adversarial

distortion. To measure the distortion of adversarial examples, we rely on the distance metrics L_0, L_1, L_2, and L_∞. We compare the robustness of QNNs and ANNs when facing adversarial examples specifically tailored for the neural network under attack (Section 6.3.1) and how adversarial examples crafted on full-precision ANNs transfer to QNNs (Section 6.3.2). To answer the research question *Q2*, the experiments are performed for QNNs with 8-bits and QNNs with 16-bits. Our work targets the quantization policy of TensorFlow Lite Micro, which restricts the bit-width to these two options.

Table 27 (Appendix A) details the configuration of the attacks. We configured the attacks such that the accuracy of the target ANN drops below the value of random choice (10% for CIFAR-10, 50% for VWW, and 25% for Coffee). Then, we applied the attacks with the same parameters to the respective QNN (direct attacks) or quantized the adversarial examples previously produced and applied them to QNNs (adversarial transferability). Other possible approaches to configure the attacks would be to (i) stop the attack when the distortion reaches an upper bound instead of stopping the attack upon a fixed accuracy degradation or (ii) stop the attack only when the adversarial accuracy drops to 0% in every model/bit-width combo and then compare the distortion. Regarding the former, it must be noted that not all attacks optimize the same distortion (L_p) metric. Considering that there are 4 distortion metrics, this approach will increase the engineering effort of our work as we would have to control 4 variables (L_0, L_1, L_2, and L_∞) instead of 1 (accuracy). Regarding the latter, it would require distinct configurations of the attacks for different bit-widths, which would require more engineering effort.

To answer the research questions *Q3* and *Q4*, we split the evaluation of defenses into three parts. Firstly, we apply each defense in full-precision and quantized models. For train-based defenses, we first retrain the full-precision ANN with the defense constraints and then we quantize the resulting ANN, obtaining the retrained QNN. For input pre-processing defenses, we simply apply the defense before the ANN/QNN input. The configuration of each defense is detailed in Table 28 (Appendix A). For each defense, we experimented with a different set of hyper-parameters around the default values suggested by the defenses' authors and selected the configuration that perform the best against the PGD attack. Secondly, we quantitatively test each defense against the attacks that most impact the adversarial accuracy of QNNs. The evaluation includes six attacks (three white-box and three gray- and black-box). Thirdly, we perform a qualitative analysis of each defense. We evaluate whether they are suitable or not for TinyML and if so, which adjustments must be made. The results are detailed in Section 6.4.

6.2.7 Threats to Validity

The experiments were conducted by two independent researchers on two separate but equal platforms: STM NUCLEO-L552ZE-Q. Whenever possible, we used the implementation of attacks and defenses available on the ART framework [160], except for AutoAttack and SAT. Regarding AutoAttack, the original implementation is significantly different from the implementation of ART. SAT is not available on ART and is originally implemented in PyTorch. In these two cases, we borrowed the original implementations and adapted them to work with TensorFlow 2. Despite all efforts, there may be slight variations compared to

the original implementations. To encourage further reproducibility of our results, all artifacts are public and available open-source[1].

6.2.8 Summary of Observations

First, quantization increases the average point distance to the decision boundary and makes it more difficult for attacks to optimize over the loss surface. This effect can lead to either the explosion or vanishing of the estimated gradient, a phenomenon known as gradient masking. Second, adversarial examples crafted on full-precision ANNs do not transfer well to QNNs due to gradient misalignment and quantization-shift. Lastly, quantization can mitigate small perturbations but also amplify bigger perturbations.

While input pre-processing defenses show impressive denoising results for small perturbations, their effectiveness diminishes as the perturbation increases. On the other hand, train-based defenses generally increase the average point distance to the decision boundary and this property remains even after quantization. However, not all defenses can be applied in TinyML applications. Train-based defenses are the most easily portable but less robust for smaller, fine-grained perturbations. The presence of a DSP or FPU on the target MCU potentiates the implementation of input pre-processing defenses.

6.3 Robustness Against Adversarial Examples

In this section, we empirically evaluate the robustness of QNNs and compare their performance against their full-precision ANNs. Table 27 (Appendix A) details the configuration of the attacks. We compare the effect of direct attacks and the adversarial example transferability from float-32 ANNs to int-16 and int-8 QNNs. For both scenarios, we evaluate the adversarial accuracy and distortion under L_0, L_1, L_2, and L_∞ norms. Table 17 details the accuracy of the models under test before any attack.

	CIFAR-10	**VWW**	**Coffee**
Float-32	88.30%	91.30%	97.00%
Int-16	88.20%	91.00%	97.00%
Int-8	87.50%	91.00%	97.00%

Table 17: Test accuracy of the models before adversarial attack.

6.3.1 Direct Attacks

We compared the robustness of QNNs against full-precision ANNs using the gray-box and black-box attacks described in Table 15. We do not include white-box attacks in this evaluation as these attacks can not be directly applied to QNNs. Results are detailed in Table 18.

[1] `https://gitlab.com/ESRGv3/QNN-Adversarial-Robustness.git`

Attack		CIFAR-10					Visual Wake Words					Coffee Dataset				
		Acc	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞
GRAY-BOX																
ZOO	*Float-32*	24.70%	141.14	4.40	0.31	0.05	54.00%	13502.90	462.85	4.48	0.20	92.00%	16291.00	390.52	3.02	0.08
	Int-16	72.00%	51.97	2.02	0.14	0.02	69.00%	19269.35	693.58	6.60	0.29	80.00%	34388.56	896.46	6.82	0.18
	Int-8	87.50%	0.00	0.00	0.00	0.00	84.00%	4961.24	204.20	1.95	0.09	97.00%	5168.19	151.97	1.18	0.03
Square-L_2	*Float-32*	47.30%	3044.04	14.75	0.87	0.22	45.30%	12721.11	98.78	1.80	0.28	34.10%	74699.49	218.60	1.80	0.22
	Int-16	46.60%	1139.35	14.01	0.83	0.21	42.80%	12244.77	216.54	2.52	0.28	35.40%	64190.22	2715.53	8.57	0.22
	Int-8	67.80%	443.77	11.44	0.75	0.22	64.60%	7176.22	352.30	3.64	0.32	65.20%	16830.86	627.71	3.11	0.23
Square-L_∞	*Float-32*	6.10%	3045.62	33.26	0.65	0.01	37.60%	16671.67	248.55	1.78	0.01	59.40%	90578.27	1356.36	3.94	0.01
	Int-16	16.40%	1845.00	27.57	0.54	0.01	40.80%	16614.91	247.64	1.76	0.01	54.00%	62088.17	929.52	2.73	0.01
	Int-8	82.30%	1125.94	17.60	0.36	0.01	86.00%	11173.88	167.20	1.23	0.01	78.70%	27792.02	435.28	1.35	0.01
BLACK-BOX																
Boundary	*Float-32*	7.70%	3070.36	8.15	0.19	0.01	91.30%	2764.59	3.59	0.02	0.00	2.20%	150489.77	2209.77	7.15	0.09
	Int-16	8.00%	2989.68	9.10	0.21	0.02	91.00%	2428.34	3.55	0.03	0.00	0.00%	150347.67	2684.71	8.69	0.11
	Int-8	6.50%	2450.47	61.34	1.38	0.08	91.00%	1928.49	20.30	0.16	0.00	1.00%	144495.22	7928.35	25.56	0.29
GeoDA	*Float-32*	11.00%	2413.11	36.95	0.77	0.03	91.30%	828.96	414.74	2.80	0.03	0.10%	145536.22	1561.76	5.09	0.06
	Int-16	22.00%	2284.17	83.51	1.72	0.06	91.00%	1368.56	687.31	4.77	0.05	1.00%	142126.62	2374.74	7.67	0.07
	Int-8	13.00%	2193.12	98.67	2.08	0.08	91.00%	826.04	412.77	2.79	0.03	4.50%	132656.10	10057.61	31.51	0.26

Table 18: Quantization impact on direct attacks. Red highlights samples that affect the accuracy and whose noise is detectable to the human eye. Yellow/blue highlights samples that do not affect the accuracy and whose noise is detectable/not detectable to the human eye.

A1. Point distance to the decision boundary. The results presented in Table 18 indicate that int-8 models exhibit greater robustness than the float-32 and int-16 versions, with int-16 being more robust than float-32 models. When considering black-box attacks, we observe similar adversarial accuracies between every model bit-width; however, the same is not verified for the distortion metrics. Although the L_0 norm shows the number of perturbed pixels tends to decrease as precision lowers, the other metrics (L_1, L_2, and L_∞) show the amount of distortion the perturbed pixels get is significantly higher. For reference, the L_2 distortion reported in Table 18 for the Boundary attack on the Coffee int-8 QNN ($L_2 = 25.56$) is 3.57x higher than the distortion generated by the same attack for the Coffee float-32 ANN ($L_2 = 7.15$).

The fact that the distortion significantly increases for similar or higher adversarial accuracy as the precision lowers suggests that the average point distance to the decision boundary increases as precision lowers. This finding is reinforced in the analysis *A5* of Section 6.3.2 and was previously observed in [14]. Gorsline et al. [14] defined a geometric model proving that the average distance to the decision boundary increases as the weight precision and the input dimensionality decrease.

Takeaway 1. The average point distance to the decision boundary increases as the precision of a neural network decreases, making int-8 models generally more robust to adversarial examples than int-16 and float-32 versions.

A2. Explosion/Vanishing of estimated gradients (gradient masking). Quantization makes it more difficult to optimize over the loss surface. On the one hand, when the perturbation added by the attack is small, it may be insufficient to shift the quantization bucket. Consequently, the model will return the same output, making the gradient estimated by the attack null. On the other hand, when the perturbation is large enough, the output variation could be so significant that it may explode the gradient. These scenarios, where the gradient becomes difficult to compute, are referred to as gradient masking.

Although gray- and black-box attacks reduce the chance of gradient masking, we can still observe some scenarios of gradient masking when we compare the results of direct attacks (Table 18) against adversarial transferability (Table 19) for int-8 models. This behavior is noticeable in the Square and ZOO attacks on all datasets. For this set of attacks, attacking the float-32 ANN and transferring the adversarial examples to the int-8 QNNs have more impact on the adversarial accuracy than using the attacks directly on the int-8 QNNs. For instance, when directly attacking the CIFAR-10 int-8 QNN using the Square-L_∞ attack, we register, in Table 18, an adversarial accuracy of 82,30%. However, if we use adversarial example transferability to attack this same QNN, its adversarial accuracy drops to 29,90% (Table 19).

Supported by the collected empirical evidence, we argue that quantization makes the loss surface of models harder to optimize over as quantization increases the possibility of gradient explosion or vanishing. To prove our argument, we calculate the average zero density for the gradients estimated for the float-32, int-16, and int-8 models trained on CIFAR-10. As the quantization policy we are using does not enable the exact calculus of the gradient, we use the finite difference method to estimate it. We observed an average zero density of 70.20%, 72.68%, and 88.18% for the gradients estimated in float-32, int-16, and int-8 models. As the zero density increases as the precision lowers, we can conclude that quantization increases the chance of gradient masking.

Bernhard et al. [8] arrived at a similar conclusion. They observed that the quantization of activations could make the gradient explode or vanish. However, they did not observe the same behavior for weight-only quantized models, i.e., models with weights quantized to integer precision but whose activations remain in floating-point.

Takeaway 2. Quantization increases the chance of gradient masking. Small perturbations may be erased by quantization and make the gradient vanish while bigger perturbations can make the gradient explode.

6.3.2 Transferability

We evaluate the transferability of adversarial examples from float-32 ANNs to QNNs at different bit-widths (int-16 and int-8) using white-, gray-, and black-box attacks. Figure 17 depicts the transferability results of the PGD attack, while Table 19 details the results of the remaining attacks, configured as detailed in Table 27 (Appendix A).

A3. Transferability is sensitive to quantization-shift. As observed in Figure 17, for the CIFAR-10 and Coffee datasets, the lines representing the adversarial accuracy of float-32 and int-16 models overlap for the entire range of distortion. However, the same does not occur with the int-8 model. While for float-32 and int-16, the adversarial accuracy decreases smoothly as distortion increases, for the int-8 model, the adversarial accuracy decreases in buckets - it only decreases after a certain distortion threshold. We argue that this is a consequence of the quantization shift, the most widely-known effect of quantization [8, 14,

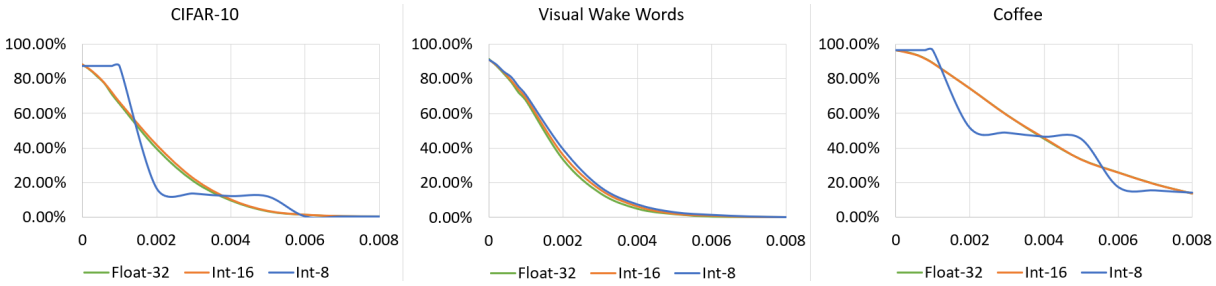


Figure 17: Accuracy *vs.* L_∞ distortion for PGD attack: adversarial example transferability from float-32 ANNs to int-16 and int-8 QNNs.

Attack		CIFAR-10					Visual Wake Words					Coffee Dataset				
		Acc	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞
WHITE-BOX																
DeepFool	*Float-32*	6.60%	3056.59	3.20	0.08	0.01	7.50%	27552.32	22.88	0.22	0.02	2.30%	123801.09	124.11	0.80	0.04
	Int-16	46.10%					49.90%					8.00%				
	Int-8	78.40%					69.90%					40.60%				
JSMA	*Float-32*	0.00%	3036.19	5.64	0.73	0.20	0.00%	46.20	5.69	0.81	0.32	0.00%	87.68	29.83	3.54	0.81
	Int-16	22.40%					39.00%					26.00%				
	Int-8	34.00%					49.40%					28.40%				
C&W-L_2	*Float-32*	0.00%	3065.49	2.95	0.08	0.01	0.10%	25457.08	16.39	0.18	0.02	0.00%	131356.72	96.62	0.71	0.05
	Int-16	34.20%					32.30%					7.40%				
	Int-8	75.70%					50.50%					46.20%				
C&W-L_∞	*Float-32*	0.00%	3072.00	13.40	0.26	0.01	0.00%	27647.93	121.68	0.81	0.01	0.00%	148330.23	765.26	2.42	0.01
	Int-16	0.30%					1.70%					0.80%				
	Int-8	5.60%					3.30%					3.30%				
EAD	*Float-32*	0.00%	3038.66	1.96	0.09	0.02	0.00%	2989.34	7.95	0.21	0.03	0.00%	7874.78	42.19	0.74	0.06
	Int-16	36.70%					35.80%					9.10%				
	Int-8	68.70%					52.90%					27.20%				
AutoAttack	*Float-32*	7.10%	3052.49	9.35	0.17	0.00	1.30%	24522.23	91.38	0.56	0.00	6.00%	136814.84	1065.17	2.87	0.01
	Int-16	8.00%					1.80%					6.00%				
	Int-8	9.80%					2.20%					9.50%				
GRAY-BOX																
ZOO	*Float-32*	24.70%	141.14	4.40	0.31	0.05	54.00%	13502.90	462.85	4.48	0.20	92.00%	16291.00	390.52	3.02	0.08
	Int-16	31.30%					61.00%					92.00%				
	Int-8	39.60%					64.00%					92.00%				
Square-L_2	*Float-32*	47.30%	3044.04	14.75	0.87	0.22	45.30%	12721.11	98.78	1.80	0.28	34.10%	74699.49	218.60	1.80	0.22
	Int-16	51.50%					48.50%					36.50%				
	Int-8	56.30%					51.70%					38.90%				
Square-L_∞	*Float-32*	6.10%	3045.62	33.26	0.65	0.01	37.60%	16671.67	248.55	1.78	0.01	59.40%	90578.27	1356.36	3.94	0.01
	Int-16	37.40%					63.20%					68.70%				
	Int-8	29.90%					67.70%					59.90%				
BLACK-BOX																
Boundary	*Float-32*	7.70%	3070.36	8.15	0.19	0.01	91.30%	2764.59	3.59	0.02	0.00	2.20%	150489.77	2209.77	7.15	0.09
	Int-16	63.10%					91.00%					55.10%				
	Int-8	69.70%					91.00%					43.80%				
GeoDA	*Float-32*	11.00%	2413.11	36.95	0.77	0.03	91.30%	828.96	414.73	2.80	0.03	0.10%	145536.22	1561.76	5.09	0.06
	Int-16	66.30%					91.00%					47.70%				
	Int-8	66.20%					91.00%					45.20%				

Table 19: Quantization impact on transferability. Red highlights samples that affect the accuracy and whose noise is detectable to the human eye. Yellow/blue highlights samples that do not affect the accuracy and whose noise is detectable/not detectable to the human eye.

49–52]. As quantization maps more than one value on the same quantization bucket, quantization can amplify or attenuate the effect of distortion. Quantization can boost adversarial accuracy by eliminating small perturbations; however, it can amplify bigger perturbations.

Nevertheless, the effect of quantization shift depends on the target model and attack. We observe in Figure 17 that PGD achieves much better transferability results on VWW than on the other two models. VWW is the only binary classification model and at first sight, it might appear that binary models are less

sensitive to the quantization-shift phenomenon. However, the results detailed in Table 19 for the VWW model clearly show this is a particular behavior of the PGD-VWW combo. In contrast to PGD, in DeepFool, JSMA, C&W-L_2, EAD, and Square-L_∞, the adversarial accuracy of the VWW models at different bit-widths is considerably different for the same distortion. Combining this observation with the results registered for the CIFAR-10 and Coffee models, we state that the success of transferability (transfer adversarial examples crafted on float-32 ANNs to int-16 or int-8 QNNs) is highly dependent on the attack-model combo.

Takeaway 3. Adversarial example transferability is influenced by quantization-shift. Quantization can mitigate the impact of small perturbations while amplifying the effects of larger perturbations.

A4. Gradient misalignment. In addition to the quantization-shift phenomenon, the poor transferability results generally observed from float-32 to int-16 and int-8 models can also be a consequence of gradient misalignment, as previously disclosed by Bernhard et al. [8]. Quantization can map multiple float-32 values into the same int-16 or int-8 value, moving distinct ANN weights and activations into the same quantization bucket. Therefore, the gradient estimated for float-32 ANNs may differ for the equivalent int-16 or int-8 models and even have different directions, with some attacks being more affected than others by this phenomenon.

To support our argument, we measured the average cosine similarity between the gradients of float-32, int-16, and int-8 models trained on the three target datasets. Cosine similarity calculates the cosine of the angle between two vectors, which reflects their directional similarity. As an exact calculus of the gradient is not possible for QNNs under the TensorFlow Lite Micro quantization policy, we first dequantized the QNNs and converted them to float-32 ANNs. The results are detailed in Table 20. As observed, only the Coffee models have gradient vectors pointing in almost the same direction (cosine similarity close to 1). The cosine similarity between float-32 and int-16/int-8 bit-widths for CIFAR-10 and VWW models is 0, which means that the gradient vectors are orthogonal to each other. These results suggest that quantization causes gradient misalignment, decreasing the success of adversarial transferability between models at different bit-width.

	CIFAR-10	**VWW**	**Coffee**
	Float-32		
Float-32	1.00	1.00	1.00
Int-16	0.00	0.00	0.99
Int-8	0.00	0.00	0.99

Table 20: Gradient cosine similarity: ANNs vs. QNNs.

Takeaway 4. By mapping different ANN weights and activations into the same integer value, quantization can cause gradient misalignment: the gradient estimated for float-32 ANNs may not be the same for the equivalent int-16 or int-8 models and even have different directions.

A5. Point distance to the decision boundary (A1 recall). DeepFool determines the minimum L_2 distortion required to move a given sample into a different class. As a result, DeepFool is commonly used to assess the minimum L_2 distance to the nearest decision boundary. We observe in Table 19, in every dataset, that for the same L_2 distance measured by DeepFool, the adversarial accuracy increases as the precision decreases. This behavior suggests that quantization alters the decision boundaries of neural networks, making the average point distance to the decision boundary increase as the precision of the neural network decreases. This observation supports the argument *A1*.

6.4 Defenses Against Adversarial Examples

In this section, we empirically evaluate how defenses aiming to enhance the robustness of ANNs perform when applied to int-16 and int-8 QNNs. As detailed in Table 16, our empirical evaluation includes two types of defenses: (i) train-based and (ii) input pre-processing. The former implies the retraining of the model to be protected; the latter cleans the sample to remove adversarial noise before it is used as input for the neural network.

For both defense types, the evaluation is performed under white-box, gray-box, and black-box scenarios. White-box attacks are used to evaluate adversarial transferability from ANNs to QNNs, while gray-box and black-box attacks are used to attack the QNNs directly. We reduce the scope of attacks included in this part of the evaluation to the attacks that impact the most the adversarial accuracy of QNNs before any defense:

- **White-box:** We reduce the scope to DeepFool, C&W-L_∞, and AutoAttack. DeepFool is useful to determine the point distance to the decision boundary, as it returns the minimum L_2 distance to the closest boundary. C&W-L_∞ and AutoAttack were the only white-box attacks registering good adversarial transferability across different bit-widths.

- **Gray-box:** We only include the Square Attack as the results detailed in Table 18 suggest that ZOO does not perform well against the neural networks we are using.

- **Black-box:** We include both the Boundary and GeoDA attacks; however, we do not use them against VWW models as they have failed to produce adversarial examples (Table 18).

We consider that a defense succeeds in defending a model against an attack when the accuracy drop for the baseline test accuracy (detailed in Table 17) is below 20%. We chose a 20% threshold, as we believe that a model exhibiting only a 20% decrease in accuracy under attack is notably robust. By subtracting 20% from the baseline test accuracy of the three models under consideration (Table 17), we arrive at accuracies of 67.50%, 71%, and 77% for CIFAR-10, VWW, and Coffee models, respectively. These values remain significantly higher than the accuracy of random choice for each of these models.

6.4.1 Train-Based Defenses

To compare how train-based defenses perform on QNNs, we first retrain the float-32 ANN according to the defense constraints and then quantize the resulting ANN to int-16 and int-8 to obtain the retrained QNNs. Directly retraining the QNNs is not possible as it is not possible to perform an exact calculus of the gradient. The accuracy of the retrained ANNs/QNNs is detailed in Table 21.

	CIFAR-10				**Visual Wake Words**				**Coffee Dataset**			
	Distil.	*PGD AdvT*	*Ens. AdvT*	*Sink. AdvT*	*Distil.*	*PGD AdvT*	*Ens. AdvT*	*Sink. AdT*	*Distil.*	*PGD AdvT*	*Ens. AdvT*	*Sink. AdvT*
Float-32	82.90%	82.17%	85.77%	81.24%	82.91%	78.72%	74.85%	77.86%	95.29%	95.87%	97.37%	95.38%
Int-16	82.77%	82.21%	85.85%	81.29%	82.90%	78.48%	74.93%	77.99%	95.32%	95.84%	97.40%	95.32%
Int-8	82.58%	81.76%	85.62%	81.27%	82.94%	78.47%	74.76%	77.86%	95.00%	96.01%	97.43%	95.06%

Table 21: Accuracy of ANNs/QNNs enhanced with train-based defenses.

The retrained ANNs and QNNs relative to a given defense are tested similarly to vanilla ANNs and QNNs: (i) gray- and black-box attacks are applied directly to each ANN or QNN and (ii) white-box attacks are used to evaluate the transferability from float-32 ANNs to QNNs. Results are detailed in Table 22. For synthesis purposes, Table 22 only reports the L_2 distortion. For a more comprehensive distortion report, please refer to Appendix B.

D1. Point distance to the decision boundary. To analyze the effect of the evaluated defenses on the average point distance to the decision boundary, we rely on DeepFool, as it determines the L_2 distance to the closest decision boundary. When we compare the results of DeepFool for models without any defense (Table 19) against defense-enhanced models (Table 22), we verify that the L_2 norm only increased consistently across the three models for the PGD Adversarial Training and SAT, with a little advantage for the later defense. For reference, the most significant increase was registered for the VWW models, where the average point distance to the decision boundary increased from $L_2 = 0.22$ (Table 19) to $L_2 = 2.88$ (Table 22). In turn, Defensive Distillation only increased the average point distance to the decision boundary of CIFAR-10 models and by a slimmer margin ($L_2 = 0.08$ vs. $L_2 = 0.09$), while Ensemble Adversarial Training only for Coffee ($L_2 = 0.80$ vs. $L_2 = 0.85$).

Interestingly, int-8 models enhanced with PGD Adversarial Training and SAT present a lower adversarial accuracy than those enhanced with the other defenses. This might suggest that despite PGD Adversarial Training and SAT increasing the average point distance to the decision boundary, this feature does not hold after quantization. However, the reduced adversarial accuracy of int-8 models enhanced with PGD Adversarial Training and SAT occurs because of the quantization-shift phenomenon. As these defenses significantly increase the average point distance to the decision boundary, DeepFool returns a higher perturbation that can move the quantized adversarial examples to the next quantization bucket and have a higher impact on the adversarial accuracy of int-8 and int-16 models.

Attack		CIFAR-10			Visual Wake Words			Coffee Dataset		
		Float-32	*Int-16*	*Int-8*	*Float-32*	*Int-16*	*Int-8*	*Float-32*	*Int-16*	*Int-8*
DEFENSIVE DISTILLATION										
TRANSFERABILITY										
DeepFool	*Acc*	8.60%	47.40%	75.80%	14.30%	70.20%	78.10%	3.50%	12.10%	49.40%
	L_2		0.09			0.19			0.61	
C&W-L_∞	*Acc*	0.00%	0.50%	5.00%	0.00%	1.00%	4.50%	0.00%	0.00%	0.00%
	L_2		0.26			0.82			2.57	
AutoAttack	*Acc*	65.10%	65.20%	65.70%	68.40%	69.20%	69.70%	91.20%	91.30%	91.80%
	L_2		0.04			0.11			0.12	
DIRECT ATTACK										
Square-L_2	*Acc*	50.00%	50.00%	80.00%	54.50%	58.00%	88.50%	30.50%	33.50%	77.50%
	L_2	0.38	0.37	0.31	0.67	0.56	0.35	1.24	1.22	0.38
Square-L_∞	*Acc*	71.00%	72.50%	85.50%	79.50%	82.00%	87.00%	86.50%	90.00%	95.50%
	L_2	0.13	0.12	0.02	0.12	0.04	0.02	0.18	0.15	0.01
Boundary	*Acc*	4.50%	5.50%	80.00%	-	-	-	2.50%	2.00%	3.00%
	L_2	0.19	0.19	0.98	-	-	-	6.37	7.29	30.32
GeoDA	*Acc*	5.00%	7.00%	3.00%	-	-	-	1.00%	2.00%	2.00%
	L_2	1.94	2.96	4.32	-	-	-	6.78	9.21	21.24
PGD ADVERSARIAL TRAINING										
TRANSFERABILITY										
DeepFool	*Acc*	11.00%	37.20%	43.40%	14.60%	41.80%	43.60%	4.80%	16.00%	25.90%
	L_2		0.42			2.80			1.73	
C&W-L_∞	*Acc*	2.00%	3.50%	18.00%	25.40%	36.40%	42.60%	12.50%	14.00%	29.50%
	L_2		0.66			1.76			4.61	
AutoAttack	*Acc*	69.30%	70.70%	73.20%	78.90%	81.00%	81.40%	70.10%	70.50%	76.90%
	L_2		0.03			0.05			0.80	
DIRECT ATTACK										
Square-L_2	*Acc*	63.50%	44.50%	54.00%	76.00%	78.50%	83.50%	60.00%	61.00%	67.00%
	L_2	0.82	7.87	2.23	1.80	15.87	4.51	1.76	10.09	5.53
Square-L_∞	*Acc*	66.00%	72.00%	79.00%	81.00%	85.00%	88.00%	88.00%	93.00%	95.50%
	L_2	0.61	0.53	0.39	1.92	1.85	1.04	3.96	3.04	1.03
Boundary	*Acc*	18.50%	17.50%	79.50%	-	-	-	2.50%	3.00%	1.50%
	L_2	0.81	0.93	3.42	-	-	-	16.12	17.23	49.39
GeoDA	*Acc*	13.00%	13.00%	13.00%	-	-	-	2.00%	2.00%	2.00%
	L_2	1.29	2.62	5.12	-	-	-	8.88	15.38	45.65
ENSEMBLE ADVERSARIAL TRAINING										
TRANSFERABILITY										
DeepFool	*Acc*	9.50%	50.10%	77.40%	24.40%	66.10%	70.40%	2.50%	6.70%	38.10%
	L_2		0.08			0.06			0.85	
C&W-L_∞	*Acc*	0.00%	1.00%	4.00%	0.00%	1.00%	0.50%	0.00%	0.00%	7.00%
	L_2		0.25			0.65			2.57	
AutoAttack	*Acc*	5.30%	6.20%	8.20%	0.00%	1.50%	2.30%	6.20%	6.20%	10.40%
	L_2		0.17			0.46			2.86	
DIRECT ATTACK										
Square-L_2	*Acc*	40.00%	34.00%	40.00%	21.50%	20.50%	42.50%	38.00%	32.00%	68.00%
	L_2	0.83	3.38	2.84	1.49	2.24	1.87	1.64	5.11	4.49
Square-L_∞	*Acc*	6.50%	19.00%	66.00%	2.00%	13.50%	36.50%	53.50%	74.00%	83.50%
	L_2	0.56	0.45	0.28	1.54	1.15	0.77	2.63	0.90	0.50
Boundary	*Acc*	8.50%	8.50%	83.50%	-	-	-	1.00%	1.00%	1.50%
	L_2	0.18	0.19	1.22	-	-	-	7.36	8.26	25.87
GeoDA	*Acc*	1.00%	0.00%	3.00%	-	-	-	1.00%	1.00%	10.00%
	L_2	0.78	1.03	5.87	-	-	-	6.72	6.40	40.38
SINKHORN ADVERSARIAL TRAINING										
TRANSFERABILITY										
DeepFool	*Acc*	10.90%	44.30%	45.80%	16.70%	24.90%	28.30%	3.70%	14.40%	25.60%
	L_2		0.45			2.88			1.84	
C&W-L_∞	*Acc*	3.50%	7.50%	19.00%	28.70%	34.10%	37.30%	14.90%	15.40%	25.10%
	L_2		0.65			1.44			4.37	
AutoAttack	*Acc*	67.10%	69.60%	72.30%	76.80%	78.30%	78.30%	68.10%	68.30%	75.50%
	L_2		0.03			0.05			0.85	
DIRECT ATTACK										
Square-L_2	*Acc*	61.60%	48.00%	47.50%	69.60%	71.20%	74.20%	45.20%	50.50%	64.80%
	L_2	0.79	5.15	2.73	1.71	12.96	3.39	1.90	6.03	3.29
Square-L_∞	*Acc*	63.40%	67.90%	73.60%	79.60%	81.20%	85.00%	85.80%	86.40%	93.60%
	L_2	0.58	0.49	0.30	1.84	1.76	1.12	4.31	3.25	1.15
Boundary	*Acc*	20.40%	18.60%	17.30%	-	-	-	1.00%	2.00%	3.00%
	L_2	0.86	0.95	3.60	-	-	-	18.31	20.14	44.71
GeoDA	*Acc*	4.00%	7.00%	10.00%	-	-	-	93.60%	6.40%	86.80%
	L_2	2.84	5.28	10.03	-	-	-	7.01	30.06	34.60

Table 22: Evaluation of train-based defenses (L_2 distortion). Red highlights samples that affect the accuracy and whose noise is detectable to the human eye. Green highlights samples where accuracy drop is less than 20%.

Takeaway 5. PGD Adversarial Training and SAT consistently increase the average point distance to the decision boundary, and this feature holds after quantization. Defensive Distillation and Ensemble Adversarial Training are not consistent in this topic.

D2. Adversarial example transferability. The train-based defenses considered in this study do not significantly affect the transferability of adversarial examples from float-32 ANNs to int-16 or int-8 QNNs. Without any defense in place, we observe in Table 19 that AutoAttack and C&W-L_∞ attacks hold good transferability results across all bit-widths, registering a maximum accuracy loss of 5.60% registered for C&W-L_∞ attack on CIFAR-10. With defense in place, the maximum adversarial accuracy loss when transferring adversarial examples crafted on these attacks rises to 17.20%. This value is registered for the VWW models enhanced with PGD Adversarial Training when attacked by C&W-L_∞ (Table 22).

This behavior is verified even for Ensemble Adversarial Training, specifically developed to counteract adversarial example transferability. However, after empirical validation, we can attest that this defense is not ready to handle adversarial example transferability between models at different bit widths. The defense must address the quantization-shift and gradient misalignment phenomenon to counteract adversarial example transferability adequately. To the best of our knowledge, no existing defense has accounted for these factors.

Takeaway 6. None of the training-based defenses mitigate adversarial example transferability. We envision this is a consequence of none of them smoothing the quantization-shift and gradient misalignment, which are required to tackle the transferability.

D3. Defense transferability. We observe that the robustness guarantees hold after quantization: whenever the defense is robust to protect a float-32 ANN, it is also robust in the int-16 and int-8 QNNs. However, the opposite is not valid. Let us consider the effect of Defensive Distillation against the Square-L_∞ attack. We observe that the defense delivers solid results across the three float-32 ANNs used for evaluation, and this behavior extends to the lower QNN's bit widths. However, let us consider the results of the same defense in the face of the Square-L_2 attack. We observe that Defensive Distillation greatly enhances the robustness of int-8 QNNs, but the behavior does not apply to the higher bit widths. This results from the combined robustness of quantization and the defense itself.

Takeaway 7. Every time a defense is effective in a full-precision ANN, it is likewise effective in the corresponding QNNs. A defense may, however, be adequate to defend a low-precision QNN but insufficient to defend a full-precision ANN.

6.4.2 Pre-Processing Defenses

Pre-processing defenses are evaluated against the adversarial examples produced to attack the vanilla ANNs and QNNs before any defense. We do not craft new adversarial examples for the Defense+ANN or Defense+QNN system as pre-processing defenses are known to disrupt the well-known gradient-masking phenomenon, in which the gradient becomes useless to create an adversarial example, giving a false sense of security [8, 14, 50]. The evaluation of Pixel Defend was restricted to CIFAR-10 due to hardware limitations: PixelCNN is a heavy network whose amount of neurons grows proportionally with the size of input images. Therefore, we limited the evaluation scope to the dataset with smaller images. The configuration of Feature Squeezing and Pixel Defend is detailed in Table 28 (Appendix A), while the test accuracy of the ANNs/QNNs enhanced with these defenses is detailed in Table 23. Tables 24 and 25 detail the results observed for Feature Squeezing and Pixel Defend, respectively.

	CIFAR-10		**VWW**	**Coffee**
	Feat. Sq.	*Pixel Defend*	*Feat. Sq.*	*Feat. Sq.*
Float-32	83.70%	84.80%	97.00%	96.80%
Int-16	83.60%	85.10%	96.90%	97.00%
Int-8	83.00%	84.90%	96.60%	96.90%

Table 23: Accuracy of ANNs/QNNs enhanced with pre-processing defenses.

D4. Feature Squeezing vs. Pixel Defend. Feature Squeezing can denoise more adversarial examples than Pixel Defend. Feature Squeezing returned a higher adversarial accuracy for almost every attack and bit-width, making it a preferable defense from the robustness point-of-view. Regarding the implementation at the deep edge, the amount of processing that Feature Squeezing requires can effectively

Attack		**CIFAR-10**					**Visual Wake Words**					**Coffee Dataset**				
		Acc	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞
FEATURE SQUEEZING																
TRANSFERABILITY																
	Float-32	81.40%					97.00%					53.90%				
DeepFool	*Int-16*	81.30%	3056.59	3.20	0.08	0.01	97.00%	27552.32	22.88	0.22	0.02	54.00%	123801.09	124.11	0.80	0.04
	Int-8	81.10%					96.90%					63.50%				
	Float-32	70.50%					94.30%					13.60%				
C&W-L_∞	*Int-16*	70.40%	3072.00	13.40	0.26	0.01	93.30%	27647.93	121.68	0.81	0.01	13.40%	148330.23	765.26	2.42	0.01
	Int-8	71.40%					93.70%					15.20%				
	Float-32	70.10%					95.60%					9.60%				
AutoAttack	*Int-16*	70.70%	3052.49	9.35	0.17	0.00	94.80%	24522.23	91.38	0.56	0.00	9.60%	136814.84	1065.17	2.87	0.01
	Int-8	70.60%					95.30%					14.40%				
DIRECT ATTACK																
	Float-32	69.40%	3044.04	14.75	0.87	0.22	93.40%	12721.11	98.78	1.80	0.28	50.50%	74699.49	218.60	1.80	0.22
Square-L_2	*Int-16*	71.00%	1139.35	14.01	0.83	0.21	92.00%	12244.77	216.54	2.52	0.28	53.00%	64190.22	2715.53	8.57	0.22
	Int-8	73.80%	443.77	11.44	0.75	0.22	94.00%	7176.22	352.30	3.64	0.32	68.80%	16830.86	627.71	3.11	0.23
	Float-32	76.30%	3045.62	33.26	0.65	0.01	96.00%	16671.67	248.55	1.78	0.01	77.20%	90578.27	1356.36	3.94	0.01
Square-L_∞	*Int-16*	78.00%	1845.00	27.57	0.54	0.01	96.20%	16614.91	247.64	1.76	0.01	69.00%	62088.17	929.52	2.73	0.01
	Int-8	80.40%	1125.94	17.60	0.36	0.01	95.80%	11173.88	167.20	1.23	0.01	82.40%	27792.02	435.28	1.35	0.01
	Float-32	80.90%	3070.36	8.15	0.19	0.01	-	-	-	-	-	84.30%	150489.77	2209.77	7.15	0.09
Boundary	*Int-16*	85.00%	2989.68	9.10	0.21	0.02	-	-	-	-	-	83.00%	150347.67	2684.71	8.69	0.11
	Int-8	82.50%	2450.47	61.34	1.38	0.08	-	-	-	-	-	75.50%	144495.22	7928.35	25.56	0.29
	Float-32	81.20%	2413.11	36.95	0.77	0.03	-	-	-	-	-	50.80%	145536.22	1561.76	5.09	0.06
GeoDA	*Int-16*	85.00%	2284.17	83.51	1.72	0.06	-	-	-	-	-	61.00%	142126.62	2374.74	7.67	0.07
	Int-8	75.00%	2193.12	98.67	2.08	0.08	-	-	-	-	-	50.00%	132656.10	10057.61	31.51	0.26

Table 24: Evaluation of Feature Squeezing. Green highlights samples where accuracy dropped by less than 20%. For CIFAR-10 and VWW the defense succeeded for every attack and bit-width tested.

Attack		CIFAR-10				
		Acc	L_0	L_1	L_2	L_∞
PIXEL DEFEND						
TRANSFERABILITY						
DeepFool	*Float-32*	77.20%	3056.59	3.20	0.08	0.01
	Int-16	77.40%				
	Int-8	83.20%				
C&W-L_∞	*Float-32*	53.90%	3072.00	13.40	0.26	0.01
	Int-16	53.80%				
	Int-8	58.60%				
AutoAttack	*Float-32*	39.40%	3052.49	9.35	0.17	0.00
	Int-16	39.60%				
	Int-8	55.20%				
DIRECT ATTACK						
Square-L_2	*Float-32*	65.30%	3044.04	14.75	0.87	0.22
	Int-16	66.40%	1139.35	14.01	0.83	0.21
	Int-8	71.70%	443.77	11.44	0.75	0.22
Square-L_∞	*Float-32*	75.80%	3045.62	33.26	0.65	0.01
	Int-16	76.60%	1845.00	27.57	0.54	0.01
	Int-8	81.20%	1125.94	17.60	0.36	0.01
Boundary	*Float-32*	80.00%	3070.36	8.15	0.19	0.01
	Int-16	84.00%	2989.68	9.10	0.21	0.02
	Int-8	74.00%	2450.47	61.34	1.38	0.08
GeoDA	*Float-32*	78.80%	2413.11	36.95	0.77	0.03
	Int-16	79.00%	2284.17	83.51	1.72	0.06
	Int-8	37.50%	2193.12	98.67	2.08	0.08

Table 25: Evaluation of Pixel Defend

be handled by Arm Cortex-M MCUs; however, only on MCUs that feature a compatible DSP or FPU. If the color bit-depth compression can be implemented as a simple shift operation, the spatial smoothing filter is more computationally heavy and would benefit from a DSP-enabled MCU. In contrast, Pixel Defend is not feasible in edge devices powered by Arm Cortex-M MCUs as it imposes a heavy generative model as input pre-processing. The PixelCNN used in this evaluation has more than 2M parameters, which do not fit in MCUs with a few megabytes.

Takeaway 8. Feature Squeezing can denoise a larger fraction of adversarial examples than Pixel Defend. In contrast to Pixel Defend, Feature Squeezing is portable to edge devices powered by Arm Cortex-M MCUs as long as they feature a DSP.

D5. Amount of perturbed pixels (L_0 distortion). Feature Squeezing delivers better results for the CIFAR-10 and VWW models than for Coffee. We argue that the worst performance on Coffee models results from the larger distortion created by the attacks. The Coffee dataset features considerably bigger images than the CIFAR-10 and VWW datasets, naturally leading to attacks to perturbing more pixels. For defenses based on filter smoothing, such as Feature Squeezing, if many pixels are distorted, the filter will struggle to revert the effect of the noise as most neighbor pixels are also very distorted.

However, a closer analysis of Feature Squeezing results on Coffee models reveals that the Boundary attack perturbs more pixels (L_0 distortion) than other attacks, yet Feature Squeezing still manages to denoise more adversarial examples than on other attacks. This difference becomes more pronounced when compared to white-box attacks, which perturb almost the same magnitude of pixels as the Boundary

attack. The reason behind this lies in the other aspect of Feature Squeezing: the reduction of color bit-width. Acting like quantization, it can remove small perturbations while amplifying larger ones. Since white-box attacks can access exact gradients, they can induce more significant perturbations on more crucial pixels. Conversely, the Boundary attack cannot compute the exact gradient and may overemphasize less relevant pixels. Consequently, while the spatial smoothing filter's performance may be similar across all attacks, the reduction of color bit-width proves more successful in denoising adversarial examples crafted with the Boundary attack than white-box attacks.

Takeaway 9. Feature Squeezing is effective against small perturbations, but the spatial smoothing filter starts to fall short as the amount of perturbed pixels increases.

6.4.3 Train-Based vs. Pre-Processing Defenses

From the set of train-based defenses, PGD Adversarial Training and SAT take the lead in terms of robustness, with a little advantage for the later defense. Nevertheless, we believe that Ensemble Adversarial Training would provide better results if the adversarial examples used during training were generated using more powerful attacks. For the particular case of Pixel Defend, we see no better performance when compared to the much simpler Feature Squeezing.

Regarding the implementation at the deep edge, train-based defenses are preferable as they do not incur additional overhead or interfere with the original model architecture. However, for DSP-enabled MCUs, combining PGD Adversarial Training or SAT with Feature Squeezing could deliver a robust ML system at the cost of probably insignificant overhead.

Takeaway 10. Train-based defenses do not incur additional computation during inference and, therefore, are preferable over pre-processing defenses to be deployed in edge devices powered by Arm Cortex-M MCUs.

6.5 Related Work and Gap Analysis

6.5.1 Related Work

Shumailov et al. [49] pioneered in the evaluation of adversarial example transferability. The authors concluded that QNNs operating in int-8 or lower bit widths show defensive behavior. The authors also envision that clipping activations change the feature space of ANNs, which could offer marginal protection against transferability.

Along the same research line, i.e., adversary example transferability, Duncan et al. [51] concluded that when the magnitude of the perturbation is small, quantization increases adversarial robustness by

eliminating small perturbations; however, when the magnitude of the perturbation is greater than a given threshold, quantization acts as a distortion amplifier (quantization-shift).

Bernhard et al. [8] were the first to evaluate QNNs robustness on a non-white-box scenario (using gray-box attacks). The authors concluded that QNNs offer only marginal protection as gray-box attacks can still fool the network. Additionally, the authors state that adversarial transferability is sensitive to the quantization-shift phenomenon and the gradient misalignment.

Gorsline et al. [14] considered the findings of previous works as a baseline and defined a geometric model to evaluate QNN robustness. Through this geometric model, the authors mathematically proved that the mean distance to the decision boundary decreases as the weight precision and the input dimensionality increase. Additionally, the authors introduced the concept of critical attack strength - the attack strength at which the adversarial accuracy of QNNs matches the adversarial accuracy of ANNs.

Lin et al. [50] followed a different approach from previous works and proposed a training algorithm to tackle quantization-shift. More specifically, the authors proposed a quantization-aware training algorithm that maintains the Lipschitz constant of the network below 1 - the Lipschitz constant measures how much the output changes when the input changes. Additionally, the authors demonstrate that this mechanism can be combined with other defenses, such as Feature Squeezing or Adversarial Training, to enhance the robustness of QNNs.

Song et al. [52] evaluated the effect of adversarial examples on 8-bit inferior QNNs. They observed that for precisions under 5 bits, adversarially trained QNNs are more vulnerable to adversarial examples than the equivalent ANN. To tackle this issue, they proposed an iteration of PGD Adversarial Training specialized for QNNs.

6.5.2 Gap Analysis

In this section, we put our work in perspective with the works previously described along three different dimensions. Table 26 guides our discussion.

Attacks. Information on QNN's adversarial robustness is scattered across multiple works that do not share the same quantization policy. In some cases, this has led to contradictory conclusions [50, 51]. While Lin et al. [50] states that QNNs are more vulnerable to adversarial examples, Duncan et al. [51] suggests that QNNs show increased robustness when compared to ANNs. Nevertheless, both of these works only consider the quantization-shift phenomenon (T3) and let out all the other factors that also influence the robustness of a QNN. If we take a closer look at Table 26, we observe that the most complete work till date comes from Bernhard et al. [8]; however, the authors do not evaluate the point distance to the decision boundary (T1) and do not follow a state-of-the-art quantization policy. Given this background, our work systematizes and empirically evaluates all the conclusions drawn from previous works, under a common quantization policy, which is widely used in TinyML (TensorFlow Lite Micro).

	Work						
	Shumailov et al. 2019 [49]	Bernhard et al. 2019 [8]	Duncan et al. 2020 [51]	Gorsline et al. 2021 [14]	Lin et al. 2019 [50]	Song et al. 2021 [52]	**Our Work**
ATTACKS							
Point distance to the decision boundary (T1)				•			•
Gradient masking (T2)		•					•
Transferability vs. quantization-shift (T3)	•	•	•	•	•	•	•
Gradient misalignment (T4)		•					•
DEFENSES							
Point distance to the decision boundary (T5)							•
Adversarial example transferability (T6)							•
Defense transferability to TinyML (T7)							•
Compare relevant pre-processing defenses (T8)							•
Distortion of adversarial examples (T9)							•
Train-Based vs. Pre-Processing defenses (T10)							•
QUANTIZATION							
Policy	Mayo Tool	Binary Connect Dorefa Net	TF Lite Micro	N.D.	N.D.	N.D.	TF Lite Micro
Bit-width	[4; 28]	[1; 4]	8	[1; 10]	[1; 5]	[1; 8]	8; 16
Quantized Parameters	Weights Activations	Weights Activations	Weights Activations	Weights	Weights Activations	Weights	Weights Activations

Table 26: Gap analysis: adversarial robustness of QNNs. Filled-circles (●) indicate that a given work empirically evaluates a given takeaway. Empty-circles (○) indicate that a given work does not empirically evaluate a given takeaway. N.D. means Non-Defined.

Compared to previous research, our empirical evaluation relies on a far more extensive list of attacks. To the best of our knowledge, we are the only ones including a parameter-free attack (AutoAttack), which reduces the chance of bad results due to human error. In the black-box scenario, which is the most realistic, we are the first to include powerful attacks. Previous works have either neglected this scenario or have only included simple attacks that add random noise to images [50, 52].

Defenses. The works [50, 52] are the only ones addressing the topic of defenses against adversarial examples for QNNs; however, in a different approach. Lin et al. [50] proposes a novel training algorithm to reduce the effect of quantization-shift on QNNs and couples it with adversarial training and Feature Squeezing. Song et al. [52] proposes an iteration of PGD Adversarial Training targeting weight-quantized models (QNNs whose weights are quantized but activations remain in floating-point) with precision equal to or inferior to 8-bits.

Our work follows a different approach and verifies how current defenses for full-precision ANNs perform applied to QNNs running on MCUs (TinyML). We are the first to empirically evaluate; (i) how quantization interferes with the success of defenses (T5, T7, T9); (ii) how defenses affect the adversarial example transferability from ANNs to QNNs (T6); (iii) which type of pre-processing defenses are preferable to be deployed on MCUs (T8); and (iv) which type of defense (train-based vs. pre-processing) is preferable and in which circumstances (T10).

Quantization and TinyML. As mentioned in Section 6.2.2, QNNs can be deployed in a myriad of devices, ranging from ultra-low-power MCUs to ASICs and GPUs. However, each target imposes different requirements on the QNN architecture and, above all, quantization policy. Consequently, QNNs targeting GPU devices are closer to conventional ANNs than QNNs targeting TinyML applications. The difference can be expressed in terms of the quantization policy, bit-width, size of kernel filters, and depth of the network. Most of the previous works tend to not be clear about the quantization policy or tend to use policies that enable the approximation of the gradients through STE or Mirror Descent techniques, which are not applicable in TensorFlow Lite Micro.

Although the work from Duncan et al. [51] uses the same quantization policy as our work, the authors only evaluate the adversarial transferability vs. quantization-shift phenomenon (T3). To the best of our knowledge, our work pioneers research on adversarial example robustness for TinyML applications. Our evaluation includes three QNNs, two composing the image applications from the MLPerf Tiny benchmark [24], and another specifically developed to be executed on Arm Cortex-M MCUs [140].

6.6 Discussion

Although the effect of adversarial examples on QNNs is a topic that has emerged in the literature over the past few years, the information and findings available are quite scattered and sometimes contradictory at first sight. While some works claim that quantization increases the robustness against adversarial examples [51], others claim the opposite [50]. What sometimes these works fail to communicate is the type of devices they target. QNNs targeting GPUs or TPUs usually follow different quantization policies than those targeting MCUs. Our work empirically evaluates all the conclusions drawn from previous works under a common quantization policy, tailored for the TinyML scenario (TensorFlow Lite Micro). Furthermore, we are the first to empirically evaluate how current defenses tailored for full-precision ANNs perform when applied to QNNs and adapted to the TinyML environment. Our work unveils findings supported by existing literature (T1, T2, T4), extends existing knowledge (T3), and introduces entirely novel contributions (T5-T10). Although our work targets TinyML applications, only the takeaway T8 is dependent on the architecture of the target processor class. The remaining takeaways are independent of the architecture of the target processor class, i.e., they hold for any ML system following the quantization policy of TensorFlow Lite Micro. As a final remark, we envision that developing train-based defenses that can smooth the quantization-shift and gradient misalignment issues is the way to enhance the robustness of QNNs in TinyML applications.

6.7 Conclusion

With this work, we have conducted the most comprehensive empirical evaluation of adversarial example attacks and defenses on QNNs targeting TinyML applications. The study encompassed three QNNs

targeting edge ML applications on ten attacks and six popular defenses. With that, we drew a set of observations that provide evidence about (i) the effectiveness of adversarial examples on QNNs, (ii) the impact of bit-width on the adversarial example robustness, (iii) the robustness of defenses in QNNs, and (iv) the deployability of state-of-the-art defenses in resource-constrained MCUs. We are opening all artifacts in the hopes of enabling independent validation of results and encouraging further exploration of the security of QNNs, namely the development of adversarial defenses that meet the requirements of TinyML.

Appendix A. Attacks/Defenses Configuration

Table 27 details the configuration of the attacks. We configured the attacks such that the accuracy of the target ANN drops below the value of random choice (10% for CIFAR-10, 50% for VWW, and 25% for Coffee). Table 28 details the configuration of the defenses. For each defense, we experimented with different sets of hyper-parameters around the default values suggested by the defenses' authors and selected the configuration that delivered the best results against the PGD attack.

Attack		**CIFAR-10**	**VWW**	**Coffee**
		WHITE-BOX		
DeepFool	Max. Iterations	100	100	100
	Overshoot	0.008	0.008	0.008
JSMA	Theta	0.08	0.08	0.08
	Gamma	1.00	1.00	1.00
C&W-L_2	Max. Iterations	10	10	10
	Binary Search Steps	10	10	10
	Initial Constant	0.01	0.01	0.01
C&W-L_∞	Max. Iterations	10	10	10
	Learning Rate	0.01	0.01	0.01
PGD	Epsilon	[0.0002; 0.0008]	[0.0002; 0.0008]	[0.0002; 0.0008]
	Step Size	[0.00002; 0.00008]	[0.00002; 0.00008]	[0.00002; 0.00008]
	Restarts	20	20	20
EAD	Max. Iterations	10	10	10
	BS Steps	10	10	10
	Initial Constant	0.01	0.01	0.01
AutoAttack	Epsilon	0.004	0.004	0.009
		GRAY-BOX		
ZOO	Max. Iterations	10	50	10
	BS Steps	5	1	2
	Initial Constant	0.01	0.01	0.01
SA-L_2	Epsilon	1	2	2
	Max. Iterations	2000	1000	2000
	Initial Fraction	0.80	0.80	0.80
SA-L_∞	Epsilon	0.015	0.015	0.015
	Max. Iterations	1000	1000	1000
	Initial Fraction	0.05	0.05	0.05
		BLACK-BOX		
BA	Epsilon	1.0	1.0	1.0
	Delta	0.10	0.10	0.10
	Initial Size	100	500	100
	Max. Iterations	500	500	500
GeoDA	BS Tolerance	0.0001	0.0001	0.01
	DCT Dimension	75	75	10

Table 27: Configuration of adversarial attacks.

Defense		CIFAR-10	VWW	Coffee
TRAIN-BASED DEFENSES				
Defensive Distillation	Temperature	20	20	20
	Alpha	0.1	0.1	0.1
	Epochs	200	50	25
PGD Adversarial Training [1]	Epsilon	0.008	0.008	0.008
	Step Size	0.0008	0.0008	0.0008
	Iterations	10	10	10
	Epochs	50	50	50
Ensemble Adversarial Training	PGD	Same as (1)	Same as (1)	Same as (1)
	FGSM - Epsilon	0.008	0.008	0.008
	Epochs	50	50	50
Sinkhorn Adversarial Training	PGD	Same as (1)	Same as (1)	Same as (1)
	Sink. Epsilon	1.0	1.0	1.0
	Sink. Iterations	50	50	50
	Sink. Reduction	Sum	Sum	Sum
INPUT PRE-PROCESSING DEFENSES				
Feature Squeezing	Bit-Depth (ANNs)	4	8	8
	Window Size	2	6	2
Pixel Defend	Epsilon	38	-	-

Table 28: Configuration of adversarial defenses.

Appendix B. Evaluation of Train-Based Defenses

Tables 29 and 30 represent the full-version of Table 22 and detail the results registered for train-based defenses against the most powerful attacks that we evaluate in Section 6.3. Red highlights samples that affect the accuracy and whose noise is detectable to the human eye. Green highlights samples where accuracy dropped by less than 20% when compared to the baseline accuracy of the model (Table 17).

Attack		CIFAR-10					Visual Wake Words					Coffee Dataset				
		Acc	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞
DEFENSIVE DISTILLATION																
TRANSFERABILITY																
DeepFool	*Float-32*	8.60%					14.30%					3.50%				
	Int-16	47.40%	3053.42	3.32	0.09	0.01	70.20%	27547.79	19.97	0.19	0.01	12.10%	116214.80	82.05	0.61	0.04
	Int-8	75.80%					78.10%					49.40%				
C&W-L_∞	*Float-32*	0.00%					0.00%					0.00%				
	Int-16	0.50%	3072.00	13.26	0.26	0.01	1.00%	27647.93	122.96	0.82	0.01	0.00%	148162.39	793.89	2.57	0.01
	Int-8	5.00%					4.50%					5.00%				
AutoAttack	*Float-32*	65.10%					68.40%					91.20%				
	Int-16	65.20%	577.18	2.18	0.04	0.00	69.20%	4852.48	17.79	0.11	0.00	91.30%	5861.31	43.52	0.12	0.00
	Int-8	65.70%					69.70%					91.80%				
DIRECT ATTACK																
Square-L_2	*Float-32*	50.00%	495.38	8.23	0.38	0.08	54.50%	3522.41	37.12	0.67	0.11	30.50%	32385.66	127.22	1.24	0.19
	Int-16	50.00%	443.33	7.72	0.37	0.08	58.00%	2503.31	27.58	0.56	0.10	33.50%	34753.23	131.58	1.22	0.19
	Int-8	80.00%	129.75	3.48	0.31	0.10	88.50%	502.51	9.52	0.35	0.08	77.50%	3436.38	29.54	0.38	0.06
Square-L_∞	*Float-32*	71.00%	462.09	6.90	0.13	0.00	79.50%	1052.38	15.66	0.12	0.00	86.50%	3346.93	50.18	0.18	0.00
	Int-16	72.50%	442.67	6.61	0.12	0.00	82.00%	329.25	4.93	0.04	0.00	90.00%	3149.90	47.24	0.15	0.00
	Int-8	85.50%	50.26	0.79	0.02	0.00	87.00%	149.82	2.24	0.02	0.00	95.50%	46.87	0.74	0.01	0.00
Boundary	*Float-32*	4.50%	3070.26	8.11	0.19	0.01	-	-	-	-	-	2.50%	150492.08	1969.80	6.37	0.08
	Int-16	5.50%	3036.74	8.39	0.19	0.01	-	-	-	-	-	2.00%	150288.12	2253.76	7.29	0.09
	Int-8	80.00%	2376.41	42.93	0.98	0.06	-	-	-	-	-	3.00%	143583.36	9455.81	30.32	0.32
GeoDA	*Float-32*	5.00%	2624.92	94.40	1.94	0.06	-	-	-	-	-	1.00%	145405.53	2059.10	6.78	0.08
	Int-16	7.00%	2583.83	143.51	2.96	0.10	-	-	-	-	-	2.00%	140810.23	2839.88	9.21	0.09
	Int-8	3.00%	2402.82	203.68	4.32	0.16	-	-	-	-	-	2.00%	128660.2	6619.05	21.24	0.20

Table 29: Evaluation of train-based adversarial defenses (all distortion metrics - Part 1).

Attack		CIFAR-10					Visual Wake Words					Coffee Dataset				
		Acc	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞	*Acc*	L_0	L_1	L_2	L_∞
PGD ADVERSARIAL TRAINING																
TRANSFERABILITY																
	Float-32	11.00%					14.60%					4.80%				
DeepFool	*Int-16*	37.20%	3056.20	14.91	0.42	0.06	41.80%	27538.51	254.41	2.80	0.22	16.00%	121362.79	195.93	1.73	0.13
	Int-8	43.40%					43.60%					25.90%				
	Float-32	2.00%					25.40%					12.50%				
C&W-L_∞	*Int-16*	3.50%	3010.56	33.16	0.66	0.02	36.40%	20625.36	260.48	1.76	0.02	14.00%	130004.22	1428.27	4.61	0.02
	Int-8	18.00%					42.60%					29.50%				
	Float-32	69.30%					78.90%					70.10%				
AutoAttack	*Int-16*	70.70%	390.56	1.54	0.03	0.00	81.00%	2042.75	8.06	0.05	0.00	70.50%	38292.20	297.76	0.80	0.00
	Int-8	73.20%					81.40%					76.90%				
DIRECT ATTACK																
	Float-32	63.50%	1092.93	9.67	0.82	0.24	76.00%	13039.11	82.14	1.80	0.30	60.00%	67406.34	164.02	1.76	0.26
Square-L_2	*Int-16*	44.50%	1403.09	383.02	7.87	0.39	78.50%	14821.89	2257.20	15.87	0.40	61.00%	55669.68	3249.16	10.09	0.26
	Int-8	54.00%	731.95	84.06	2.23	0.25	83.50%	9379.48	492.04	4.51	0.32	67.00%	20236.04	1462.49	5.53	0.26
	Float-32	66.00%	2046.87	30.57	0.61	0.01	81.00%	19256.98	287.24	1.92	0.01	88.00%	88885.91	1329.74	3.96	0.01
Square-L_∞	*Int-16*	72.00%	1711.28	25.55	0.53	0.01	85.00%	18345.18	273.35	1.85	0.01	93.00%	70101.88	1049.96	3.04	0.01
	Int-8	79.00%	1196.17	18.69	0.39	0.01	88.00%	9395.54	140.43	1.04	0.01	95.50%	19585.81	306.58	1.03	0.01
	Float-32	18.50%	2732.76	34.97	0.81	0.07	-	-	-	-	-	2.50%	150488.67	4990.87	16.12	0.20
Boundary	*Int-16*	17.50%	2729.74	40.76	0.93	0.07	-	-	-	-	-	3.00%	150333.42	5337.81	17.23	0.21
	Int-8	79.50%	2484.01	152.47	3.42	0.21	-	-	-	-	-	1.50%	145254.45	15508.68	49.39	0.47
	Float-32	13.00%	2041.18	60.58	1.29	0.05	-	-	-	-	-	2.00%	143865.45	2702.16	8.88	0.10
GeoDA	*Int-16*	13.00%	2041.11	120.81	2.62	0.10	-	-	-	-	-	2.00%	139045.66	4782.20	15.38	0.15
	Int-8	13.00%	2019.72	238.58	5.12	0.20	-	-	-	-	-	2.00%	136013.92	14460.92	45.65	0.36
ENSEMBLE ADVERSARIAL TRAINING																
TRANSFERABILITY																
	Float-32	9.50%					24.40%					2.50%				
DeepFool	*Int-16*	50.10%	3055.41	3.11	0.08	0.01	66.10%	27525.41	6.98	0.06	0.00	6.70%	123907.36	132.95	0.85	0.05
	Int-8	77.40%					70.40%					38.10%				
	Float-32	0.00%					0.00%					0.00%				
C&W-L_∞	*Int-16*	1.00%	3072.00	12.72	0.25	0.01	1.00%	27647.95	100.67	0.65	0.01	0.00%	148553.92	810.09	2.57	0.01
	Int-8	4.00%					0.50%					7.00%				
	Float-32	5.30%					0.00%					6.20%				
AutoAttack	*Int-16*	6.20%	2461.41	9.34	0.17	0.00	1.50%	20739.19	73.75	0.46	0.00	6.20%	136836.20	1061.78	2.86	0.01
	Int-8	8.20%					2.30%					10.40%				
DIRECT ATTACK																
	Float-32	40.00%	1262.79	13.30	0.83	0.22	21.50%	12047.17	98.35	1.49	0.20	38.00%	62177.60	204.96	1.64	0.20
Square-L_2	*Int-16*	34.00%	1348.97	149.08	3.38	0.26	20.50%	11460.77	209.95	2.24	0.22	32.00%	48361.32	1490.06	5.11	0.21
	Int-8	40.00%	912.97	114.92	2.84	0.24	42.50%	3585.24	145.94	1.87	0.24	68.00%	15991.71	1136.66	4.49	0.21
	Float-32	6.50%	1827.15	27.29	0.56	0.01	2.00%	15074.52	224.56	1.54	0.01	53.50%	56926.61	852.58	2.63	0.01
Square-L_∞	*Int-16*	19.00%	1474.81	22.04	0.45	0.01	13.50%	10759.44	160.21	1.15	0.01	74.00%	18142.97	271.85	0.90	0.00
	Int-8	66.00%	835.95	13.09	0.28	0.01	36.50%	6892.77	103.09	0.77	0.01	83.50%	8740.61	136.87	0.50	0.00
	Float-32	8.50%	3070.34	8.02	0.18	0.01	-	-	-	-	-	1.00%	150485.45	2275.01	7.36	0.10
Boundary	*Int-16*	8.50%	3044.58	8.48	0.19	0.01	-	-	-	-	-	1.00%	150351.08	2552.88	8.26	0.10
	Int-8	83.50%	2413.35	53.70	1.22	0.08	-	-	-	-	-	1.50%	144081.12	8012.23	25.87	0.30
	Float-32	1.00%	2532.78	36.60	0.78	0.03	-	-	-	-	-	1.00%	143831.33	2087.90	6.72	0.06
GeoDA	*Int-16*	0.00%	2553.48	47.62	1.03	0.04	-	-	-	-	-	1.00%	139099.88	2055.59	6.40	0.06
	Int-8	3.00%	2522.78	280.73	5.87	0.21	-	-	-	-	-	10.00%	137627.38	13111.86	40.38	0.30
SINKHORN ADVERSARIAL TRAINING																
TRANSFERABILITY																
	Float-32	10.90%					16.70%					3.70%				
DeepFool	*Int-16*	44.30%	3056.98	15.84	0.45	0.06	24.90%	27542.34	261.91	2.88	0.22	14.40%	118859.27	204.69	1.84	0.14
	Int-8	45.80%					28.30%					25.60%				
	Float-32	3.50%					28.70%					14.90%				
C&W-L_∞	*Int-16*	7.50%	2964.48	32.83	0.65	0.02	34.10%	19712.98	210.62	1.44	0.01	15.40%	125949.44	1341.46	4.37	0.02
	Int-8	19.00%					37.30%					25.10%				
	Float-32	67.10%					76.80%					68.10%				
AutoAttack	*Int-16*	69.60%	419.50	1.65	0.03	0.00	78.30%	2151.66	8.49	0.05	0.00	68.30%	40545.09	313.89	0.85	0.00
	Int-8	72.30%					78.30%					75.50%				
DIRECT ATTACK																
	Float-32	61.60%	1078.53	9.63	0.79	0.23	69.60%	12112.00	79.32	1.71	0.29	45.20%	76364.58	190.76	1.90	0.28
Square-L_2	*Int-16*	48.00%	1216.14	240.27	5.15	0.32	71.20%	13841.80	1812.69	12.96	0.36	50.50%	61724.23	1735.81	6.03	0.26
	Int-8	47.50%	889.01	111.22	2.73	0.24	74.20%	8205.99	334.43	3.39	0.30	64.80%	17574.66	731.36	3.29	0.25
	Float-32	63.40%	1934.09	28.89	0.58	0.01	79.60%	18024.90	268.82	1.84	0.01	85.80%	98801.94	1479.51	4.31	0.01
Square-L_∞	*Int-16*	67.90%	1609.22	24.05	0.49	0.01	81.20%	17148.37	255.68	1.76	0.01	86.40%	70997.25	1062.85	3.25	0.01
	Int-8	73.60%	896.59	14.01	0.30	0.01	85.00%	10558.39	157.93	1.12	0.01	93.60%	23383.81	366.28	1.15	0.01
	Float-32	20.40%	2714.14	37.25	0.86	0.07	-	-	-	-	-	1.00%	150490.52	5685.44	18.31	0.22
Boundary	*Int-16*	18.60%	2716.08	41.60	0.95	0.07	-	-	-	-	-	2.00%	150370.31	6262.59	20.14	0.22
	Int-8	17.30%	2429.01	160.42	3.60	0.22	-	-	-	-	-	3.00%	144851.56	14062.23	44.71	0.42
	Float-32	4.00%	2534.17	138.65	2.84	0.10	-	-	-	-	-	93.60%	145633.62	2247.01	7.01	0.06
GeoDA	*Int-16*	7.00%	2502.89	251.41	5.28	0.19	-	-	-	-	-	6.40%	143881.06	9858.08	30.06	0.21
	Int-8	10.00%	2535.37	480.88	10.03	0.36	-	-	-	-	-	86.80%	135027.11	11173.99	34.60	0.27

Table 30: Evaluation of train-based adversarial defenses (all distortion metrics - Part 1).

7

Shifting Capsule Networks from the Cloud to the Deep Edge

This chapter was published as:

Miguel Costa, Diogo Costa, Tiago Gomes, and Sandro Pinto "*Shifting Capsule Networks from the Cloud to the Deep Edge*", In: *ACM Transactions on Intelligent Systems and Technology*. 2022. doi: 10.1145/3544562.

Abstract

CapsNets are an emerging trend in image processing. In contrast to a CNN, CapsNets are not vulnerable to object deformation, as the relative spatial information of the objects is preserved across the network. However, their complexity is mainly related to the capsule structure and the dynamic routing mechanism, which makes it almost unreasonable to deploy a CapsNet, in its original form, in a resource-constrained device powered by a small MCU. In an era where intelligence is rapidly shifting from the cloud to the edge, this high complexity imposes serious challenges to the adoption of CapsNets at the very edge. To tackle this issue, we present an API for the execution of quantized CapsNets in Arm Cortex-M and RISC-V MCUs. Our software kernels extend the Arm CMSIS-NN and RISC-V PULP-NN to support capsule operations with 8-bit integers as operands. Along with it, we propose a framework to perform post-training quantization of a CapsNet. Results show a reduction in memory footprint of almost 75%, with accuracy loss ranging from 0.07% to 0.18%. In terms of throughput, our Arm Cortex-M API enables the execution of primary capsule and capsule layers with medium-sized kernels in just 119.94 and 90.60 milliseconds, respectively (STM32H755ZIT6U, Cortex-M7 @ 480 MHz). For the GAP-8 SoC (RISC-V RV32IMCXpulp @ 170 MHz), the latency drops to 7.02 and 38.03 milliseconds, respectively.

7.1 Introduction

Convolutional layers have shown an impressive capacity to extract features - edges and shapes - from image pixels [53–55]. However, to reduce the dimension of feature maps, and consequently the number of parameters to learn, convolutional layers are typically connected through pooling layers in a CNN. Pooling layers take the most important features present in a region of the feature map generated by a convolution and pass those features to the following layer. However, this comes at the cost of decreased capacity to recognize pose and object deformation as the relative spatial information of the features is lost across the network [53–55]. Furthermore, CNNs have shown to be vulnerable to adversarial examples, i.e., inputs to a ML model that an attacker intentionally designs to compromise the integrity of the decision in a process similar to an optical illusion [166, 167]. To tackle these issues, Sabour et al. [53] proposed a new class of deep learning architectures, named CapsNets, to extract and classify features from images, while preserving and learning spatial information.

The core unit of a CapsNet is a capsule, a group of neurons organized as vectors. Each vector dimension represents a different parameter of the same object in the image [53] and the length of the vector represents the probability of the image containing the object. To date, there are four main variants of CapsNets: (i) transforming auto-encoders [124], (ii) routing by agreement between capsules [53], (iii) expectation-maximization routing capsules [125], and (iv) stacked capsule autoencoders [168]. However, the most prominent remains the routing by agreement between capsules [53], which is the target on this work. Dynamic routing is used to calculate a series of coefficients that determine the probability of an object in a lower capsule layer, being part of an object represented in a higher capsule layer [53].

In contrast to CNNs, CapsNets are equivariant, learning proportion and pose changing [55]. CapsNets already outperform CNNs in some healthcare [169] and automation mechanisms [54, 65, 170–173].

Despite being promoted as the next ground-breaking algorithm in deep learning [54], CapsNets in their original form are computationally intensive in terms of memory requirements (bandwidth and size) and latency [62–68]. For the same dataset, the original CapsNet has a ratio of MAC operations per memory near 100x higher than AlexNet [62], which is also a heavy CNN. In an era where concerns with cloud and internet infrastructure are disrupting the shift of ML to the deep edge [6, 7, 10, 18], the feasibility of CapsNets is still mainly confined to powerful cloud servers. To shift ML to the edge, researchers have been developing mechanisms for compressing ANNs and CNNs, discarding floating-point representation, and adopting a lighter fixed-point representation [18, 19]. If the potential of fixed-point computation in ANNs and CNNs is already known and ultimately led to the development of APIs tailored for the execution of QNNs in low-end MCUs [18, 19] (e.g., Arm Cortex-M and RISC-V), the feasibility of CapsNets under such constraints is still unknown.

To fill this gap, we evaluate the feasibility of CapsNets on resource-constrained platforms, powered by small MCUs targeted for intelligent IoT edge nodes. For this purpose, we extended the software kernels from CMSIS-NN [18] and PULP-NN [19] to support the deployment of CapsNets on Arm Cortex-M and RISC-V MCUs, respectively. We extended these APIs with functions to support capsule operations with int-8 data and evaluated their feasibility in terms of latency, accuracy penalty, and memory footprint reduction. As ML models are typically trained using floating-point precision, we developed a toolchain to perform post-training quantization of CapsNets developed in TensorFlow. Results show that this framework can reduce the memory footprint of a CapsNet by almost 75%, with a maximum accuracy loss of 0.18%. Our API was tested in three different Arm Cortex-M cores, i.e., STM32L4R5ZIT6U (Cortex-M4 @ 120 MHz), STM32H755ZIT6U (Cortex-M7 @ 480 Mhz), and STM32L552ZET6QU (Cortex-M33 @ 110 MHz). For the fastest MCU (STM32L4R5ZIT6U), our API enables the execution of primary capsule and capsule layers with medium-sized kernels in just 119.94 and 90.60 milliseconds, respectively. The API for the RISC-V RV32IMCXpulp architecture was tested on a Gapuino v1 board. For an octa-core setting at 170 MHz, the latency of medium-sized primary capsule and capsule layers dropped to 7.02 and 38.03 milliseconds, respectively. Our work is available open-source[1]. We believe our work will open a path in the application space of CapsNets, mainly in areas till now only explored by CNNs [3, 174] and encourage the development of lighter CapsNet architectures.

In summary, we make the following contributions: (i) the first public and open-source software kernels to enable the execution of CapsNets in Arm Cortex-M and RISC-V (RV32IMCXpulp) MCUs (Section 7.3); (ii) a public and open-source framework to quantize CapsNets from floating-point precision to int-8, compliant with the $Qm.n$ data format, used by Arm CMSIS-NN and RISC-V PULP-NN libraries (Section 7.4); (iii) the first public evaluation of the feasibility of CapsNets in resource-constrained MCUs featuring Arm Cortex-M and RISC-V (RV32IMCXpulp) cores (Section 7.5).

[1] `https://gitlab.com/ESRGv3/q7-\glspl{CapsNet}.git`

7.2 Background

7.2.1 Capsule Networks: Overview and Evolution

The first concept of capsules applied in deep learning was introduced in 2011 by Hinton et al. [124] to tackle the lack of equivariance affecting CNNs and to show how neural networks can be used to recognize pose. Capsules were described as sets of neurons, organized as vectors, and each capsule represented a given object of the input image. In contrast to a CNN, in which layers take as input/output scalars, capsules deal with vectors. They encapsulate posture information (position, direction, etc.) with other instantiation parameters (width, height, etc.) that together represent an object or object part. Each layer in a CapsNet is made up of a series of capsules and CapsNets follow a part-whole hierarchy. Capsules at one layer predict larger objects, represented by capsules at higher layers, via transformation matrices applied to their output vectors. For an object to be detected, capsules at a lower layer must agree in their spatial arrangement. This characteristic is highlighted in Figure 18. As observed, for the CapsNet in green, the pose of the mouth is not spatially aligned with the remaining parts of the face, resulting in a distorted human face to the human eye. As CapsNets learn the spatial relation of object parts, they can easily spot this distortion. In contrast, a traditional CNN would recognize the input image as a human face as CNNs learn the parts but not their spatial arrangement. Researchers have been proposing numerous CapsNet architectures since the proposal of the capsule concept in 2011. Here we review the most relevant architectures developed to date. For a more comprehensive overview of other architectures, we refer readers to works [54, 55, 175, 176].

TAEs are the first CapsNet architecture [124]. They were designed to encode the input image and regenerate it in the same original pose. TAEs differed from other auto-encoders as they were designed to explicitly learn the transformation matrix that represents the relation of one part of an image (e.g. eyes) to the position of other parts (e.g. nose). A TAE's capsule is composed of two distinct units: (i) recognition units and (ii) generative units. The recognition units were designed to compute the pose parameters and the probability that the object represented by the capsule is present in the input image. The generative units take this data as input to reconstruct the object. The major bottleneck of TAEs is that they require the transformation matrix to be supplied externally.

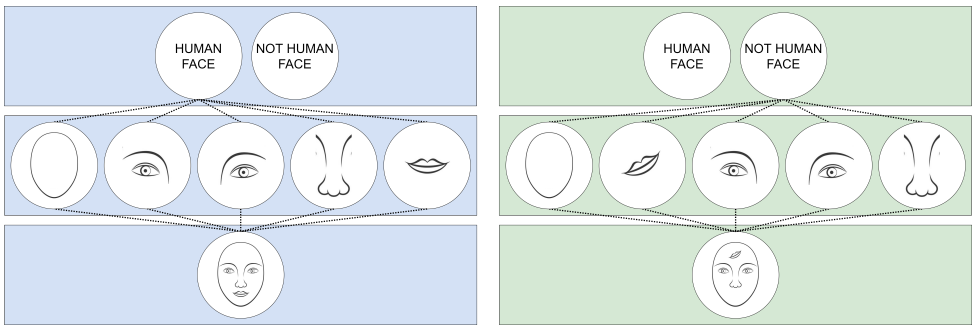


Figure 18: Basic principle of CapsNet.

The previous capsule structure was updated in 2017 for classification tasks. Sabour et al. [53] succeeded in implementing a CapsNet with state-of-the-art accuracy of 99.75% on MNIST and triggered the hype around CapsNets. This architecture features a convolutional layer, a primary capsule layer, and a class capsule layer. The convolutional layer extracts the most relevant features of the input image, tackling the need to externally provide the pose matrix of the objects. The primary capsule layer is the first capsule layer following the convolutional layer. Primary capsules have a convolutional kernel and encode the instantiation parameters of the lower-level objects of the input image. The last layer is the class capsule layer, which makes the final prediction. Capsules at different layers connect through a dynamic routing algorithm, which computes the probability of the object represented by a capsule in layer L being part of the objects represented by the capsules in layer $L + 1$. Sabour et al. [53] updated the capsule structure in this CapsNet architecture. Capsules are still a group of neurons that output a vector with instantiation parameters. However, Sabour et al. [53] gave meaning to the length of the vector - it represents the probability of the object being present in the input image. This CapsNet architecture is the most supported by the research community and is the focus of this work. Section 7.2.2 describes it in detail.

In 2018, Hinton et al. [125] proposed to replace the input and output vectors of capsules with matrices and updated the dynamic routing with an expectation-maximization algorithm. Instead of a pose vector, each capsule comprises a $4x4$ pose matrix and an activation parameter that represents the probability of the object is present in the input image. This tackles two of the biggest bottlenecks of the previous CapsNet architecture. Firstly, representing capsules with matrices of n elements instead of vectors reduces the number of parameters in transformation matrices between capsules from n^2 to n. Secondly, it avoids the use of a *squash* function to shrink the vectors to a maximum length of 1, which prevents the existence of any sensible objective function in the network. The new dynamic routing considers every capsule in a higher layer represented by a Gaussian distribution, defined by a mean and a standard deviation, and the capsules in a lower layer as the data points to be mapped to the Gaussian distributions. This algorithm encompasses two phases: (i) the assignment of data points to clusters (expectation) and (ii) the update of the centroids giving the assignments (maximization). This architecture achieved an accuracy of 98.2% on the smallNORB dataset.

The previous CapsNet architecture was updated in 2019 by Kosiorek et al. [168] for unsupervised learning applications. In addition to the pose matrix and activation parameter, this newer version of capsules (stacked capsule autoencoders) features a vector that represents other properties than pose, such as deformation, velocity, or color. While previous versions of capsules predict the pose of a parent capsule from the pose of the lower-layer capsules, this new architecture works in the opposite direction. A capsule that represents an object at one layer, predicts the pose of the parts represented in the lower layers. Consequently, there is no more need for iterative routing at inference time as every part is explained as a mixture of predictions from different objects. Stacked capsule autoencoders feature two main parts: (i) part capsule autoencoder and (ii) object capsule autoencoder. Stacked capsule autoencoders achieved a state-of-the-art accuracy for unsupervised classification on SVHN (55%) and MNIST (98.7%).

7.2.2 Capsule Networks with Dynamic Routing

Dynamic routing between capsules was the first CapsNet architecture proposed for classification tasks. Its successor, matrix capsules with expectation-maximization routing, promised to reduce the number of parameters to learn and to deliver better results in more complex datasets like smallNORB. Nevertheless, at the time of accomplishment of this work, the first version still got the most valuable support from the research community. Furthermore, it is the only version whose original implementation is publicly available [2]. Figure 19 depicts its general architecture.

A CapsNet based on dynamic routing between capsules starts with a convolutional layer, used to extract the most relevant features of the input image. In their model tuned for the MNIST dataset, Sabour et al. [53] employed 256 convolution kernels with dimensions 9x9 and a stride of 1, followed by a ReLU activation. The activated feature map is then sent to every capsule in the primary capsule layer. The primary capsule layer is the first capsule layer of the network and is where the inverse of the rendering process starts. Primary capsules have a convolutional kernel and are designed to refine the objects of interest identified by the previous convolutional layer and to determine the instantiation parameters, including spatial information. Each primary capsule represents a different object from the lowest level objects of the input image. The output of a primary capsule is a vector, where each dimension represents a given parameter of the object. The length of the output vector represents the probability of the object is present in the input image. For this purpose, the output of each capsule is squashed by the non-linear activation detailed in Equation 7.1. It preserves the original orientation of the output vector but normalizes its length to the range [0-1]. In their model tuned for the MNIST dataset, Sabour et al. [53] propose a primary capsule layer with 32 convolutional capsules, each one with 8-dimensional features. The convolution relies on a 9x9 kernel and a stride of 2.

$$v_j = \frac{||s_j||^2}{1 + ||s_j||^2} \frac{s_j}{||s_j||} \tag{7.1}$$

The primary capsule layer is followed by a single or multiple capsule layer. In the last capsule layer, each capsule represents a different object/class to be recognized by the network. For all but the primary capsules, the input is a weighted sum of the output vectors from the capsules in the layer below. For a given capsule j in layer $L + 1$, each output vector u_i from layer L is multiplied by the respective weight matrix W_{ij} as detailed in Equation 7.2. The resulting vector $\hat{u}_{ij}$ (prediction vector) is submitted to the

[2] https://github.com/Sarasra/models/tree/master/research/capsules

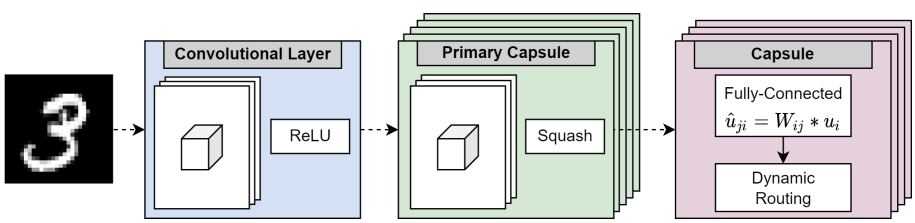


Figure 19: General architecture of a CapsNet with dynamic routing between capsules.

Algorithm 1: Dynamic routing in CapsNets.

1: for all capsule i in layer L and capsule j in layer $L+1$: $b_{ij} = 0$
2: **for** r iterations **do**
3: for all capsule i in layer L: $c_i = softmax(b_i)$
4: for all capsule j in layer $L+1$: $s_j = \sum_i c_{ij} * \hat{u}_{ji}$
5: for all capsule j in layer $L+1$: $v_j = squash(s_j)$
6: for all capsule i in layer L and capsule j in layer $L+1$: $b_{ij} = b_{ij} + \hat{u}_{ji} * v_j$
7: **end for**
8: **return** v_j

iterative dynamic routing mechanism. This mechanism computes the coupling coefficients c_{ij} between each capsule in layer L and capsules in layer $L+1$. Coupling coefficients determine the probability of the object represented by a capsule in layer L being part of a capsule in layer $L+1$.

$$s_j = \sum_i c_{ij} * \hat{u}_{ji}, \quad \hat{u}_{ji} = W_{ij} * u_i \tag{7.2}$$

Algorithm 1 details the dynamic routing algorithm [53]. In the first iteration of the routing process, the prediction vector of a capsule i in layer L is equally routed to every capsule j of layer $L+1$. This occurs as logits b_{ij} of the coupling coefficients are initialized as zeros. In the next iterations, each coupling coefficient c_{ij} is refined by measuring the agreement between the prediction vector of capsule i and the output of capsule j. The agreement is calculated as the scalar product between the prediction vector $\hat{u}_{ij}$ and the output vector v_j. This agreement is added to the initial logits b_{ij}. The output of a capsule at a given iteration is computed as the scalar product of the prediction vectors and the respective coupling coefficients. To ensure that the coupling coefficients between a capsule of layer L and all capsules from layer $L+1$ sum to 1, the logits b_{ij} that reflect the level of agreement between capsules are submitted to a *softmax* function before the output vector is calculated. As the output vector is calculated, it is submitted to a *squash* function to ensure that its length belongs to the interval [0, 1]. The routing algorithm is repeated for a given number of iterations, previously defined or hyper-tuned by the neural network designer. In their network specifically tuned for the MNIST dataset, Sabour et al. [53] uses a unique capsule layer, composed of 10 16-dimensional capsules.

7.2.3 Quantization

Quantization refers to techniques for mapping floating-point tensors to integer tensors, with lower bit-width [98, 99, 101, 102, 177]. As a consequence, a quantization process involves the encoding of (i) the sign, (ii) the integer part, and (iii) the fractional part of a float in a single integer value. In binary, quantized numbers are typically represented in $Qm.n$ notation, where m specifies the number of bits allocated for the integer part and n the number of bits allocated for the fractional part [102]. The main goal of quantization is to improve inference latency and reduce the memory footprint of neural networks. However, the success of a quantization procedure is highly dependent on the memory specification and ISA of the platform [102].

Post-training quantization vs. quantization-aware training. Quantization-aware training lowers the impact of the quantization on model accuracy by simulating the low-precision of weights during training. It considers the quantization error as part of the loss returned by the loss function, which the optimization algorithm tries to minimize [98, 99, 101, 102, 177]. In post-training quantization, the model is quantized without any fine-tuning during training.

Uniform vs. non-uniform quantization. Quantizing floating-point values to lower bit-width integers can return values that are equally (uniform quantization) or non-equally (non-uniform quantization) spaced [98, 102, 177]. For a fixed bit-width, non-uniform quantization may return more accurate ML models, as it captures with higher resolution the more important value regions [102, 177]. The quantization step is tuned to minimize the quantization error [98, 102, 177]. Nevertheless, non-uniform quantization typically introduces non-negligible overhead in the computation of quantized models on general-purpose hardware (GPUs and CPUs), sometimes leading to a latency higher than using the floating-point ML model [98, 102]. As a consequence, uniform quantization is the de-facto method to perform fast computations on resource-constrained devices [102, 177]. Equation 7.3 represents a typical uniform quantization function, where r is the float to be quantized, S is a scalar, and Z is a zero-point or offset.

$$Q(r) = round(r/S) - Z \tag{7.3}$$

Scaling and offset. In uniform quantization, the scaling factor and the zero-point are the core of the quantization function [99, 102, 177, 178]. The scaling factor specifies the step size of the quantizer, dividing a given range of values into equally spaced partitions. The zero-point is an integer offset used to ensure that a zero is quantized with no error when the range of values to be quantized is not centered around 0. To define a scaler (Equation 7.4), it is required to firstly define the target $Qm.n$ format and the range of floating-point values to be quantized ($[x_{min}, x_{max}]$) [99, 177]. The clipping range can be symmetric ($|x_{min}| = |x_{max}|$) or asymmetric ($|x_{min}| != |x_{max}|$). For an asymmetric setting, the zero-point is calculated as defined in Equation 7.5 [99]. Note that the zero-point is rounded to an integer value. Asymmetric quantization often results in a tight clipping range, which can reduce the impact of quantization on model accuracy [102, 177]. Nevertheless, symmetric quantization is widely adopted in practice as asymmetric quantization can introduce non-negligible overhead during inference [177].

$$S = \frac{x_{max} - x_{min}}{2^b - 1} \tag{7.4}$$

$$z = -round(x_{min} * S) - 2^b - 1 \tag{7.5}$$

Quantization granularity. Two metrics should be considered when selecting the granularity of quantization: (i) impact on model accuracy and (ii) computational cost [99]. The minimum impact on model

accuracy is achieved if the scaling factor and the zero-point are calculated for each weight and activation. However, this would result in a tremendous computation overhead and memory consumption as scaling and zero-point factors must be saved for each weight/activation. These factors combined may hamper the main objective of the quantization [102, 177]. In contrast, the minimum computational cost is achieved if the scaling factor and the zero-point are calculated for the whole neural network [102, 177]. However, this may result in an accuracy drop that can not be neglected. As a consequence, performing the quantization layer-by-layer or filter-by-filter are the most common strategies [98, 102].

Fixed bit-width vs. mixed bit-width. In a fixed bit-width setting, the same bit-width is used to quantize the whole network [98, 177]. In a mixed bit-width setting, the bit-width of each layer or filter is tuned for better model accuracy and/or memory footprint reduction [98, 177]. Activations and weights for layers/filters whose contribution to the final output is smaller get lower bit-width, while layers/filters that contribute the most get higher bit-widths. For the same memory footprint, this strategy has the potential to reduce the degradation of the model accuracy, however at the cost of a more complex quantization scheme and inference-pass software [98, 177].

Static vs. dynamic quantization. In static quantization, the scaler and the zero-point factors are calculated before inference, using a representative dataset. In dynamic quantization, these parameters are computed dynamically during the inference pass for each input [102, 177]. In this approach, weights and activations are stored in memory in low-precision bit-width, but the operations (matrix multiplication and convolution) are performed in floating-point arithmetic. In comparison to static quantization, dynamic quantization lowers the impact of quantization on model accuracy; however, it incurs a performance overhead that may be prohibited for low-end MCUs [102, 177].

7.2.4 CMSIS-NN

CMSIS-NN [18] is a series of open-source software kernels optimized to enable the deployment of neural networks on resource-constrained Arm Cortex-M MCUs. These software kernels are built upon the CMSIS abstraction layer and are optimized for Cortex-M MCUs that support SIMD. This API is organized in two parts: *NNFunctions* and *NNSupportFunctions*. *NNFunctions* include functions for the computation of the most popular layer types, such as convolution and fully-connected. *NNSupportFunctions* include support functions for data precision conversion and activations, which are used in the computation of *NNFunctions*. These kernels support both 8-bit and 16-bit data. The quantization is assumed to be uniform, symmetric, and to return values with fixed bit-width. The quantization must also be static and conducted layer-by-layer. As the quantization relies on a power-of-two scaling, the scaling of a layer output is implemented as a bitwise shift operation.

For a convolutional layer, input data is assumed to be in HWC format, i.e., channel first. In terms of activation functions, CMSIS-NN supports ReLU, *sigmoid*, and *tanh* [179]. ReLU is implemented as

a simple clipping function that maps negative values to 0 and maintains the value of positive values. *Sigmoid* and *tanh* are implemented using a LUT approach. These kernels were firstly tested by Lai et al. [18] on the STM32F746ZG MCU (Arm Cortex-M7) running a CNN featuring three convolutional layers and one fully-connected layer, at 216 MHz. CMSIS-NN was able to classify 10.1 images per second with an accuracy drop of only 0.04%.

7.2.5 PULP-NN

PULP-NN [19] is an open-source API optimized for the execution of QNNs in RISC-V RV32IMCXpulp MCUs. As this API is based on CMSIS-NN dataflow, it is also divided into two parts: *NNKernels* and *NNUtils*. *NNKernels* implement matrix multiplication and the standard layer types of neural networks, while *NNUtils* include support functions for data precision conversion and batch normalization. PULP-NN supports 8-bit and sub-byte (4-bit, 2-bit, and 1-bit) data types. However, in terms of activation functions its offering is more limited as it only supports ReLU. For the 8-bit setting, the quantization of weights is processed as for CMSIS-NN. Notwithstanding, the quantization of activations follows a different approach, as PULP-NN only supports non-negative activations. Consequently, the quantization must be asymmetric. Similar to CMSIS-NN, data is assumed to be in the HWC layout.

The key innovation of PULP-NN is the support for multi-core processing. For a convolution, the workload can be split along the height, width, or channel dimension of the output feature map. For a fully-connected layer, the workload is split along the output neurons. This library was firstly tested by Garofalo et al. [19] in a GAP-8 MCU executing a CNN quantized to int-8 data and featuring three convolutional layers and one fully-connected layer. In an octa-core configuration at 170 MHz, PULP-NN enables this CNN to run in 30x and 19.6x fewer clock cycles than the CMSIS-NN library, running on STM32L476 (Armv7-M, Cortex-M4) and STM32H743 (Armv7-M, Cortex-M7) MCUs, respectively.

7.3 Software Kernels

This section describes the software kernels that combined enable the deployment of CapsNets based on dynamic routing in Arm Cortex-M and RISC-V MCUs. As CapsNets architectures are very disruptive from generation to generation, these software kernels cannot be easily extended to the other CapsNets architectures detailed in Section 7.2.1. With the exception of the matrix multiplication kernels, all the other kernels are very specific to CapsNets with dynamic routing.

7.3.1 Matrix Multiplication

Matrix multiplication is a memory-intensive operation. Consequently, reducing memory access by maximizing data reuse at the register file level is a key factor in increasing the throughput of CapsNets. As Armv7E-M, Armv8-M, and RISC-V RV32IMCXpulp are 32-bit computing architectures and our software

kernels use 8-bit data as operands, matrix multiplication can strongly benefit from a SIMD to read four operands from memory at once. The maximum throughput would be achieved by combining this SIMD with a SIMD for the simultaneous MAC of two words containing four 8-bit operands (4x8-bit word). Since the ISAs of Armv7E-M and Armv8-M are very distinct from the ISA of RISC-V (RV32IMCXpulp), the matrix multiplication kernel was tuned for each computing architecture.

7.3.1.1 Arm Cortex-M

CMSIS-NN already provides a function for the multiplication of int-8 matrices: *arm_mat_mult_q7*. However, for all but the latest ARMv8.1-M architecture, this function does not rely on any type of hardware acceleration, including SIMD instructions, to read from memory, and for MAC operations. It simply iterates over the rows of the source matrix A and over the columns of the source matrix B, reading one element at a time and multiplying them. It uses a 32-bit accumulator, which saturates at 8-bit after the proper right shifting determined in the quantization process. *arm_mat_mult_q7* does not use any loop unrolling technique or parallelism. For a 4x4 sized matrix multiplication kernel, this function requires 8 load operations without sign extension and 4 MACs. As matrix multiplication is a very critical operation within the capsule computation, we tried to optimize the CMSIS-NN baseline function and developed two additional functions for quantized matrix multiplication: (i) *mat_mult_q7_trb* and (ii) *mat_mult_q7_simd*.

mat_mult_q7_trb differs from *arm_mat_mult_q7* as it transposes the source matrix B before entering in the MAC loop. As all matrices are stored in memory in height-width format, the parameters of the source matrix B can not be accessed contiguously. We believe this can reduce the complexity in the calculus of the memory address of the operands during a MAC operation. The general workflow of *mat_mult_q7_trb* is outlined in Figure 20.

mat_mult_q7_simd uses the SIMD feature of Armv7E-M and Armv8-M to read from memory and for MAC operations. For the signed MAC operation, the ISAs of these architectures do not feature instructions with 4x8-bit operands. However, they still feature instructions with 2x16-bit integers as operands. This means that we have to trade off the additional speedup delivered by SIMD instructions with the additional computation to perform sign extension. *mat_mult_q7_simd* also transposes the source matrix B prior to the MAC loop; however, it also sign-extends it to 16-bit. This function is also composed of (i) one outer loop

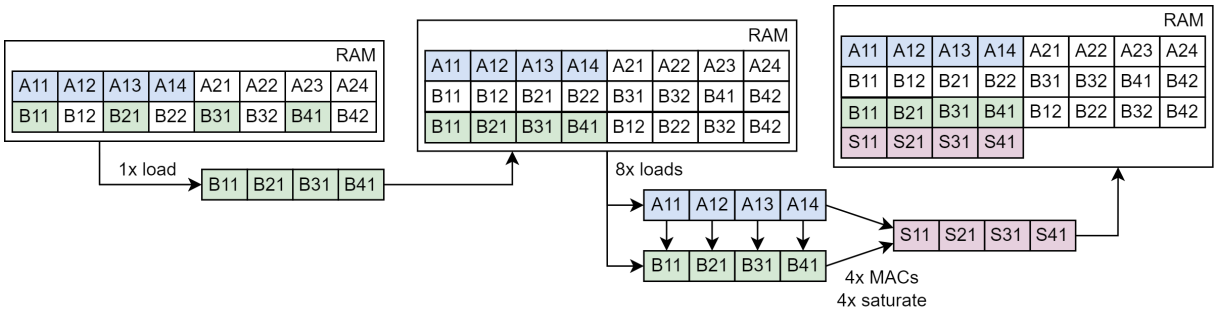


Figure 20: Matrix multiplication in Arm Cortex-M MCU using *mat_mult_q7_trb*.

Algorithm 2: *mat_mult_q7_simd(a, b, shift, r)* for Arm Cortex-M.

transpose and sign extend b to 16-bit: $b_trb = matrix_q7_to_q15_transposed(b)$
for row i in a **do**
  initialize accumulator: $sum = 0$
  read and expand int-8 word from a into two int-16 words: $read_and_pad(i, a1, a2)$
  for column j in $num_cols_b >> 2$ **do**
    read two int-16 from b_trb: $b1 = read_q15x2_ia(j)$
    read two int-16 from b_trb: $b2 = read_q15x2_ia(j)$
    MAC: $sum = __SMLAD(a1, b1, sum)$
    MAC: $sum = __SMLAD(a2, b2, sum)$
  end for
  for column j in $num_cols_b\%4$ **do**
    MAC: $sum = sum + (i * j)$
  end for
  fix quantization format: $sum = __SSAT(sum >> shift, 8)$
  copy accumulator to r: $^*r++ = sum$
end for

over the rows of the source matrix A and (ii) an inner loop over the columns of the transposed and sign-extended matrix B. However, as the SMLAD instruction takes 2x16-bit operands, we applied loop unrolling on the inner loop and compute 2 MACs simultaneously. For a 4x4 sized matrix multiplication kernel, it requires 4 load operations with sign extension and 2 MACs. Algorithm 2 details the $mat_mult_q7_simd$.

7.3.1.2 RISC-V

The ISA of RISC-V does not support SIMD instructions. However, in contrast to Arm Cortex-M, the RV32IMCXpulp extension provides SIMD instructions for MAC operations with 4x8-bit words as operands. RISC-V SoCs featuring the RV32IMCXpulp extension are usually provided with a fabric core and a multi-core acceleration cluster tailored for ANN computation [21]. The fabric core is an MCU built around the RISC-V core and extended for efficient digital signal processing. The acceleration cluster is turned on from the fabric controller when the programmer wants to offload computation-intensive kernels. Both the fabric controller and the acceleration cluster feature the RV32IMCXpulp extension. Nevertheless, PULP-NN does not feature any built-in function for the signed and quantized matrix multiplication in int-8 format. As a consequence, we developed three additional functions: (i) *mat_mult_q7*, (ii) *mat_mult_q7_trb*, and (iii) *mat_mult_q7_simd*. They are based on the Cortex-M implementation, but are tuned to extract the most of the acceleration cluster.

The *mat_mult_q7* does not rely on any SIMD instruction for reading and MAC operations, nor transposes the source matrix B prior to the MAC loop. It is the equivalent of the *arm_mat_mult_q7* but re-designed to support parallelization over the rows of the output matrix. The number of cores used in the parallelization must be a power of two (2^n). Similar to its Arm Cortex-M counterpart, *mat_mult_q7* requires 8 load operations without sign extension and 4 MACs for a 4x4 sized matrix multiplication kernel.

Algorithm 3: *mat_mult_q7_simd(a, b, shift, r)* for RISC-V.

1: transpose b: $b_trb = transpose(b)$
2: **for** row i in $a[start_row, stop_row]$ **do**
3: initialize accumulator: $sum = 0$
4: read int-8 word from a: $a1 = read(i)$
5: **for** column j in $num_cols_b >> 2$ **do**
6: read int-8 word from b_trb: $b1 = read(j)$
7: MAC: $sum = __builtin_pulp_sdotsp4(a1, b1, sum)$
8: **end for**
9: **for** column j in $num_cols_b\%4$ **do**
10: MAC: $sum = sum + (i * j)$
11: **end for**
12: fix quantization format: $sum = __builtin_pulp_clip_r(sum >> shift, 127)$
13: copy accumulator to r: $^*r++ = sum$
14: **end for**

mat_mult_q7_trb is very similar to the Arm implementation (Figure 20) - it was designed to evaluate the advantages of transposing the source matrix B prior to the MAC loop. Notwithstanding, there are two main differences. Firstly, it is tuned for multi-core processing. The transpose of the source matrix B is parallelized on the acceleration cluster over the row dimension and the same applies to the MAC loop. Secondly, the sign saturate is performed via a GCC built-in for the RISC-V RV32IMCXpulp (*__builtin_pulp_clip_r*). For a 4x4 matrix multiplication kernel, this function requires 8 load operations without sign extension and 4 MACs.

When considering the *mat_mult_q7_simd* function (Algorithm 3), the RISC-V RV32IMCXpulp version is more efficient as the ISA of this architecture features a SIMD instruction for the MAC of 4x8-bit operands. This function also transposes the source matrix B prior to the MAC loop. Both the matrix transposing and the MAC loop is performed in the same multi-core fashion as for *mat_mult_q7_trb*. When compared to Arm for a 4x4 matrix multiplication kernel, it requires 2 loads without sign extension and 1 MAC operation instead of 4 loads with sign extension and 2 MACs.

7.3.2 Squash Activation and Vector Norm

The *squash* activation function is designed to receive as input a 2D matrix, where each row corresponds to a vector to be squashed. For each row, the computation starts with the calculation of the vector length and ends with the squashing of each vector entry. The calculation of the vector norm uses a 32-bit integer to accumulate the sum of the power of two of each vector entry. As computing the exact value of a square root leads to severe overhead and latency in both architectures, we approximate the square root operation using the Newton-Raphson method [141].

The Newton-Raphson method is one of the most efficient techniques to approximate the zeroes of a mathematical function ($f(x) = 0$) [141]. It takes as input an initial approximation and then enters into an iterative process that produces successively better approximations. This iterative process is based on

Algorithm 4: Newton-Raphson method to calculate the square root of n.

1: take $n/2$ as the initial approximate root: $x_0 = n/2$
2: calculate the next approximate root according to Equation 7.7: $x_1 = \frac{(x_n+n)/x_n}{2}$
3: **while** $x_1 < x_0$ **do**
4: update the current approximate root: $x_0 = x_1$
5: update the next approximate root: $x_1 = \frac{(x_n+n)/x_n}{2}$
6: **end while**
7: **return** x_0

the evaluation of the derivative of the zero approximation. The square root calculus can be translated as finding the solution to Equation 7.6, which determines Equation 7.7 to find the next optimal solution at each iteration of the Newton-Raphson method [141]. The calculus ends whenever the absolute difference between the current zero-approximation and the zero-approximation projected for the next iteration is below 1 [141]. Algorithm 4 details the implementation of the Newton-Raphson method in our work.

$$f(x_n) = x_n^2 - n \tag{7.6}$$

$$x_{n+1} = \frac{(x_n + n)/x_n}{2} \tag{7.7}$$

As the length of a vector is calculated, each vector entry is squashed. In full-precision arithmetic, the output of *squash* is in the range $[0, 1]$. Therefore, the minimal precision loss is achieved if the output of *squash* is quantized in absolute Q0.7 format, i.e., all bits except for the sign bit represent the fractional part of a number. To comply with this quantization format, we tuned the original *squash* activation (Equation 7.1) as detailed in Equation 7.8, where i_{qn} and o_{qn} represent the number of virtual fractional bits in the input and output data, respectively. By extending the scaling granularity to the squared norm and norm parameters, we avoid the accuracy loss when the floating-point norm is below 0. For more details on the absolute and virtual quantization formats, we refer to Section 7.4. Equation 7.8 embeds the quantization of the *squash* activation output in the *squash* activation itself, avoiding the use of floating-point division and dynamic quantization, which introduces severe overhead. For the RISC-V implementation, the *squash* kernel can be offloaded to the acceleration cluster and parallelized along the vectors of the input matrix. The number of vectors is equally divided between each core. In case a fair division is not possible, the last core handles the remaining vectors.

$$v_j = \frac{||s_j|| << (o_{qn} - i_{qn})}{1 << i_{qn} + ||s_j||^2 >> i_{qn}} * s_j \tag{7.8}$$

7.3.3 Primary Capsule Layer

The primary capsule layer is the first capsule layer of a CapsNet. It has a convolutional kernel and can be observed as a convolutional layer with *squash* activation. Nevertheless, we have to consider that a primary capsule layer is 4-dimensional, while CMSIS-NN and PULP-NN only support convolution over 3D data. This

characteristic of a primary capsule layer can be observed in Figure 19 - it is characterized by the (i) height, (ii) width, and (iii) depth of the capsules, as well by the (iv) number of capsules. Instead of implementing a 3D convolutional kernel to perform convolution over 4D data, we borrowed the implementation strategy of Sabour et al. [53] and implemented primary capsule layers with 2D convolution kernels and then perform reshape. The third dimension of the data submitted to these 2D convolution kernels corresponds to the last two dimensions of the capsule layer *(capsule depth x number of capsules)*. The output of the convolution is reshaped as a 2D vector where the first dimension corresponds to *capsule height x capsule width x number of capsules* and the last dimension correspond to *capsule depth*. These 2D vectors get squashed along the last dimension, ensuring that the magnitude of the capsule depth sums to 1. The output feature map of the squash activation is then reshaped to match the original 4D shape of the primary capsule layer. As detailed in Section 7.4, quantized primary capsule layers must scale the output feature map of the convolution prior to the squash activation to minimize the accuracy loss. For this purpose, for both Arm Cortex-M and RISC-V, our software kernel requires the programmer to pass two scaling factors: (i) one for the bias and (ii) another for the outputs.

7.3.3.1 Arm Cortex-M

CMSIS-NN features two functions for int-8 convolutions: *arm_convolve_HWC_q7_basic_nonsquare* and *arm_convolve_HWC_q7_fast_nonsquare*. The former is a basic function, while the later is optimized for reduced latency. Despite the reduced latency, the later version imposes constraints on the dimension of the input and output channels, which must be multiples of 4 and 2, respectively. Considering that these restrictions are too severe and may not fit all the design scenarios, we decided to provide the API with two functions for the primary capsules: one based on the basic convolution (*pcap_q7_basic*) and another based on the fast convolution (*pcap_q7_fast*). These functions are already designed to leverage most of the SIMD features in Armv7E-M and Armv8-M architectures. Nevertheless, as the remaining CMSIS-NN functions, they are not ready to parallelize the convolution over multi-cores. CMSIS-NN already embeds in their convolution kernels the scaling of the output feature map. The scaling is performed as a bitwise-shift operation for the bias and the outputs. The *squash* activation is detailed in Section 7.3.2. For more details on the convolution kernels implemented by CMSIS-NN, we refer to the work [18].

7.3.3.2 RISC-V

PULP-NN does not provide any function that can be directly used to implement the convolution part of a primary capsule. Convolution functions provided by PULP-NN involve the clipping of negative activations to zero, which is incompatible with the fundamentals of CapsNets. Clipping negative values limit the direction scope and distort the magnitude of the output vector, introducing an additional non-linearity that CapsNets are not designed to support. Nevertheless, RISC-V (RV32IMCXpulp) already features SIMD and GCC builtin-functions for the dot-product of two vectors with 4x8-bit signed data. Furthermore, it

also provides a builtin-function for the saturate operation. Given the background, we borrowed the PULP-NN functions implementing convolutional layers (*pulp_nn_conv_Co_parallel, pulp_nn_conv_Ho_parallel, pulp_nn_conv_HoWo_parallel*) and adapted them for signed int-8 data. We updated the data type of all arguments and intermediate variables in unsigned integer to signed integer and replaced the GCC built-ins for the dot-product and saturate operations, which are now performed by *__builtin_pulp_sdotsp4* and *__builtin_pulp_clip_r*, respectively. The batch normalization and ReLU activation were also updated. The new convolution functions - (i) *pulp_nn_conv_Co_parallel_q7*, (ii) *pulp_nn_conv_Ho_parallel_q7*, (iii) and *pulp_nn_conv_HoWo_parallel_q7* - maintain the same inner loop of the convolution used by PULP-NN. They are also optimized to run on the acceleration cluster. Consequently, our software kernels feature three different functions for the primary capsule layer, each one devoted to a different parallelization strategy - parallelization can be performed along the channel (*pcap_Co_q7*), height (*pcap_Ho_q7*), or height vs. width (*pcap_HoWo_q7*). Pixels are equally divided by all cores. If a fair division is not possible, the last core handles the remaining. Nevertheless, primary capsule kernels can also run in the fabric controller. For more details on the inner loop of the convolution, we refer to the work [19]. The *squash* activation is detailed in Section 7.3.2.

7.3.4 Capsule Layer

Algorithm 5 details the workflow of the kernel implementing a capsule layer (*capsule_layer_q7*). This workflow is mainly implemented by a series of four support functions, starting with the *calc_inputs_hat* which calculates the prediction vectors, which is then submitted to the dynamic routing. The dynamic routing starts with the *calc_coupling_coefs* which computes the coupling coefficients by applying a softmax function to the logits representing the agreement between capsules in consecutive layers. These coupling coefficients and the prediction vectors are further used as inputs to *calc_caps_output*, which returns the output vectors of the capsule for the current routing iteration. If the dynamic routing is not in the last iteration, the function *calc_agreement_w_prev_caps* calculates the agreement between capsules in layer $L + 1$ and capsules in layer L. Each one of these support functions is detailed below.

Algorithm 5: *capsule_layer_q7(args)* for Arm Cortex-M and RISC-V.

1: initialize logits b_{ij} as zeroes: $memset(b, 0, num_caps * input_num_caps)$
2: calculate prediction vectors: $calc_inputs_hat(b, layer_input, shift, pred_vectors)$
3: **for** routing r in $num_routings$ **do**
4: calculate coupling coefficients c_ij: $calc_coupling_coefs(b, c)$
5: calculate output vectors: $calc_caps_output(pred_vectors, c, shift, out_vectors)$
6: **if** $r < num_routings - 1$ **then**
7: calculate agreement between capsules:
$calc_agreement_w_prev_caps(pred_vectors, out_vectors, shift, b)$
8: **end if**
9: **end for**

calc_inputs_hat. Multiplies the input of the capsule layer by the respective weight matrix. These weights connect each feature of each capsule in layer L to each feature of each capsule in layer $L+1$. As a consequence, the original weight matrix is 4D *[total capsules in layer $L+1$, total capsules in layer L, capsule size in layer $L+1$, capsule size in layer L]* and not directly supported by our matrix multiplication kernels. Nevertheless, performing a direct 4D matrix multiplication would require the replication of the input feature map for each capsule, increasing the memory footprint. Instead, *calc_inputs_hat* regards the 2 inner dimensions of the weight matrix as valid matrix multiplication dimensions and the outer dimensions as batch size. *calc_inputs_hat* has an outer loop that iterates over all capsules of layer $L+1$ and an inner loop that iterates over all capsules of layer L. In the inner loop, the weights connecting a capsule i in layer L to a capsule j in layer $L+1$ are multiplied by the input features of capsule j (output features of capsule i). The multiplication of these two matrices uses the fastest of the kernels described in Section 7.3.1 - *mat_mult_q7_trb* for Arm and *mat_mult_q7_simd* for RISC-V. The resulting prediction vector is then submitted to the iterative dynamic routing where the output vector will be iteratively updated.

calc_coupling_coefs. First function of the dynamic routing and computes the coupling coefficients c_{ij}. More specifically, *calc_coupling_coefs* applies a softmax function over the logits b_{ij} that measure the agreement between capsules at different layers. Each logit b_{ij} measures the agreement between a capsule i in the previous layer (layer L) and a capsule j of the current layer (layer $L+1$). Applying a softmax function to these logits ensures that the importance of a capsule i in layer L for all capsules of the current layer $L+1$ sums to 1. For the Armv7E-M and Armv8-M, we make direct use of the *arm_softmax_q7* function provided by CMSIS-NN. In contrast, PULP-NN does not provide any function implementing softmax. We developed a *softmax* function based on the Arm implementation.

calc_caps_output. Multiplies the coupling coefficients with the prediction vectors, returning the output vectors. If coupling coefficients are already expressed in a 2D matrix, the same is not applied to the prediction vector, which is represented by a 3D matrix *[total capsules in layer $L+1$, total capsules in layer L, capsule size in layer $L+1$]*. Consequently, *calc_caps_output* regards the 2 inner dimensions as the valid matrix multiplication dimensions and the outer dimension as batch size. Matrix multiplication is performed using the fastest of the kernels described in Section 7.3.1. When the matrix multiplication loop finishes, the output vectors are squashed using the kernel described in Section 7.3.2.

calc_agreement_w_prev_caps. Determines the degree of agreement between the weighted output of a capsule in layer L (prediction vectors) with the output of capsules (output vectors) in layer $L+1$. The agreement is determined as the product of the respective matrices. As the prediction vector is represented by a 3D matrix *[total capsules in layer $L+1$, total capsules in layer L, capsule size in layer $L+1$]*, *calc_caps_output* consider the inner two dimensions as the valid matrix multiplication dimensions and the outer dimension as batch size. The result is then summed up to the logits b_{ij}. For this purpose, we rely on 2D matrix addition kernels.

7.4 CapsNet Quantization

Algorithm 6 details the internals of the quantization framework. It takes as input a CapsNet developed in TensorFlow and a quantization dataset and returns the quantized weights and bias, as well as the output and bias shifts. Except for primary capsule and capsule layers, the framework considers activation functions as independent layers. Embedding an activation in dense or convolutional layers may return faulty quantized models.

As detailed in Algorithm 6, after loading the model and the dataset, the quantization process starts with the quantization of the model's weights. The quantization itself is performed as detailed in Algorithm 7. For each layer, the framework searches for the maximum absolute weight (max_abs) and calculates the number m of integer bits required to represent it. As we are using a power-of-two scaling ($Ax2^n$), the number of integer bits to represent a value A is calculated as the ceiling of $log_2(max_{abs})$. The remaining n bits are considered for the fractional part. Nevertheless, to increase precision in layers with very small weights (maximum absolute weight lower than $1/127$), our framework virtually increases the number of fractional bits, bringing the maximum norm of the quantized weights closer to the saturation barrier without never going over it. In absolute or physical terms, every weight is still in $Q0.7$ format as it still fits in eight bits, but virtually the quantization format can surpass the eight-bit barrier. It must be noted that the first bit of the eight available is used for the sign. This $Qm.n$ quantization format allows the representation of values comprised in the uniform and symmetric range $[-max_abs, max_abs]$. After getting the quantization $Qm.n$ format, the framework proceeds to the calculus of the quantized weights, where the floating-point number A is multiplied by 2 powered to the number of fractional bits n ($Ax2^n$). The quantization finishes with the clipping of values to the range $[-128, 127]$. The quantization of the model's bias is performed under the same workflow.

This quantization strategy enables the addition and multiplication of quantized values with minimal overhead. However, the results must be scaled to avoid saturation problems. As we are using a power-of-two-scaling, scaling can be performed as a bitwise shift operation. In the context of CapsNets, scaling

Algorithm 6: Quantization framework for CapsNets.

load CapsNet from .h5 file
load quantization reference dataset
quantize weights: $wq_{ij} = round(w_{ij}/S)$
quantize bias: $bq_{ij} = round(b_{ij}/S)$
calculate amount of output/bias shift:
  for layer l in CapsNet **do**
    for matrix multiplication, matrix addition or convolution k **do**
      get Qm.n format of the input, output, and intermediates
      calculate amount of output shift: $out_s = f_{ia} + f_{ib} - f_o$
      calculate amount of bias shift: $bias_s = f_{ia} + f_{ib} - f_b$
    end for
  end for
return wq, bq, s

Algorithm 7: Quantization algorithm for CapsNets.

1: get Qm.n format:
2: find maximum absolute value max_{abs}
3: find amount m of bits to represent the range $[-max_{abs}, max_{abs}]$: the ceiling of $log_2(max_{abs})$
4: calculate amount n of bits for the fractional part:
5: $n = 7 - m$
6: **while** $(max_{abs} >> (n+1)) <= 127$ **do**
7: $n++$
8: **end while**
9: convert to quantized integer:
10: multiply floating-point values A by 2 powered to the number of fractional bits: $Ax2^n$
11: clip quantized integers for the range [-128, 127]

takes part of the convolutional, primary capsule, and capsule layers as these layers apply matrix multiplication and/or addition. Convolutional and primary capsule layers only go through a single matrix multiplication process, and therefore only take one scaling factor for the bias and another for the output. However, capsule layers involve at least three distinctive matrix multiplications and one matrix addition. When considering the four support functions of a capsule layer kernel, only *calc_coupling_coefs* does not require any scaling factor. *calc_inputs_hat* requires one output scaling factor and *calc_caps_output* requires one for each iteration of the dynamic routing. As *calc_agreement_w_prev_caps* comprises matrix multiplication and addition, it requires two output scaling factors per iteration of the dynamic routing unless for the last one.

Scaling factors are calculated as detailed in Algorithm 6. For each matrix multiplication or addition in each CapsNet layer, the reference quantization dataset is used to determine the maximum absolute value for the input and output data. Then, the framework applies Algorithm 7 to get the quantization format ($Qm.n$) for this data. At this step, the framework is ready to calculate the number of right shifts for every matrix multiplication/addition. Considering f_{ia} as the number of fractional bits for the input matrix A, f_{ib} as the number of fractional bits for the input matrix B, and f_o as the number of fractional bits for the output, the amount of right shifts (output scaling factor) is calculated as detailed in line 9 of Algorithm 6. The same process is carried out to calculate the left bias shift of convolutional and primary capsule layers. The bias shift is calculated as detailed in line 10 of Algorithm 6. f_b represents the number of fractional bits for the bias.

7.5 Evaluation

We evaluated the software kernels and the quantization framework on three imaging datasets: (i) MNIST [180], (ii) smallNORB, and [181] (iii) CIFAR-10 [182]. MNIST is a large database of handwritten digits between 0 and 9. It contains 60k training images and 10k testing images, both shaped as $28x28x1$. smallNORB is a dataset designed to develop applications for 3D object recognition from shape. It contains 5 generic categories of toys - (i) four-legged animals, (ii) human figures, (iii) airplanes, (iv) trucks, and (v)

Dataset	Convolutional Layers		Primary Capsule Layers		Capsule Layers		Optimizer	
	Layer #	Configuration	Layer #	Configuration	Layer #	Configuration	Type	LR
MNIST	1	Filters: 16 Kernel Size: 7 Stride: 1 Activation: ReLU	2	Capsules: 16 Dimension: 4 Kernel Size: 7 Stride: 2	3	Capsules: 10 Dimension: 6 Routings: 3	Adam	0.001
smallNORB	1	Filters: 32 Kernel Size: 7 Stride: 1 Activation: ReLU	2	Capsules: 16 Dimension: 4 Kernel Size: 7 Stride: 2	3	Capsules: 5 Dimension: 6 Routings: 3	Adam	0.00025
CIFAR-10	1-4	Filters: [32, 32, 64, 64] Kernel Size: 4 x [3] Stride: [1, 1, 2, 2] Activation: 4 x [ReLU]	5	Capsules: 16 Dimension: 4 Kernel Size: 3 Stride: 2	6	Capsules: 10 Dimension: 5 Routings: 3	Adam	0.00025

Table 31: Baseline reference CapsNets for evaluation.

cars - imaged under varying light conditions, elevations, and azimuths. Both training and testing batches are composed of 24.3k images, shaped as $96x96x2$. CIFAR-10 consists of 60k $32x32x3$ colour images, representing 10 different objects. There are 50k images for training and 10k for testing.

For each dataset, we trained a full-precision and baseline reference CapsNet. We do not intend to build CapsNets with state-of-the-art accuracy on these datasets but rather build a CapsNet that after quantization fits in the limited memory resources of the target low-power MCUs without compromising the accuracy. We have built CapsNets that after quantization take no more than 80% of the RAM available in the least performing target MCU. At the moment of this evaluation, our software kernels do not support tiling. Thus, we have to ensure that both the CapsNet parameters and at least one sampling image can fit in the available RAM without compromising the execution of our software kernels. Table 31 details the architecture and hyper-parameters of the CapsNets used for testing.

The evaluation included three Arm Cortex-M MCUs: STM32L4R5ZIT6U (Cortex-M4 @ 120 MHz, 640 KB RAM), STM32H755ZIT6U (Cortex-M7 @ 480 MHz, 1 MB RAM), and STM32L552ZET6QU (Cortex-M33 @ 110 MHz, 512 KB RAM). Hence, our evaluation encompasses CPUs featuring Armv7-M architecture (Cortex-M4 and Cortex-M7), and the more recent Armv8-M (Cortex-M33). At the time of this evaluation, the market's scarcity of in-silicon MCUs featuring the RISC-V RV32IMCXpulp ISA severely limited the testing setup. For RISC-V, our software kernels were tested on a single development board featuring a GAP-8 MCU (GreenWaves GAPuino v1), with a fabric controller running at 250 MHz and an acceleration cluster with 8 cores running at 170 MHz. All cores share 512 KB of RAM. Although there are several open-source RISC-V cores using RISC-V RV32IMC ISA, using soft-cores deployed in an FPGA to test our software kernels would not enable a fair comparison with the Arm Cortex-M MCUs.

7.5.1 Quantization

The quantization framework was evaluated in terms of memory footprint reduction and accuracy loss. As detailed in Section 7.4, the quantization mechanism returns a series of output and bias shifts that must be performed during inference time. We consider these parameters as part of the memory footprint inherent to the quantized CapsNet. Table 32 details the accuracy and memory footprint of the full precision CapsNets against their quantized counterparts. While a full-precision CapsNet requires 4 bytes per parameter, the quantized version only requires 1 byte. This represents a theoretical memory saving of 75%, which is very near to the memory saving achieved by our quantization framework (74.99%). This demonstrates that the size of the additional parameters required by quantized CapsNets for bias and output shifting is almost negligible. This applies to conventional 3-layered CapsNets but also deeper CapsNets as the one developed for the CIFAR-10 dataset.

Dataset	**Memory Footprint (KB)**			**Accuracy**		
	Float-32	**Int-8**	**Int-8 Saving**	**Float-32**	**Int-8**	**Int-8 Loss**
MNIST	1187.20	296.82	74.99%	99.01%	98.83%	0.18%
smallNORB	1182.34	295.61	74.99%	92.56%	92.49%	0.07%
CIFAR-10	461.19	115.33	74.99%	78.54%	78.38%	0.16%

Table 32: Evaluation of the quantization framework for CapsNets.

In terms of accuracy, the maximum loss was achieved for the MNIST dataset (0.18%) and the minimum for the smallNORB (0.07%). This shows that the accuracy loss resulting from the quantization process does not necessarily grow with the complexity of the dataset. However, it is visible that CapsNets do not perform so-well on complex datasets like CIFAR-10. CapsNets like to account for everything in the image so it does better when there is no background noise or the background does not constantly change. As this is not the case with CIFAR-10, CapsNets will perform poorly when compared to other less noisy datasets.

7.5.2 Software Kernels

7.5.2.1 Matrix Multiplication

To select the fastest matrix multiplication algorithm for each target architecture, we considered the multiplication of two randomized matrices with dimensions 20x30 and 30x40. The content of each matrix was kept constant during the entire test and the latency was measured in milliseconds and clock cycles. As detailed in Section 7.3.1.1, for the Arm Cortex-M architecture we tested three different algorithms: (i) *mat_mult_q7_simd*, (ii) *mat_mult_q7_trb*, and (iii) *arm_mat_mult_q7*. The last one is used as a baseline reference as it does not apply any SIMD or matrix transposing operation to simplify memory access during MAC. Results are detailed in Table 33.

Contrary to what was expected, for the Arm Cortex-M implementation, the maximum performance is not observed for the algorithm relying on SIMD instructions. This occurs because the Arm ISA only

	STM32L4R5ZIT6U *(Armv7E-M, Cortex-M4)*		**STM32H755ZIT6U** *(Armv7E-M, Cortex-M7)*		**STM32L552ZET6QU** *(Armv8-M, Cortex-M33)*	
	Clock Cycles	**Milliseconds**	**Clock Cycles**	**Milliseconds**	**Clock Cycles**	**Milliseconds**
arm_mat_mult_q7	704395	5.87	790989	1.65	654738	5.96
mat_mult_q7_trb	655415	5.47	574532	1.20	605769	5.51
mat_mult_q7_simd	730562	6.09	757482	1.58	697749	6.35

Table 33: Evaluation of matrix multiplication functions for Arm Cortex-M MCUs.

supports MAC instructions with 16-bit operands, requiring the signal extension of each 8-bit operand. Despite providing SIMD instructions for the sign extension operation, it still introduces too much performance overhead. *mat_mult_q7_simd* is even slower than the *arm_mat_mult_q7*, which does not feature any optimization. The *mat_mult_q7_trb* and *arm_mat_mult_q7* functions are, on average, 1.15x and 1.04x faster than *mat_mult_q7_simd*, respectively. The maximum performance is registered for the algorithm that transposes the second input matrix before the matrix multiplication process. By simplifying the calculus of memory addresses during MAC, *mat_mult_q7_trb* is on average 1.15x faster than the approach based on SIMD instructions, and 1.10x faster than the Arm CMSIS implementation.

Regarding the RISC-V implementation, the results are detailed in Table 34. As GAP-8 SoC features an acceleration cluster tailored for neural network computation, we tested our matrix multiplication functions leveraging the cluster. Tests were performed in single-core and octa-core settings. Opposed to Arm Cortex-M, the fastest algorithm here is based on SIMD instructions - *mat_mult_q7_simd*. This happens because the RISC-V RV32IMCXpulp features SIMD instructions for the simultaneous MAC of 8-bit operands, in contrast to Arm Cortex-M that only supports 16-bit operands and requires data expansion. For the octa-core setting, *mat_mult_q7_simd* is $2.10x$ faster than *mat_mult_q7_trb* and $2.05x$ faster than *mat_mult_q7*. In a single-core configuration, the speedup of *mat_mult_q7_simd* in relation to the other implementations is slightly higher: $2.20x$ faster than *mat_mult_q7_trb* and $2.15x$ faster than *mat_mult_q7*. When comparing the performance of the single-core with the octa-core implementation, we notice that the octa-core is 6.32x to 6.63x faster than the single-core. The combination of the RISC-V RV32IMCXpulp SIMD instructions with multi-core processing can outperform the STM32H755ZIT6U even though this Arm MCU runs at a much higher clock frequency than the RISC-V MCU (480 MHz vs. 170 MHz).

	GAP-8 (single-core) *RISC-V RV32IMCXpulp*		**GAP-8 (octa-core)** *RISC-V RV32IMCXpulp*	
	Clock Cycles	**Milliseconds**	**Clock Cycles**	**Milliseconds**
mat_mult_q7	696951	4.10	105250	0.62
mat_mult_q7_trb	715602	4.21	107784	0.64
mat_mult_q7_simd	323844	1.91	51238	0.31

Table 34: Evaluation of matrix multiplication functions for RISC-V MCUs.

7.5.2.2 Primary Capsule Layer

We tested the primary capsule kernels in terms of latency on the three previously described CapsNets. For the Armv7E-M and Armv8-M architectures we developed two different functions to implement a primary capsule layer: (i) *pcap_q7_basic* and (ii) *pcap_q7_fast*. The former is a slower version without constraints on the input and output channel sizes. The latter is a faster version but only supports multiples of 4 and 2 for the input and output channel - a constraint that is met by the three CapsNets used for evaluation. Table 35 details the latency for the three Arm Cortex-M MCUs in the classification of a single image.

As expected, *pcap_q7_fast* outperforms the baseline reference for all datasets - it is at least 1.08x faster. Nevertheless, the most noticeable impact on the latency is caused by the size of the kernel. The biggest kernel, inherent to the smallNORB CapsNet, is 2.73x bigger than the smallest of the kernels, designed for the CIFAR-10 dataset. However, the computation of the smallNORB kernel is 33.32x to 34.23x slower than the computation of the CIFAR-10 kernel. This means that the computation time does not grow linear with the kernel size of the primary capsule layer, but rather exponentially. Given the background, big kernels must be avoided whenever possible.

For the RISC-V RV32IMCXpulp architecture, we developed three different functions implementing a primary capsule layer, each parallelizing a different spatial dimension of the output feature map over the cores of the acceleration cluster (Section 7.3.3). Table 36 details the latency of the three implemented functions on the three reference CapsNets. As expected, it is not possible to mention a function that is the fastest for every dataset or CapsNet architecture. As the latency is directly impacted by the spatial dimension that is parallelized, it all depends on the shape of the input feature map. *pcap_ho_q7* is simultaneously the fastest function for the MNIST dataset and the slowest for the other two. Similar to Arm, the latency grows exponentially with the kernel size of the primary capsule layer. While the kernel of the smallNORB is only 2.73x bigger than the kernel of the CIFAR-10, the computation of the smallNORB kernel is at least 33.39x slower (single-core). This effect is not attenuated in a multi-core configuration.

When compared to Arm, the RISC-V implementation completely outperform by almost two orders of magnitude. RISC-V RV32IMCXpulp ISA provides SIMD instructions for the simultaneous MAC of two 4x8-bit operands, being considerably faster in matrix multiplication. For a single-core setting on the smallNORB dataset, STM32L552ZET6QU is 5.01x slower than the RISC-V MCU (*pcap_q7_fast vs. pcap_co_q7*).

Dataset		**STM32L4R5ZIT6U** *(Armv7E-M, Cortex-M4)*		**STM32H755ZIT6U** *(Armv7E-M, Cortex-M7)*		**STM32L552ZET6QU** *(Armv8-M, Cortex-M33)*	
		Clock Cycles	**Milliseconds**	**Clock Cycles**	**Milliseconds**	**Clock Cycles**	**Milliseconds**
MNIST	*pcap_q7_basic*	65.79M	548.25	63.49M	132.29	51.34M	466.77
7x7x16x64 (M)	*pcap_q7_fast*	60.12M	500.97	57.57M	119.94	46.65M	424.13
smallNORB	*pcap_q7_basic*	406.35M	3386.29	389.62M	811.70	316.95M	2881.32
7x7x32x64 (L)	*pcap_q7_fast*	372.55M	3104.57	355.22M	740.03	289.06M	2627.78
CIFAR-10	*pcap_q7_basic*	12.09M	100.75	11.40M	23.75	9.26M	84.17
3x3x64x64 (S)	*pcap_q7_fast*	11.18M	93.19	10.50M	21.87	8.50M	77.30

Table 35: Evaluation of primary capsule functions for Arm Cortex-M MCUs.

Dataset		GAP-8 (single-core) *RISC-V RV32IMCXpulp*		GAP-8 (octa-core) *RISC-V RV32IMCXpulp*	
		Clock Cycles	**Milliseconds**	**Clock Cycles**	**Milliseconds**
MNIST	*pcap_co_q7*	9.45M	55.59	1.58M	9.27
7x7x16x64 (M)	*pcap_ho_q7*	9.40M	55.27	1.19M	7.02
	pcap_howo_q7	9.49M	55.85	1.18M	6.95
smallNORB	*pcap_co_q7*	57.69M	339.35	9.40M	55.32
7x7x32x64 (L)	*pcap_ho_q7*	58.27M	342.76	11.48M	67.53
	pcap_howo_q7	57.70M	339.39	11.40M	67.07
CIFAR-10	*pcap_co_q7*	1.73M	10.15	0.27M	1.59
3x3x64x64 (S)	*pcap_ho_q7*	1.74M	10.26	0.43M	2.55
	pcap_howo_q7	1.72M	10.15	0.22M	1.30

Table 36: Evaluation of primary capsule functions for RISC-V MCUs.

7.5.2.3 Capsule Layer

We tested the function implementing a capsule layer on the three reference datasets. Tables 37 and 38 detail the results for the Arm Cortex-M and RISC-V MCUs, respectively. Considering only the single-core scenario, we observe that the RISC-V implementation outperforms Arm. This arises as a direct consequence of the more efficient matrix multiplication kernel deployed in RISC-V (Section 7.5.2.1). As detailed in Section 7.3.4, three of the four support functions composing the capsule layer rely on matrix multiplication. As matrix multiplication is more efficient in RISC-V MCUs, due to the use of appropriate SIMD, the capsule layer function will deliver results with lower latency. When comparing the fastest of the Arm Cortex-M MCUs in terms of clock cycles (STM32L552ZET6QU) against the RISC-V MCU in single-core mode, the latter is on average 3.95x faster. Regarding the octa-core setting of the RISC-V MCU, we can observe that this setting is on average 7.43x faster than the single-core setting. This means that the *cap_parallel_q7* function is very near to the theoretical perfect parallelization over all the cores of the acceleration cluster.

Dataset		STM32L4R5ZIT6U *(Armv7E-M, Cortex-M4)*		STM32H755ZIT6U *(Armv7E-M, Cortex-M7)*		STM32L552ZET6QU *(Armv8-M, Cortex-M33)*	
		Clock Cycles	**Milliseconds**	**Clock Cycles**	**Milliseconds**	**Clock Cycles**	**Milliseconds**
MNIST *10x1024x6x4 (L)*	*cap_q7*	40.63M	338.56	49.63M	103.40	23.54	213.97
smallNORB *5x1600x6x4 (M)*	*cap_q7*	32.12M	267.65	43.49M	90.60	20.45	185.90
CIFAR-10 *10x64x5x4 (S)*	*cap_q7*	9.55M	79.58	14.22M	29.63	6.91	62.81

Table 37: Evaluation of capsule functions for Arm Cortex-M MCUs.

Dataset		GAP-8 (single-core) *RISC-V RV32IMCXpulp*		GAP-8 (octa-core) *RISC-V RV32IMCXpulp*	
		Clock Cycles	**Milliseconds**	**Clock Cycles**	**Milliseconds**
MNIST *10x1024x6x4 (L)*	*cap_parallel_q7*	20.32M	119.52	7.96M	46.83
smallNORB *5x1600x6x4 (M)*	*cap_parallel_q7*	16.26M	95.64	6.46M	38.03
CIFAR-10 *10x64x5x4 (S)*	*cap_parallel_q7*	4.55M	26.77	1.92M	11.28

Table 38: Evaluation of capsule functions for RISC-V MCUs.

7.6 Related Work

Zhang et al. [65] proposed a hybrid computing architecture to accelerate the routing procedure of CapsNets during inference in GPU platforms. The authors claim that even in modern GPUs, CapsNets exhibit low efficiency due to their routing procedure. Through runtime profiling, Zhang et al. [65] showed that the inefficiency of the routing procedure emerges from (i) tremendous data access to off-chip memory and (ii) intensive synchronizations to avoid race conditions on the limited GPU on-chip memory. To tackle this issue, the authors proposed a hybrid computing architecture named PIM-CapsNet. PIM-CapsNet offloads the routing procedure to an in-memory computation solution. Zhang et al. [65] stated their hybrid computing architecture is 2.44x faster than a GPU-only execution. The previous work was later extended with software optimizations. After profiling, Zhang et al. [66] stated that (i) nearby capsules have similar coupling coefficients (ii) and not all capsules require the same routing iterations to get stable coupling coefficients. Zhang et al. [66] proposed then to group capsules in 3x3 groups, where only the central capsule is used in the computation of the coupling coefficients. To address the latter finding, Zhang et al. [66] grouped capsules in two distinct groups: (i) one containing the capsules whose coupling coefficients are relatively stable and does not require the total routing iterations; (ii) and the other containing the capsules that benefit from the maximum number of iterations. With this software optimization, Zhang et al. [66] achieved a significantly 3.41x speedup.

To counteract the intensive matrix multiplications and the complexity of the dynamic routing, Marchisio et al. [64] proposed an RTL-level hardware architecture. At the core of their hardware accelerator, there is a processing element array, which is responsible for all matrix operations. This hardware accelerator also includes units specialized to speed up the ReLU, *squash*, and *softmax* activations. Despite only supporting inputs as 8-bit integers, the bit-width of internal operations varies along the pipeline. To trade off the area, delay, and energy consumption of the accelerator, Marchisio et al. [64] developed a framework that takes as input a given CapsNet model and a general architecture of the CapsNet accelerator and returns a set of optimal parameters for the accelerator. Results showed that the accelerator is 6x faster than a GPU-only solution (Nvidia GTX1070) [64]. Based on this work, Park et al. [67] proposed an accelerator enhanced for 3D CapsNets. This newer accelerator relies on a convolution core with 4 arrays of PEs, each one designed for multiplications with 8-bit operands. For the design of the dynamic routing core, Park et al.

[67] stated that most of the coupling coefficients practically maintain their value between updates. The dynamic routing core was designed to skip the route operation when the variation is less than 0.005%. These optimizations combined resulted in an accelerator that consumes 53.27% less energy than the solution proposed by Marchisio et al. [64].

In an attempt to reduce the memory footprint of CapsNets, Marchisio et al. [62] proposed a quantization framework. This framework explores different layer-wise and operation-wise precisions for obtaining a quantized CapsNet that trades off the classification accuracy and memory footprint. For this purpose, it takes as input a trained CapsNet model, a library of rounding schemes, an accuracy tolerance, and a maximum memory budget. For a CapsNet trained on the CIFAR-10 dataset, the framework reduces the memory footprint by 83% with only 0.15% accuracy loss.

Kakillioglu et al. [68] proposed a pruning technique to reduce the large set of parameters in CapsNets, enabling their deployment in resource-constrained devices. Weight pruning techniques work by removing connections between neurons of a trained model. Weights in a tensor are ranked according to their magnitude and then the smallest of them, for a given percentage, are set to zero. The layer-wise pruning technique developed by Kakillioglu et al. [68] enabled the deployment of CapsNets using 84.93% to 97.01% less memory than the original model proposed by Sabour et al. [53].

7.6.1 Gap Analysis

Table 39 puts our work in perspective against previous works. As observed, while prior research focused on accelerating the inference pass of CapsNets on GPUs, ASICs, and FPGA platforms, this article goes a step further and evaluates the deployment of CapsNets on COTS Arm and RISC-V MCUs targeted for intelligent IoT nodes. We extended readily-available APIs (CMSIS-NN and PULP-NN) for quantized ANNs to support both primary capsule and capsule layers. In addition to the price savings, our work has the potential to accelerate the adoption of CapsNets on the edge as it does not require novel and specific hardware solutions to be deployed. With effect, previous works tend to extend classic computing platforms with custom hardware units to accelerate CapsNets. To comply with the 8-bit data of our software kernels, we developed a quantization framework, also compatible with CMSIS-NN and PULP-NN.

Inspired by the work of Marchisio et al. [62], future work may encompass the tuning of our software kernels, along with CMSIS-NN and PULP-NN APIs, to support mixed bit-width. We believe that by giving more precision to weights with a higher impact on the final decision, we could take a step further in minimizing the effects of CapsNet quantization on accuracy. Following the work by Kakillioglu et al. [68], we can also employ a pruning scheme to enhance our quantization framework. Preference must be given to structured pruning to avoid increased overhead during inference. Nevertheless, results detailed in Section 7.3 show that our software kernels, at this stage of development, already achieved impressive gains in terms of memory footprint and latency, while relying on low-cost platforms available on the market. It is infeasible to compare the latency speedup, memory footprint reduction, and accuracy loss of our solution with the others in Table 39 as the baseline reference is not the same.

R&D Study	Target Platform	Problems Addressed	Proposal
Zhang et al. [108]	GPU	Latency	1. Hardware accelerator to offload the computation of the dynamic routing from the GPU
Zhang et al. [66]	GPU	Latency	1. Software optimization to optimize the dynamic routing on an hardware accelerator coupled to a GPU
Marchisio et al. [64]	FPGA	Latency Energy	1. RTL-level hardware architecture for CapsNet inference 2. Software to optimize the RTL architecture given a CapsNet
Park et al. [67]	ASIC	Latency Energy	1. RTL-level hardware architecture for 3D CapsNet inference
Marchisio et al. [62]	GPU FPGA	Memory	1. CapsNet quantization tool supporting mixed bit-width
Kakillioglu et al. [68]	GPU CPU	Memory	1. Weight pruning tailored for CapsNets
Costa et. al	MCU	Memory Latency	1. CapsNet quantization tool supporting fixed int-8 data 2. API for CapsNets inference in Armv7E-M, Armv8-M MCUs 3. API for CapsNets inference in RISC-V (RV32IMCXpulp) MCUs

Table 39: Gap analysis: Capsule Networks at the edge.

7.7 Conclusion

In this article, we proposed the first public and open-source software kernels to enable the deployment of CapsNets in Arm Cortex-M and RISC-V MCUs. We extended the CMSIS-NN and PULP-NN to support the inference pass of CapsNets. As CapsNets are typically trained in 32-bit floating-point and low-end MCUs usually do not feature an FPU, we also proposed a framework, built upon TensorFlow, to quantize CapsNets to int-8 data.

Results show that our work succeeds in addressing the latency and memory footprint problems inherent to CapsNets, proving that it is possible to implement them efficiently on the deep edge. Our quantization framework reduces the memory footprint of a CapsNet by about 75% with negligible accuracy loss. Furthermore, the latency induced by our software kernels on the execution of primary capsule and capsule layers with medium-sized kernels can be lower than a few tenths of a second. To the best of our knowledge, this is the first work evaluating the feasibility of implementing CapsNets on resource-constrained MCUs.

Bibliography

[1] R. Miotto et al. “Deep learning for healthcare: review, opportunities and challenges”. In: *Briefings in Bioinformatics* (2017) (cit. on pp. 2, 87).

[2] S. Ahmed et al. “Artificial intelligence and machine learning in finance: A bibliometric review”. In: *Research in International Business and Finance* (2022) (cit. on pp. 2, 87).

[3] M. Costa et al. “Detecting Driver’s Fatigue, Distraction and Activity Using a Non-Intrusive Ai-Based Monitoring System”. In: *Journal of Artificial Intelligence and Soft Computing Research* (2019) (cit. on pp. 2, 117).

[4] Y. Li et al. “Deep Learning for LiDAR Point Clouds in Autonomous Driving: A Review”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2021) (cit. on pp. 2, 87).

[5] M. G. S. Murshed et al. “Machine Learning at the Network Edge: A Survey”. In: *ACM Comput. Surv.* (2021) (cit. on pp. 2, 62, 87).

[6] I. Stoica et al. *A Berkeley View of Systems Challenges for AI*. Tech. rep. EECS Department, University of California, Berkeley, 2017 (cit. on pp. 2, 3, 62, 87, 88, 117).

[7] H. Li, K. Ota, and M. Dong. “Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing”. In: *IEEE Network* (2018) (cit. on pp. 2, 117).

[8] R. Bernhard, P.-A. Moellic, and J.-M. Dutertre. “Impact of Low-Bitwidth Quantization on the Adversarial Robustness for Embedded Neural Networks”. In: *International Conference on Cyberworlds (CW)*. 2019 (cit. on pp. 2, 3, 5, 40, 50, 51, 87, 88, 90, 99, 101, 106, 109, 110).

[9] P. M. VanNostrand et al. “Confidential Deep Learning: Executing Proprietary Models on Untrusted Devices”. In: *CoRR* (2019) (cit. on pp. 2, 4, 7, 36, 37, 47, 48, 62, 63, 70, 72).

[10] M. P. Véstias et al. “Moving Deep Learning to the Edge”. In: *Algorithms* (2020) (cit. on pp. 2, 117).

[11] F. Mo et al. “DarkneTZ: Towards Model Privacy at the Edge Using Trusted Execution Environments”. In: *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. 2020 (cit. on pp. 2, 4, 7, 36, 39, 48, 49, 62, 63, 71–73).

[12] S. P. Bayerl et al. “Offline Model Guard: Secure and Private ML on Mobile Devices”. In: *Proceedings of the 23rd Conference on Design, Automation and Test in Europe*. 2020 (cit. on pp. 2, 4, 7, 36, 37, 47, 48, 62, 63, 70, 72).

[13] L. Hanzlik et al. “MLCapsule: Guarded Offline Deployment of Machine Learning as a Service”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2021 (cit. on pp. 2, 4, 7, 36, 37, 47, 48, 62, 63, 70, 72).

[14] M. Gorsline, J. Smith, and C. Merkel. “On the Adversarial Robustness of Quantized Neural Networks”. In: *Proceedings of Great Lakes Symposium on VLSI*. 2021 (cit. on pp. 2, 3, 5, 42, 50, 51, 87, 88, 90, 98, 99, 106, 109, 110).

[15] R. Liu et al. “SecDeep: Secure and Performant On-Device Deep Learning Inference Framework for Mobile and IoT Devices”. In: *International Conference on Internet-of-Things Design and Implementation*. 2021 (cit. on pp. 2, 4, 7, 36, 37, 47, 48, 62, 63, 70, 72).

[16] J. Hou et al. “Model Protection: Real-Time Privacy-Preserving Inference Service for Model Privacy at the Edge”. In: *IEEE Transactions on Dependable and Secure Computing* (2022) (cit. on pp. 2, 4, 7, 36, 39, 48, 49, 62, 63, 71–73).

[17] X. Xie et al. “Memory-Efficient and Secure DNN Inference on TrustZone-enabled Consumer IoT Devices”. In: *IEEE INFOCOM - IEEE Conference on Computer Communications*. 2024 (cit. on pp. 2, 4, 7, 36, 38, 48, 62, 63, 70, 72).

[18] L. Lai, N. Suda, and V. Chandra. “CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs”. In: *CoRR* (2018) (cit. on pp. 2, 3, 5, 7, 14–16, 51, 63, 65, 68, 88, 90, 117, 123, 124, 129).

[19] A. Garofalo et al. “PULP-NN: A Computing Library for Quantized Neural Network inference at the edge on RISC-V Based Parallel Ultra Low Power Clusters”. In: *IEEE International Conference on Electronics, Circuits and Systems*. 2019 (cit. on pp. 3, 7, 16, 68, 88, 90, 117, 124, 130).

[20] R. David et al. “TensorFlow Lite Micro: Embedded ML on TinyML Systems”. In: *CoRR* (2020) (cit. on pp. 3, 51, 63, 88).

[21] E. Flamand et al. “GAP-8: A RISC-V SoC for AI at the Edge of the IoT”. In: *IEEE International Conference on Application-Specific Systems, Architectures and Processors*. 2018 (cit. on pp. 3, 5, 7, 17, 68, 90, 126).

[22] *GAP9 processor | GreenWaves Technologies*. `https://greenwaves-technologies.com/gap9_processor/`. [Accessed 04-06-2024] (cit. on pp. 3, 7, 17, 68).

[23] M. Scherer et al. “A 1036 TOp/s/W, 12.2 mW, 2.72 µJ/Inference All Digital TNN Accelerator in 22 nm FDX Technology for TinyML Applications”. In: *IEEE Symposium in Low-Power and High-Speed Chips*. 2022 (cit. on pp. 3, 90).

[24] C. R. Banbury et al. “MLPerf Tiny Benchmark”. In: *CoRR* (2021) (cit. on pp. 3, 6, 14, 51, 55, 63, 64, 68, 78, 90–92, 111).

[25] N. Schizas et al. “TinyML for Ultra-Low Power AI and Large Scale IoT Deployments: A Systematic Review”. In: *Future Internet* (2022) (cit. on pp. 3, 14).

[26] P. P. Ray. “A review on TinyML: State-of-the-art and prospects”. In: *Journal of King Saud University - Computer and Information Sciences* (2022) (cit. on pp. 3, 68).

[27] Y. Abadade et al. “A Comprehensive Survey on TinyML”. In: *IEEE Access* (2023) (cit. on pp. 3, 14, 68).

[28] L. Capogrosso et al. “A Machine Learning-Oriented Survey on Tiny Machine Learning”. In: *IEEE Access* (2024) (cit. on pp. 3, 14).

[29] R. Kallimani et al. “TinyML: Tools, applications, challenges, and future research directions”. In: *Multimedia Tools and Applications* (2024) (cit. on pp. 3, 14).

[30] *TinyML: A Market Update*. Tech. rep. ABI Research, 2022 (cit. on pp. 3, 7, 68, 90).

[31] *TinyML: The Next Big Opportunity in Tech*. Tech. rep. ABI Research, 2021 (cit. on pp. 3, 7, 68, 90).

[32] *Global Microcontroller (MCU) Market Size Share Analysis - Growth Trends Forecasts (2024 - 2029)*. Tech. rep. Mordor Intelligence, 2024 (cit. on pp. 3, 7).

[33] M. Rigaki and S. Garcia. “A Survey of Privacy Attacks in Machine Learning”. In: *ACM Comput. Surv.* (2023) (cit. on pp. 4, 62).

[34] R. Shokri et al. “Membership Inference Attacks Against Machine Learning Models”. In: *IEEE Symposium on Security and Privacy (SP)*. 2017 (cit. on pp. 4, 62).

[35] H. Hu et al. “Membership Inference Attacks on Machine Learning: A Survey”. In: *ACM Comput. Surv.* (2022) (cit. on pp. 4, 62).

[36] T. Nakai, D. Suzuki, and T. Fujino. “Towards Trained Model Confidentiality and Integrity Using Trusted Execution Environments”. In: *Applied Cryptography and Network Security Workshops*. 2021 (cit. on pp. 4, 7, 36, 37, 47, 48, 63, 70, 72).

[37] M. Yang et al. “Penetralium: Privacy-preserving and memory-efficient neural network inference at the edge”. In: *Future Generation Computer Systems* (2024) (cit. on pp. 4, 7, 36, 38, 48, 63, 70, 72).

[38] F. Tramer and D. Boneh. “Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware”. In: *International Conference on Learning Representations*. 2019 (cit. on pp. 4, 7, 36, 39, 48, 49, 63, 71–73).

[39] Z. Sun et al. “ShadowNet: A Secure and Efficient On-device Model Inference System for Convolutional Neural Networks”. In: *IEEE Symposium on Security and Privacy (SP)*. 2023 (cit. on pp. 4, 7, 36, 39, 48, 49, 63, 71–73).

[40] R. C. Nair, M. Rao, and M. V N. “SecureFrameNet:A rich computing capable secure framework for deploying neural network models on edge protected system.” In: *Proceedings of the Third International Conference on AI-ML Systems*. 2024 (cit. on pp. 4, 7, 36, 39, 48, 49, 63, 71–73).

[41] Z. Liu et al. “MirrorNet: A TEE-Friendly Framework for Secure On-Device DNN Inference”. In: *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 2023 (cit. on pp. 4, 7, 36, 38, 48, 63, 70, 72, 73).

[42] S. Pinto and N. Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. In: *ACM Comput. Surv.* (2019) (cit. on pp. 4, 5, 7, 13, 17, 18, 67, 68).

[43] D. Oliveira, T. Gomes, and S. Pinto. “uTango: An Open-Source TEE for IoT Devices”. In: *IEEE Access* (2022) (cit. on pp. 4, 5, 7, 13, 18, 63, 67, 68).

[44] M. A. Mukhtar, M. K. Bhatti, and G. Gogniat. “Architectures for Security: A comparative analysis of hardware security features in Intel SGX and ARM TrustZone”. In: *2nd International Conference on Communication, Computing and Digital systems (C-CODE)*. 2019 (cit. on p. 5).

[45] X. Tan et al. “SoK: Where’s the “up”?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems”. In: *18th USENIX WOOT Conference on Offensive Technologies*. 2024 (cit. on pp. 5, 18, 68).

[46] X. Yuan et al. “Adversarial Examples: Attacks and Defenses for Deep Learning”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2019) (cit. on pp. 5, 22, 23, 88).

[47] X. Wang et al. “The security of machine learning in an adversarial setting: A survey”. In: *Journal of Parallel and Distributed Computing* (2019) (cit. on pp. 5, 88).

[48] S. Zhou et al. “Adversarial Attacks and Defenses in Deep Learning: From a Perspective of Cybersecurity”. In: *ACM Comput. Surv.* (2022) (cit. on pp. 5, 88).

[49] I. Shumailov et al. “To Compress Or Not To Compress: Understanding The Interactions Between Adversarial Attacks And Neural Network Compression”. In: *Proceedings of ML and Systems*. 2019 (cit. on pp. 5, 40, 49–51, 88, 99, 108, 110).

[50] J. Lin, C. Gan, and S. Han. “Defensive Quantization: When Efficiency Meets Robustness”. In: *International Conference on Learning Representations*. 2019 (cit. on pp. 5, 7, 42, 49–51, 88, 100, 106, 109–111).

[51] K. Duncan et al. “Relative Robustness of Quantized Neural Networks Against Adversarial Attacks”. In: *International Joint Conference on Neural Networks*. 2020 (cit. on pp. 5, 7, 41, 49–51, 88, 100, 108–111).

[52] C. Song, E. Fallon, and H. H. Li. “Improving Adversarial Robustness in Weight-Quantized Neural Networks”. In: *CoRR* (2020) (cit. on pp. 5, 43, 50, 51, 100, 109, 110).

[53] S. Sabour, N. Frosst, and G. E. Hinton. “Dynamic Routing between Capsules”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017 (cit. on pp. 6–8, 31–33, 116, 119–121, 129, 140).

[54] M. Kwabena Patrick et al. “Capsule Networks – A survey”. In: *Journal of King Saud University - Computer and Information Sciences* (2019) (cit. on pp. 6–8, 116–118).

[55] El Alaoui-Elfels, Omaima and Gadi, Taoufiq. “From Auto-encoders to Capsule Networks: A Survey”. In: *E3S Web Conf.* (2021) (cit. on pp. 6–8, 31–35, 116–118).

[56] N. Papernot et al. “Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks”. In: *IEEE Symposium on Security and Privacy (SP)*. 2016 (cit. on pp. 6, 28, 29, 95).

[57] A. Madry et al. “Towards Deep Learning Models Resistant to Adversarial Attacks”. In: *International Conference on Learning Representations*. 2018 (cit. on pp. 6, 24, 25, 29, 94, 95).

[58] F. Tramèr et al. “Ensemble Adversarial Training: Attacks and Defenses”. In: *International Conference on Learning Representations*. 2018 (cit. on pp. 6, 29, 30, 95).

[59] Q. Bouniot, R. Audigier, and A. Loesch. “Optimal Transport as a Defense Against Adversarial Attacks”. In: *International Conference on Pattern Recognition*. 2021 (cit. on pp. 6, 29, 30, 95).

[60] W. Xu, D. Evans, and Y. Qi. “Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks”. In: *CoRR* (2017) (cit. on pp. 6, 29, 30, 95).

[61] Y. Song et al. “Pixeldefend: Leveraging generative models to understand and defend against adversarial examples”. In: *International Conference on Learning Representations*. 2018 (cit. on pp. 6, 29, 31, 95).

[62] A. Marchisio et al. “Q-CapsNets: A Specialized Framework for Quantizing Capsule Networks”. In: *57th ACM/IEEE Design Automation Conference (DAC)*. 2020 (cit. on pp. 6, 45, 52, 53, 117, 140, 141).

[63] A. Marchisio et al. “DESCNet: Developing Efficient Scratchpad Memories for Capsule Network Hardware”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020) (cit. on pp. 6, 117).

[64] A. Marchisio et al. “FEECA: Design Space Exploration for Low-Latency and Energy-Efficient Capsule Network Accelerators”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2021) (cit. on pp. 6, 44, 45, 52, 117, 139–141).

[65] X. Zhang et al. “Enabling Highly Efficient Capsule Networks Processing Through A PIM-Based Architecture Design”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020 (cit. on pp. 6, 43, 52, 53, 117, 139).

[66] X. Zhang et al. “Enabling Highly Efficient Capsule Networks Processing Through Software-Hardware Co-Design”. In: *IEEE Transactions on Computers* (2021) (cit. on pp. 6, 43, 52, 53, 117, 139, 141).

[67] G. Park et al. “A 1.15 TOPS/W Energy-Efficient Capsule Network Accelerator for Real-Time 3D Point Cloud Segmentation in Mobile Environment”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* (2020) (cit. on pp. 6, 45, 52, 53, 117, 139, 141).

[68] B. Kakillioglu et al. “3D Capsule Networks for Object Classification With Weight Pruning”. In: *IEEE Access* (2020) (cit. on pp. 6, 46, 52, 53, 58, 117, 140, 141).

[69] M. Costa, T. Gomes, and S. Pinto. “SecureQNN: Shielding the Intellectual Property of QNNs in TinyML Systems”. In: *Submitted to IEEE Internet of Things Journal* (2025) (cit. on pp. 9, 10).

[70] M. Costa and S. Pinto. “David and Goliath: An Empirical Evaluation of Attacks and Defenses for QNNs at the Deep Edge”. In: *IEEE 9th European Symposium on Security and Privacy (EuroSP)*. 2024 (cit. on pp. 9, 11, 63).

[71] M. Costa et al. “Shifting Capsule Networks from the Cloud to the Deep Edge”. In: *ACM Trans. Intell. Syst. Technol.* (2022) (cit. on pp. 9, 85, 88).

[72] M. Costa et al. “SecureQNN: Introducing a Privacy-Preserving Framework for QNNs at the Deep Edge”. In: *Data Science and Artificial Intelligence*. 2023 (cit. on pp. 9–12, 64).

[73] D. Costa, M. Costa, and S. Pinto. “Train Me If You Can: Decentralized Learning on the Deep Edge”. In: *Applied Sciences* (2022) (cit. on pp. 9, 12, 62, 87, 91).

[74] J. Yiu. *Cortex-M for Beginners*. Tech. rep. Arm, 2016 (cit. on p. 13).

[75] M. Rafiquzzaman. *Microcontroller Theory and Applications with the PIC18F*. John Wiley & Sons, 2025 (cit. on p. 13).

[76] V. Rajapakse, I. Karunanayake, and N. Ahmed. “Intelligence at the Extreme Edge: A Survey on Reformable TinyML”. In: *ACM Comput. Surv.* (2023) (cit. on pp. 13, 14).

[77] F. V. TX Zhuo and H. Collins. *Why TinyML is a giant opportunity*. `https://venturebeat.com/ai/why-tinyml-is-a-giant-opportunity/`. [Accessed 24-01-2025] (cit. on p. 14).

[78] *Microcontroller Market Analysis by Product (8-Bit, 16-Bit, 32-Bit, 64-Bit), By Type (Peripheral Interface Controller (PIC), ARM, 8051, TriCore, Others), By Architecture, By Instruction Set, By Application, And Region Segment Forecasts, 2024 - 2030*. Tech. rep. Grand View Research, 2024 (cit. on p. 14).

[79] H. Gokul et al. “Gait Recovery System for Parkinson’s Disease using Machine Learning on Embedded Platforms”. In: *IEEE International Systems Conference (SysCon)*. 2020 (cit. on p. 14).

[80] R. Zanetti, A. Aminifar, and D. Atienza. “Robust Epileptic Seizure Detection on Wearable Systems with Reduced False-Alarm Rate”. In: *42nd Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC)*. 2020 (cit. on p. 14).

[81] C. D. C□leanu, C. L. Sîrbu, and G. Simion. “Deep Neural Architectures for Contrast Enhanced Ultrasound (CEUS) Focal Liver Lesions Automated Diagnosis”. In: *Sensors* (2021) (cit. on p. 14).

[82] J. Mishra, T. Malche, and A. Hirawat. “Embedded Intelligence for Smart Home Using TinyML Approach to Keyword Spotting”. In: *Engineering Proceedings* (2024) (cit. on p. 14).

[83] V. Tsoukas et al. “A Gas Leakage Detection Device Based on the Technology of TinyML”. In: *Technologies* (2023) (cit. on p. 14).

[84] A. N. Roshan et al. “Adaptive Traffic Control With TinyML”. In: *Sixth International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. 2021 (cit. on p. 14).

[85] N. N. Alajlan and D. M. Ibrahim. “DDD TinyML: A TinyML-Based Driver Drowsiness Detection Model Using Deep Learning”. In: *Sensors* (2023) (cit. on p. 14).

[86] Y. Zhang et al. “Enabling Autonomous Structural Inspections with Tiny Machine Learning on UAVs”. In: *IEEE Sensors Applications Symposium (SAS)*. 2024 (cit. on p. 14).

[87] A. M. Hayajneh et al. “Tiny machine learning on the edge: A framework for transfer learning empowered unmanned aerial vehicle assisted smart farming”. In: *IET Smart Cities* (2024) (cit. on p. 14).

[88] Y. S. H. Annadata, V. Moganarengam, and T. Nikoubin. “TinyML Powered Drone in Agriculture Application”. In: *IEEE 17th Dallas Circuits and Systems Conference (DCAS)*. 2024 (cit. on p. 14).

[89] Arm. *ARMv7-M Architecture Reference Manual*. 2021 (cit. on p. 15).

[90] Arm. *ARMv8-M Architecture Reference Manual*. 2024 (cit. on p. 15).

[91] J. Yiu. *Introduction to Armv8.1-M Architecture*. Tech. rep. Arm, 2019 (cit. on p. 15).

[92] Arm. *ARM Cortex-M4 Processor Technical Reference Manual*. 2010 (cit. on p. 15).

[93] Arm. *ARM Cortex-M85 Processor Technical Reference Manual*. 2024 (cit. on p. 15).

[94] Synopsys. *What is RISC-V? – How Does it Work?* `https://www.synopsys.com/glossary/what-is-risc-v.html`. [Accessed 28-01-2025] (cit. on p. 16).

[95] A. Waterman et al. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*. Tech. rep. EECS department, University of California, Berkeley, 2014 (cit. on p. 16).

[96] M. Gautschi et al. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2017) (cit. on p. 16).

[97] T. Firmware. *TrustedFirmware-M*. `https://www.trustedfirmware.org/projects/tf-m/`. [Accessed 27-01-2025] (cit. on p. 19).

[98] P. Wang et al. “Towards Accurate Post-training Network Quantization via Bit-Split and Stitching”. In: *Proceedings of the 37th International Conference on Machine Learning*. 2020 (cit. on pp. 19–21, 121–123).

[99] H. Wu et al. “Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation”. In: *CoRR* (2020) (cit. on pp. 19–21, 121, 122).

[100] A. Gholami et al. “A Survey of Quantization Methods for Efficient Neural Network Inference”. In: *CoRR* (2021) (cit. on pp. 19–22).

[101] T. Liang et al. “Pruning and quantization for deep neural network acceleration: A survey”. In: *Neurocomputing* (2021) (cit. on pp. 19–21, 121, 122).

[102] P.-E. Novac et al. “Quantization and Deployment of Deep Neural Networks on Microcontrollers”. In: *Sensors* (2021) (cit. on pp. 19–22, 121–123).

[103] V. Vanhoucke, A. Senior, and M. Z. Mao. “Improving the speed of neural networks on CPUs”. In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS*. 2011 (cit. on p. 19).

[104] M. Rakka et al. “A Review of State-of-the-art Mixed-Precision Neural Network Frameworks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2024) (cit. on pp. 19, 21).

[105] C. Szegedy et al. “Intriguing properties of neural networks”. In: *International Conference on Learning Representations*. 2014 (cit. on p. 22).

[106] M. Ozdag. “Adversarial Attacks and Defenses Against Deep Neural Networks: A Survey”. In: *Procedia Computer Science* (2018) (cit. on pp. 22, 23).

[107] H. Xu et al. “Adversarial Attacks and Defenses in Images, Graphs and Text: A Review”. In: *International Journal of Automation and Computing* (2019) (cit. on pp. 22, 23).

[108] J. Zhang and C. Li. “Adversarial Examples: Opportunities and Challenges”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2020) (cit. on pp. 22, 23, 52, 141).

[109] H. Liang et al. “Adversarial Attack and Defense: A Survey”. In: *Electronics* (2022) (cit. on pp. 22, 23).

[110] N. Papernot et al. “SoK: Security and Privacy in Machine Learning”. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018 (cit. on pp. 23, 88, 91).

[111] K. Ren et al. “Adversarial Attacks and Defenses in Deep Learning”. In: *Engineering* (2020) (cit. on p. 23).

[112] P.-Y. Chen et al. “EAD: Elastic-Net Attacks to Deep Neural Networks via Adversarial Examples”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* (2018) (cit. on pp. 23, 24, 26, 94).

[113] I. J. Goodfellow, J. Shlens, and C. Szegedy. “Explaining and Harnessing Adversarial Examples”. In: *3rd International Conference on Learning Representations, ICLR*. 2015 (cit. on pp. 24, 29, 95).

[114] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. “DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 (cit. on pp. 24, 25, 94).

[115] N. Papernot et al. “The Limitations of Deep Learning in Adversarial Settings”. In: *IEEE European Symposium on Security and Privacy (EuroSP)*. 2016 (cit. on pp. 24, 25, 94).

[116] N. Carlini and D. Wagner. “Towards Evaluating the Robustness of Neural Networks”. In: *IEEE Symposium on Security and Privacy (SP)*. 2017 (cit. on pp. 24, 25, 94).

[117] A. Kurakin, I. J. Goodfellow, and S. Bengio. “Adversarial examples in the physical world”. In: *CoRR* (2016) (cit. on pp. 24, 25).

[118] F. Croce and M. Hein. “Minimally Distorted Adversarial Examples with a Fast Adaptive Boundary Attack”. In: *Proceedings of the 37th International Conference on Machine Learning*. 2020 (cit. on pp. 24, 26).

[119] F. Croce and M. Hein. “Reliable Evaluation of Adversarial Robustness with an Ensemble of Diverse Parameter-Free Attacks”. In: *Proceedings of International Conference on Machine Learning*. 2020 (cit. on pp. 24, 26, 94, 95).

[120] P.-Y. Chen et al. “ZOO: Zeroth Order Optimization Based Black-Box Attacks to Deep Neural Networks without Training Substitute Models”. In: *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. 2017 (cit. on pp. 24, 27, 94).

[121] M. Andriushchenko et al. “Square Attack: A Query-Efficient Black-Box Adversarial Attack via Random Search”. In: *Computer Vision – ECCV 2020*. 2020 (cit. on pp. 24, 27, 94).

[122] W. Brendel, J. Rauber, and M. Bethge. “Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models”. In: *International Conference on Learning Representations*. 2018 (cit. on pp. 24, 27, 94).

[123] A. Rahmati et al. “GeoDA: A Geometric Framework for Black-Box Adversarial Attacks”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020 (cit. on pp. 24, 28, 94).

[124] G. E. Hinton, A. Krizhevsky, and S. D. Wang. “Transforming Auto-Encoders”. In: *Artificial Neural Networks and Machine Learning – ICANN 2011*. 2011 (cit. on pp. 31, 32, 116, 118).

[125] G. E. Hinton, S. Sabour, and N. Frosst. “Matrix capsules with EM routing”. In: *International Conference on Learning Representations*. 2018 (cit. on pp. 31, 32, 34, 35, 116, 119).

[126] M. Kwabena Patrick et al. “Capsule Networks – A survey”. In: *Journal of King Saud University - Computer and Information Sciences* (2022) (cit. on pp. 31–34).

[127] M. U. Haq, M. A. J. Sethi, and A. U. Rehman. “Capsule Network with Its Limitation, Modification, and Applications—A Survey”. In: *Machine Learning and Knowledge Extraction* (2023) (cit. on pp. 31–33).

[128] F. De Sousa Ribeiro et al. “Object-centric Learning with Capsule Networks: A Survey”. In: *ACM Comput. Surv.* (2024) (cit. on pp. 31–35).

[129] A. F. Al-rahhawi and N. Aydın Atasoy. “Optimizing Capsule Network Performance with Enhanced Squash Function for Classification Large-Scale Bone Marrow Cells”. In: *Black Sea Journal of Engineering and Science* (2024) (cit. on p. 34).

[130] G. Chen et al. “Enhancing classification efficiency in capsule networks through windowed routing: tackling gradient vanishing, dynamic routing, and computational complexity challenges”. In: *Complex Intelligent Systems* (2024) (cit. on p. 34).

[131] J. Li et al. “TEEm: Supporting Large Memory for Trusted Applications in ARM TrustZone”. In: *IEEE Access* (2024) (cit. on pp. 36, 38, 48).

[132] Y. Zhao et al. “Mayo: A Framework for Auto-generating Hardware Friendly Deep Neural Networks”. In: *Proceedings of the 2nd International Workshop on Embedded and Mobile Deep Learning*. 2018 (cit. on pp. 40, 50).

[133] M. Courbariaux, Y. Bengio, and J.-P. David. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: *Advances in Neural Information Processing Systems*. 2015 (cit. on p. 40).

[134] I. Hubara et al. “Binarized Neural Networks”. In: *Advances in Neural Information Processing Systems*. 2016 (cit. on p. 40).

[135] S. Zhou et al. “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients”. In: *CoRR* (2016) (cit. on p. 40).

[136] X. Huang et al. “Safety Verification of Deep Neural Networks”. In: *Computer Aided Verification*. 2017 (cit. on p. 41).

[137] Y. Guo. “A Survey on Methods and Theories of Quantized Neural Networks”. In: *CoRR* (2018) (cit. on p. 42).

[138] A. Rahoof, V. Chaturvedi, and M. Shafique. “FastCaps: A Design Methodology for Accelerating Capsule Network on Field Programmable Gate Arrays”. In: *International Joint Conference on Neural Networks (IJCNN)*. 2023 (cit. on pp. 44, 52, 53).

[139] M. Hemmati et al. “Hardware Acceleration of Capsule Networks for Real-Time Applications”. In: *27th Euromicro Conference on Digital System Design (DSD)*. 2024 (cit. on pp. 45, 52).

[140] F. de Vita et al. “Quantitative Analysis of Deep Leaf: a Plant Disease Detector on the Smart Edge”. In: *IEEE International Conference on Smart Computing (SMARTCOMP)*. 2020 (cit. on pp. 51, 92, 111).

[141] B. Ram. *Numerical Methods*. Pearson, 2009 (cit. on pp. 53, 127, 128).

[142] L. Hu et al. “Defenses to Membership Inference Attacks: A Survey”. In: *ACM Comput. Surv.* (2023) (cit. on pp. 57, 84).

[143] C. Donahue, J. McAuley, and M. Puckette. “Adversarial Audio Synthesis”. In: *International Conference on Learning Representations*. 2019 (cit. on pp. 58, 84).

[144] I. Lucan Or□an, C. Seiculescu, and C. D. C□leanu. “A Brief Review of Deep Neural Network Implementations for ARM Cortex-M Processor”. In: *Electronics* (2022) (cit. on pp. 63, 92).

[145] C. Rodrigues, D. Oliveira, and S. Pinto. “BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect”. In: *IEEE Symposium on Security and Privacy (SP)*. 2024 (cit. on p. 67).

[146] *Global Microcontroller Market Size By Product (8-Bit, 16-Bit, 32-Bit), By Application (Automobile, Communication, Healthcare), By Geographic Scope And Forecast*. Tech. rep. Verified Market Research, 2024. url: `https://www.verifiedmarketresearch.com/product/microcontroller-market` (cit. on p. 68).

[147] M. AskariHemmat et al. “Quark: An Integer RISC-V Vector Processor for Sub-Byte Quantized DNN Inference”. In: *IEEE International Symposium on Circuits and Systems (ISCAS)*. 2023 (cit. on p. 68).

[148] I. Hoyer et al. “Design of Hardware Accelerators for Optimized and Quantized Neural Networks to Detect Atrial Fibrillation in Patch ECG Device with RISC-V”. In: *Sensors* (2023) (cit. on p. 68).

[149] L. Dutta and S. Bharali. “TinyML Meets IoT: A Comprehensive Survey”. In: *Internet of Things* (2021) (cit. on p. 68).

[150] I. Goodfellow et al. “Generative Adversarial Nets”. In: *Adv. in Neural Information Processing Systems*. 2014 (cit. on p. 69).

[151] A. Creswell et al. “Generative Adversarial Networks: An Overview”. In: *IEEE Signal Processing Magazine* (2018) (cit. on p. 69).

[152] J. Gui et al. “A Review on Generative Adversarial Networks: Algorithms, Theory, and Applications”. In: *IEEE Trans. on Knowledge and Data Engineering* (2023) (cit. on p. 69).

[153] M. Kang, J. Shin, and J. Park. “StudioGAN: A Taxonomy and Benchmark of GANs for Image Synthesis”. In: *IEEE Trans. on Pattern Analysis and Machine Intelligence* (2023) (cit. on pp. 69, 74).

[154] S. Bond-Taylor et al. “Deep Generative Modelling: A Comparative Review of VAEs, GANs, Normalizing Flows, Energy-Based and Autoregressive Models”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022) (cit. on p. 74).

[155] L. Wang et al. “A State-of-the-Art Review on Image Synthesis With Generative Adversarial Networks”. In: *IEEE Access* (2020) (cit. on p. 74).

[156] J. Yosinski et al. “How transferable are features in deep neural networks?” In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. 2014 (cit. on p. 75).

[157] K. Weiss, T. M. Khoshgoftaar, and D. Wang. “A survey of transfer learning”. In: *Journal of Big Data* (2016) (cit. on p. 75).

[158] F. Zhuang et al. “A Comprehensive Survey on Transfer Learning”. In: *Proceedings of the IEEE* (2021). url: https://doi.org/10.1109/JPROC.2020.3004555 (cit. on p. 75).

[159] A. Nadalini et al. “A 3 TOPS/W RISC-V Parallel Cluster for Inference of Fine-Grain Mixed-Precision Quantized Neural Networks”. In: *IEEE Computer Society Annual Symp. on VLSI*. 2023 (cit. on p. 90).

[160] M. Nicolae et al. “Adversarial Robustness Toolbox v0.2.2”. In: *CoRR* (2018) (cit. on pp. 93, 96).

[161] N. Papernot et al. “Technical Report on the CleverHans v2.1.0 Adversarial Examples Library”. In: *CoRR* (2018) (cit. on p. 93).

[162] J. Rauber et al. “Foolbox Native: Fast adversarial attacks to benchmark the robustness of machine learning models in PyTorch, TensorFlow, and JAX”. In: *Journal of Open Source Software* (2020) (cit. on p. 93).

[163] J. Rauber, W. Brendel, and M. Bethge. “Foolbox: A Python toolbox to benchmark the robustness of machine learning models”. In: *Reliable Machine Learning in the Wild Workshop, Inter. Conf. on Machine Learning*. 2017 (cit. on p. 93).

[164] S. Zhang, H. Gao, and Q. Rao. “Defense Against Adversarial Attacks by Reconstructing Images”. In: *IEEE Transactions on Image Processing* (2021) (cit. on p. 95).

[165] C. Ferrari et al. “(Compress and Restore): A Robust Defense Against Adversarial Attacks on Image Classification”. In: *ACM Trans. Multimedia Comput. Commun. Appl.* (2023) (cit. on p. 95).

[166] J. Su, D. V. Vargas, and K. Sakurai. “One Pixel Attack for Fooling Deep Neural Networks”. In: *IEEE Transactions on Evolutionary Computation* (2019) (cit. on p. 116).

[167] K. Sadeghi, A. Banerjee, and S. K. S. Gupta. “A System-Driven Taxonomy of Attacks and Defenses in Adversarial Machine Learning”. In: *IEEE Transactions on Emerging Topics in Computational Intelligence* (2020) (cit. on p. 116).

[168] A. Kosiorek et al. “Stacked Capsule Autoencoders”. In: *Advances in Neural Information Processing Systems*. 2019 (cit. on pp. 116, 119).

[169] S. Toraman, T. B. Alakus, and I. Turkoglu. “Convolutional capsnet: A novel artificial neural network approach to detect COVID-19 disease from X-ray images using capsule networks”. In: *Chaos, Solitons Fractals* (2020) (cit. on p. 117).

[170] A. Dinesh Kumar. “Novel Deep Learning Model for Traffic Sign Detection Using Capsule Networks”. In: *arXiv e-prints* (2018) (cit. on p. 117).

[171] N. Pari S., T. Mohana, and V. Akshaya. “Real-Time Traffic Sign Detection using Capsule Network”. In: *11th International Conference on Advanced Computing (ICoAC)*. 2019 (cit. on p. 117).

[172] M. Wang et al. “Towards Linear Time Neural Machine Translation with Capsule Networks”. In: *CoRR* (2018) (cit. on p. 117).

[173] R. Katebi et al. “Galaxy morphology prediction using Capsule Networks”. In: *Monthly Notices of the Royal Astronomical Society* (2019) (cit. on p. 117).

[174] J. Lu et al. “Efficient Hardware Architecture of Convolutional Neural Network for ECG Classification in Wearable Healthcare Device”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2021) (cit. on p. 117).

[175] J. Li et al. “A Survey on Capsule Networks: Evolution, Application, and Future Development”. In: *International Conference on High Performance Big Data and Intelligent Systems (HPBD IS)*. 2021 (cit. on p. 118).

[176] Z. Sun et al. “Overview of Capsule Neural Networks”. In: *Journal of Internet Technology* (2022) (cit. on p. 118).

[177] A. Gholami et al. “A Survey of Quantization Methods for Efficient Neural Network Inference”. In: *CoRR* (2021) (cit. on pp. 121–123).

[178] B. Jacob et al. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018 (cit. on p. 122).

[179] H. Dong, Z. Ding, and S. Zhang. *Deep Reinforcement Learning: Fundamentals, Research and Applications*. Springer Singapore, 2020 (cit. on p. 123).

[180] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* (1998) (cit. on p. 133).

[181] Y. LeCun, F. J. Huang, and L. Bottou. “Learning methods for generic object recognition with invariance to pose and lighting”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2004 (cit. on p. 133).

[182] A. Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009 (cit. on p. 133).