Constrained deep learning for MEMS sensors-based applications

Minh Tri Lê

▶ **To cite this version:**

Minh Tri Lê. Constrained deep learning for MEMS sensors-based applications. Machine Learning [cs.LG]. Université Grenoble Alpes [2020-..], 2023. English. NNT : 2023GRALM035 . tel-04363136v2

HAL Id: tel-04363136

https://theses.hal.science/tel-04363136v2

Submitted on 8 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Université Grenoble Alpes

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Mathématiques et Informatique
Unité de recherche : Institut National de Recherche en Informatique et en Automatique

Deep Learning contraint pour la résolution de problèmes algorithmiques appliqués aux capteurs MEMS (Microsystème Electromécanique)

Constrained deep learning for MEMS sensors-based applications

Présentée par :

Minh Tri LÊ

Direction de thèse :

Julyan ARBEL — Directeur de thèse
CHARGE DE RECHERCHE HDR, INRIA CENTRE GRENOBLE-RHONE-ALPES

Etienne DE FORAS — Co-encadrant de thèse
Movea SAS

Rapporteurs :

MATHIEU SALZMANN
SENIOR SCIENTIST, ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE
EIMAN KANJO
PROFESSEUR, NOTTINGHAM TRENT UNIVERSITY

Thèse soutenue publiquement le **6 juillet 2023**, devant le jury composé de :

MATHIEU SALZMANN — Rapporteur
SENIOR SCIENTIST, ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE

EIMAN KANJO — Rapporteure
PROFESSEUR, NOTTINGHAM TRENT UNIVERSITY

VALERIE PERRIER — Présidente
PROFESSEUR DES UNIVERSITES, GRENOBLE INP

INNA KUCHER — Examinatrice
INGENIEUR DOCTEUR, CEA CENTRE DE PARIS-SACLAY

Invités :

ETIENNE DE FORAS
INGENIEUR DE RECHERCHE, MOVEA SAS
JULYAN ARBEL
CHARGE DE RECHERCHE HDR, INRIA CENTRE GRENOBLE-RHONEALPES

Abstract

Deep learning has become a powerful technique for solving complex problems across numerous domains, owing to its ability to learn and model intricate non-linear relationships from data. However, the substantial computational power, memory, and energy requirements of deep learning models make them unsuitable for deployment on devices with limited resources. Simultaneously, the emergence of microelectromechanical sensors (MEMS), microcontroller units (MCUs), and the Internet of Things (IoT) has resulted in a growing number of applications that depend on embedded systems for local data processing and environmental interaction. MEMS provide an interface to continuously sense data from the real world to the digital world. MCUs are low-cost devices with high consumer market volumes, where targeting the lowest-power hardware can result in billions in saving per year. Thus, delivering always-on, real-time sensing pose ultra-low power constraints, with direct and high technical and economic implications and challenges.

The intersection of deep learning and embedded systems has given rise to the field of tinyML, which offers significant opportunities and challenges. Deep learning promises the automation of the algorithm design process, which enables practitioners to customize a product behavior at will. The primary challenge lies in adapting deep learning techniques to operate efficiently on ultra-low-power MEMS-based devices with constrained resources and operations, while maintaining acceptable performance levels.

This thesis aims to provide strategies for optimizing power footprint and deploying deep learning models in ultra-low power settings for MEMS-based applications. We first investigate model compression methods, such as pruning, knowledge distillation, and quantization. Then, we enable end-to-end deployment of deep learning models for efficient inference on the most resource-constrained MCUs in the industry and state-of-the-art, effectively redefining the concept of ultra-low power as extreme-low power. Finally, we present a novel approach to generalize the quantization process, accommodating any number of bits, and extending to extreme quantization levels, such as 1-bit.

The outcomes of this research contribute to the advancement of tinyML and enable the broader adoption of intelligent sensing devices across various real-world applications.

Résumé

L'apprentissage profond est devenu une technique puissante pour résoudre des problèmes complexes dans de nombreux domaines, grâce à sa capacité à apprendre et à modéliser des relations non linéaires complexes à partir de données. Cependant, la puissance de calcul, la mémoire et les besoins énergétiques substantiels des modèles d'apprentissage profond peuvent les rendre inadaptés au déploiement sur des systèmes à ressources limitées. Parallèlement, l'émergence des capteurs microélectromécaniques (MEMS), des microcontrôleurs (MCUs) et de l'Internet des objets (IoT) a entraîné un nombre croissant d'applications qui dépendent de systèmes embarqués pour le traitement local des données et pouvant interagir avec l'environnement. Les MEMS fournissent une interface permettant de détecter en continu des données du monde réel vers le monde numérique. Les MCUs sont des plateformes à faible coût ayant des volumes de marché très élevés pour les consommateurs, où cibler le matériel à la plus faible consommation peut entraîner des économies de plusieurs milliards par an. Ainsi, offrir une inférence en temps réel et en continu impose des contraintes ultra-faibles en matière de consommation d'énergie, avec des implications et des défis techniques et économiques directs et significatifs.

L'intersection de l'apprentissage profond et des systèmes embarqués a donné naissance au domaine du tinyML, qui offre à la fois des opportunités et des défis significatifs. Le principal défi consiste à adapter les techniques d'apprentissage profond pour fonctionner efficacement sur des systèmes MEMS ultra-faible consommation avec des ressources et des opérations limitées, tout en maintenant des niveaux de performance acceptables.

Cette thèse vise à fournir des stratégies pour optimiser l'empreinte énergétique et le déploiement des modèles d'apprentissage profond dans des contextes ultra-faible consommation pour des applications basées sur des MEMS. Nous étudions d'abord les méthodes de compression de modèles, telles que l'élagage, la distillation de connaissance et la quantification. Ensuite, nous permettons le déploiement de bout en bout de modèles d'apprentissage profond pour une inférence efficace sur les MCUs les plus limités en ressources de l'industrie, redéfinissant ainsi le concept de l'inférence ultra-faible consommation en tant qu'extrême-faible consommation. Enfin, nous présentons une nouvelle approche pour généraliser le processus de quantification, compatible avec

n'importe quel nombre de bits et s'étendant à des niveaux de quantification extrêmes, tels que 1-bit.

Les résultats de ces recherches contribuent à l'avancement du domaine tinyML et permettent une adoption plus large des systèmes de détection intelligents embarqués dans diverses applications du monde réel.

Acknowledgements

This thesis would not have been possible without all the people I have met and shaped me along this Ph.D. journey.

I express my gratitude towards my supervisors, Etienne and Julyan for their support in training me to become a good researcher, patience, and most of all for keeping trust in my work throughout the years.

I want to thank my jury: Mathieu Salzmann and Eiman Kanjo for reviewing this manuscript and providing constructive feedback, Valérie Perrier for presiding the jury, and Inna Kucher for evaluating my work.

I want to thank TDK InvenSense, and in particular Bruno, Peter, and Philippe for letting me contribute to deep learning R&D at TDK and for funding this project. I also thank the algo team, Bruno, Daniela, Étienne, Jiali, Rémi, and Soufiane for their collaboration and our brainstorming and interesting discussion. Likewise, I also want to thank IFR, ISJ offices for their support. I can't wait to see the future of deep learning innovations and patents at TDK.

I want to thank the Statifiers (not the Satisfiers) with whom I shared the Ph.D. life struggles: Argheesh for always starting adventures, Alexandre for your remarkable +1 day wisdom, Benoit for the PhD work wisdom, Dasha for your kindness, Florence for being a cool chef and the Statify days, Geoffroy for your sugar and dishwashing addiction, Jacopo for teaching us “la vie d'artiste”, Jean-Baptiste for your interest, Jhouben for the best post-wake up joke, Jonathan for the cool discussions, Julien for being there or maybe not, Hanâ for the PhD journey discussions, Kostas for finding the best AC in Grenoble, Louise for never buying walking poles (yet?), Lucrezia for your practical originality for adventures, Masha for being the first BDE president, mousquetaire and your unique craziness, Meryem for being part of the BDE and mousquetaire, and having the warmest soul, Pascal for your support, Pedro for the constructive comments, Pierre for the always insightful discussions, Sophie for the essential feedback, Stéphane for the witted jokes, Théo for the best clandestine pizza ever tasted, Tin for the encouragements, and Yuchen for your determination. I also want to thank ex-Statify members, a.k.a

Mistis: Brice for your humor and our reassuring discussions, and Karina for the hikes and photoshoot.

Then, I want to thank older and new friends around the globe: Alice and Samuel for the once again Genova adventures sìsì, Augustin for always giving the best tips since the UTC, who motivated me to go for a Ph.D., and the random Bulgarian parties, Henri for appearing out of nowhere to the pot, Nico and Val for the Indo trip, Juan for the largest post-curfew party in Genova, and being the best, Jonas for the Leipzig discoveries and Rob for the best indo insights.

Lastly, I would not be there without my family: my sister, brother, and especially my parents. Con rất biết ơn ba mẹ đã hy sinh cuộc đời này cho chúng con cơ hội có được một cuộc sống tốt đẹp hơn, và hy vọng chúng con không cần phải làm việc vất vả nhiều như ba mẹ ngày xưa, và rất biết ơn ba mẹ đã dạy cho con những giá trị của công việc, sự kiên trì và tôn trọng. Ba mẹ luôn ở đây ủng hộ con, mặc dù không phải lúc nào chúng ta cũng thấu hiểu nhau nhưng con mong ba tự hào về con như là con đang tự hào được là con của ba mẹ. Cảm ơn ba mẹ.

It's not easy to do, but try — try and do what you love with people you love.
And if you can manage that, it's the definition of heaven on earth."
— Conan O'Brien

Table of contents

List of figures xiii

List of tables xvii

1 Introduction 1

1.1 Neural networks 4

1.1.1 Feedforward neural networks 4

1.1.2 Properties 5

1.1.3 Modern deep learning 6

1.1.4 From large deep learning models to tinyML 10

1.2 MEMS-based applications on ultra-low power microcontrollers 13

1.2.1 Overview 13

1.2.2 Scope of applications 14

1.2.3 Challenges of ultra-low power hardware 15

1.3 Efficient neural networks for tinyML 17

1.3.1 Efficient RNNs 17

1.3.2 Overview of model compression techniques 18

1.3.3 Summary 27

1.4 Deployment of deep learning models on ultra-low power MCUs 27

1.4.1 TinyML tools 29

1.5 Summary and contributions 31

2 Methods for design of efficient neural networks 33

2.1 Efficient RNNs 34

2.2 Knowledge distillation 35

2.2.1 MNIST dataset 35

2.2.2 HAR dataset 35

2.3 Model pruning 37

2.3.1 MNIST dataset . . . 38
2.3.2 HAR dataset . . . 38
2.4 Conclusion . . . 40
2.4.1 Discussion and limitations . . . 40
2.4.2 Future work . . . 42

3 TinyML for 8-bit neural networks on ultra-low power MCUs 45
3.1 Introduction . . . 46
3.2 TinyMLOps for ultra-low power MCUs applied to frame-based events . . 47
3.2.1 Introduction . . . 47
3.2.2 TinyMLOps for real-time sensors applied to frame-based event classification . . . 50
3.2.3 Conclusion . . . 54
3.3 ML2MCU: Towards automated large-scale deployment of tiny neural networks . . . 54
3.3.1 Dataset design . . . 55
3.3.2 Model design and training . . . 57
3.3.3 Deployment . . . 58
3.3.4 Evaluation . . . 64
3.4 Results . . . 64
3.5 Discussion . . . 65
3.5.1 Conclusion . . . 65
3.5.2 Limitations . . . 67
3.5.3 Future work . . . 67

Appendix 69
3.A Generalized confusion matrix for multiclass event classifications . . . 69
3.B AudioSample API class documentation . . . 69

4 Regularization for hybrid N-Bit weight quantization of neural networks on ultra-low power microcontrollers 75
4.1 Introduction . . . 77
4.2 Proposed method . . . 79
4.2.1 1-bit regularization . . . 80
4.2.2 N-bit generalization . . . 82
4.2.3 Scheduled regularized quantization . . . 83
4.2.4 Post-training quantization . . . 84

4.3 Experiments . . . 84
4.3.1 Datasets . . . 85
4.3.2 Model . . . 85
4.3.3 Experimental design . . . 86
4.3.4 Deployment and evaluation . . . 87
4.4 Results . . . 88
4.4.1 Regularization configuration . . . 88
4.4.2 Model configuration . . . 90
4.4.3 Key results . . . 91
4.5 Conclusion . . . 93

Appendix **95**
4.A Hybrid {1,8}-bit model byte size reference on the BTS dataset . . . 95
4.A.1 CNN1D only . . . 96
4.A.2 GRU only . . . 98
4.A.3 CNN1D+GRU only . . . 99
4.A.4 All layers . . . 101

5 Discussion **103**
5.1 Conclusion . . . 103
5.2 Future directions . . . 105

References **107**

List of figures

1.1 TinyML as the intersection between artificial intelligence and embedded systems. . . . 3
1.2 Feedforward neural network. . . . 5
1.3 Illustration of memory hierarchies for a mobile processor (left) and an Arm Cortex-M7 microcontroller (right). The microcontrollers process all computation and data transfer on-chip. . . . 15
1.4 Floating-point and fixed-point 32-bit representations. Floating-point allows a dynamic range (minimal to maximal possible value) of roughly $[-10^{38}, 10^{38}]$, compared to fixed-point $[-2^{m-1}, 2^{m-1} - 2^{-n}] \approx [-2^{15}, 2^{16}]$, which is approximately a 10^{33} smaller range (Novac et al., 2021). The smallest resolution (step between each consecutive representable value) of floating-point is $\approx 10^{-38}$ while it is $[2^{-n}] \approx 10^{-5}$ for $n = 16$ for fixed-point. 16
1.5 Unstructured pruning (left panel) versus structured pruning (middle and right panels). . . . 20
1.6 Pruning rate over epochs with a polynomial schedule function (Zhu and Gupta, 2017) with $s_f = 0.8$, $s_0 = 0$, $t_0 = 0$, $n = 13260$, $\Delta t = 100$ (Equation (1.6)). . . . 21
1.7 TinyMLOps pipeline. . . . 28

2.1 Comparison of test results of GRU baseline (Chung et al., 2014) versus MGU (Zhou et al., 2016) versus MGU1, MGU2, MGU3 (Heck and Salem, 2017) over the number of neurons (left) and parameters (right). Each training is repeated 10 independent times on an HAR dataset. . . . 34
2.2 Knowledge distillation on MNIST dataset with CNN models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is 10 times larger than the student model. 36

2.3 Knowledge distillation on HAR dataset with MLP models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is 5 times larger than the student model. 36
2.4 Knowledge distillation on HAR dataset with GRU(80) to GRU(30) models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is about 6 times larger than the student model. 37
2.5 Knowledge distillation on HAR dataset with GRU(80) to RNN(40) models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is about 11 times larger than the student model. 38
2.6 Knowledge distillation on HAR dataset with LSTM(80) to GRU(30) models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is about 8 times larger than the student model. 39
2.7 Magnitude-based (unstructured) pruning with polynomial decay (Zhu and Gupta, 2017) from one base CNN model on MNIST, with two epochs for fine-tuning. 40
2.8 Magnitude-based (unstructured) pruning with polynomial decay (Zhu and Gupta, 2017) on GRU models applied to the HAR dataset. The blue curve is obtained from five independent training of GRU models at different sizes, acting as a baseline. This baseline is the same for all three plots. The red curve is obtained from re-pruning five GRU models (large, medium, and small) for each sparsity rate (5% pruning step rate). 41

3.1 Example of a frame-by-frame event-based classification encoded by binary vectors. We observe four possible misclassifications. 49
3.2 TinyMLOps: Steps and examples of indirect iteration causes. 50
3.3 Custom metrics evaluated by varying the decision threshold and measuring event errors or latency of a GRU model on a four-class head gesture test set. The model is deployed on an Arm Cortex-M0+ MCU. 53
3.4 Illustration of an audio sample, with random samples, position, and a number of positive and negative keywords. 58
3.5 Tradeoff between size and on-device accuracy for a CGRU model applied on the bring-to-see dataset, using ML2MCU's meta-optimizer. Optimal tradeoffs are highlighted in green, suboptimal tradeoffs in red, and manually designed models in black. 59

3.6 Elu versus hard Elu function. A L^1 distance between the original Elu and our approximation ≈ 0.32 for x $\in \mathbb{R}$. 63

3.A.1 Error types of a generalized confusion matrix for multiclass event classifications with a rejection class (i.e., non-event). There is an additional error type compared to binary event classification: misclassification between positive classes. 69

4.2.1 Regularized quantization pipeline using hybrid {1, 2, 4, 8}-bits quantization with schedule option using a convolutional (CNN) GRU model with a fully-connected (FC) layer as example. 80

4.2.2 1-bit regularization functions of Equations (4.1), (4.3), (4.4), and (4.5). . 81

4.2.3 N-bit regularization functions of Equation (4.5) and Equation (4.6), $N \in \{1, 2, 4, 8\}$. 82

4.2.4 Schedule functions for quantized regularization for $t_{\text{final}} = 350$. 83

4.2.5 Scheduled regularization of GRU weights at epoch 1, 25, and 500 (left to right) to 4-bits with Reg_4 regularization and linear schedule in the background, on the BTS dataset. 85

4.4.1 Hybrid {1, 8}-bits quantization test accuracy results on the BTS (left) and GSCv2-12 dataset (right), without schedule (blue) or with schedule (green) (Section 4.2.3). The baseline is trained normally without regularization and quantized to 8-bits. For BTS, we exclude training with schedulers for CNN1D+GRU. For GSCv2-12, we exclude training with CNN1D+GRU. 88

4.4.2 Hybrid {2, 4, 8}-bit quantization test accuracy results on the BTS (left) and GSCv2-12 dataset (right), without schedule (blue) or with schedule (green). The baseline is trained without regularization and quantized to 8-bits. 89

4.4.3 Scheduler comparison of our 1-bit quantization results using our 1-bit (top) and {2, 4, 8}-bits (bottom) regularization, on the BTS (left) and GSCv2-12 (right) dataset, across all scenarios. Schedulers are ordered from fastest to slowest at mid-training time (as shown in Figure 4.2.4). The baseline is trained without regularization and quantized to 8-bits. For {2, 4, 8}-bits, inverse polynomial and step-based bounded are omitted from experiments. 90

4.4.4 Hybrid {1, 8}-bits weight quantization accuracy versus the number of CNN channels and GRU cells for various combinations of quantized layers, with ReLU or PReLU as CNN activation on the BTS dataset. 1st row: All layers, 2nd: CNN1D layer only, 3rd: CNN1D and GRU layers, and 4th: GRU layer. For reference, we also added the baseline model architecture at 6 channels and 5 cells with hybrid quantized layers. Darker colors are better. Each model here is trained using our 1-bit regularization (Section 4.2.1) without a schedule. 92

List of tables

1.1 Summary of standard architectures used in modern deep learning. 9
1.2 Reference table of standard activation functions. 9
1.3 Comparison of representative deep learning model sizes across cloud, mobile, and MCU platforms. 12
1.4 Comparison of hardware for cloud, mobile, and TinyML, including the MCU targeted in this work, provided by TDK InvenSense (Banbury et al., 2021a, Saha et al., 2022). 14
1.5 Example of sensor applications and their target MCU devices. We indicate our focus applications in italics. 15

3.1 Comparison of a model quantized to int8 deployed on an Arm Cortex-M4 MCU using tf.lite (David et al., 2021), NNoM (Ma, 2020) and our tinyMLOps solution on an activity recognition dataset. 52
3.2 Dataset reference for TDK InvenSense's MEMS-based applications. . . . 56
3.3 Benchmark results of 8-bit neural networks deployed on ultra-low power MCUs for motion detection datasets. 65
3.4 Benchmark results of 8-bit neural networks deployed on ultra-low power MCUs for audio applications. 66

4.2.1 Schedule functions for regularization. Here, d is set to 0.01 and 0.001 for schedule inverse linear and inverse polynomial respectively, and rate is set to 0.8 by default. For step-based functions, step is set to 10 by default. For step-based inverse, a higher rate leads to a faster schedule, contrary to step-based bounded. 84
4.3.1 Baseline model configuration size and parameters where (c, k) refers to the number of channels and the kernel size of the CNN1D. Weights are stored in 8-bits and activations in 32-bits integers. 86

4.A.1 Bytes size of the BTS baseline ReLU model with CNN1D weights quantized to 1-bit. 96
4.A.2 Bytes size of the BTS baseline PReLU model with CNN1D weights quantized to 1-bit. 97
4.A.3 Bytes size of the BTS baseline ReLU model with GRU weights quantized to 1-bit. 98
4.A.4 Bytes size of the BTS baseline PReLU model with GRU weights quantized to 1-bit. 98
4.A.5 Bytes size of the BTS baseline ReLU model with CNN1D+GRU weights quantized to 1-bit. 99
4.A.6 Bytes size of the BTS baseline PReLU model with CNN1D+GRU weights quantized to 1-bit. 100
4.A.7 Bytes size of the BTS baseline ReLU model with all weights quantized to 1-bit. 101
4.A.8 Bytes size of the BTS baseline PReLU model with all weights quantized to 1-bit. 101

List of Algorithms

1 High-level data augmentation for robust audio samples for keyword spotting. 57
2 Symmetric quantization of a floating-point matrix to N-bits : NBitQuantize. 61
3 Floating-point fully-connected: FloatFC. 62
4 Integer-only fixed-point fully-connected inference: QuantizedFC. 62
5 Floating-point hard Elu: FloatHardElu. 63
6 Fixed-point hard Elu: QuantizedHardElu. 63

Chapter 1

Introduction

"Everything is both simpler than we can imagine,
and more complicated that we can conceive."
— Johann Wolfgang von Goethe

Artificial intelligence. Over the last decade, *artificial intelligence* (AI) has revolutionized our daily experiences and technological advancements, empowering machines to perform tasks that traditionally require human-like intelligence, such as recognizing objects or speech or playing advanced games like Go.
Machine learning (ML) is the most prominent AI approach, which trains computers to learn patterns and representations from data without explicit programming.
Deep learning (DL) is an advanced subset of machine learning inspired by the organization of the brain, using artificial neural networks (NNs) to model and solve complex problems in a wide variety of fields, including language processing, protein generation, or automation.

Sensors and microcontrollers. Simultaneously, there has been an increase in the adoption and development of the *Internet of Things* (IoT), bringing new devices and applications into our daily lives. *Microelectromechanical sensors* (MEMS) and *microcontroller units* (MCUs) are essential hardware components of IoT, which allows hardware devices to collect and process information (movement, voice, temperature, pressure, ...) directly at the source, in their local environment, excluding the need for additional resources or external communication. Local and autonomous data processing optimizes the flow of information but inherently poses power constraints. Some applications also require

continuous data processing, which put additional power constraints. MEMS and MCUs serve as the interface to sense information between the analog and the digital world. These devices are found in a wide range of applications, including mobiles, cars, wearables, environmental monitoring, and healthcare systems. Their consumer market scales to several billion in annual sales, so a slight deviation in power constraints can result in significant costs.

TinyML. The convergence of machine learning and IoT has sparked significant interest in research and industry because it enables embedded hardware to process local data and interact with their environment in an automated and intelligent way, thus leading to the emerging field of *tinyML* (Figure 1.1). TinyML focuses on developing efficient neural network models and deployment techniques tailored for low-power, resource-constrained devices. Some examples of tinyML applications are detecting or counting events, gesture recognition, predictive maintenance, or keyword spotting, commonly found in home appliances, remote control devices, smartphones, smart watches, or augmented reality glasses.

However, the exponential growth of deep learning is closely linked to the development of powerful hardware, such as graphical processing units (GPUs), capable of supporting its large computation requirements. Therefore, deep learning has yet to reach the same growth and support on low-power devices, such as microcontrollers, to enable deep learning to run at the edge. Indeed, the power footprint of deep learning, as well as the vast landscape of embedded devices, pose new challenges but exciting opportunities that must be addressed by researchers and industrials.

Industrial context. TDK InvenSense is a world-leading manufacturer and supplier of motion, sound, or pressure sensors. Their sensors are integrated into a wide range of applications and products, including smartphones, wearables, cars, home appliances, remote controls, ... In many of these applications, the sensors must operate always-on and in real-time, which requires extreme attention to memory, latency, and power constraints. Meeting these power constraints can help reduce costs and differentiate their products in a highly competitive industry that sells billions of units annually.

Moreover, the promise of deep learning to automatically learn, extract and classify information from labeled data with state-of-the-art performance under a fast time-to-market and uniform design process is highly attractive. Practically, this would allow non-practitioners to create and design new sensor-based algorithms at will. However, while deep learning has proven high performance, it is an expensive method in terms of

data collection, training, and inference, which makes it challenging to deploy on the most constrained devices. Moreover, even a slight increase in power footprint may require the use of an upgraded hardware model, resulting in significant additional costs for consumers and a less competitive market advantage.

Therefore, researching new ways to incorporate the promises of neural networks into the extreme ultra-low power class of embedded targets poses significant challenges and opportunities.

Introduction outline. In the following section, we introduce deep learning, its applications, and limitations (Section 1.1), for low-power devices (Section 1.2) for research and industry. This has led us to explore the challenges and methods (Section 1.3), along with the tools and practices (Section 1.4) used to combine both areas, contributing to the emerging field of tinyML.

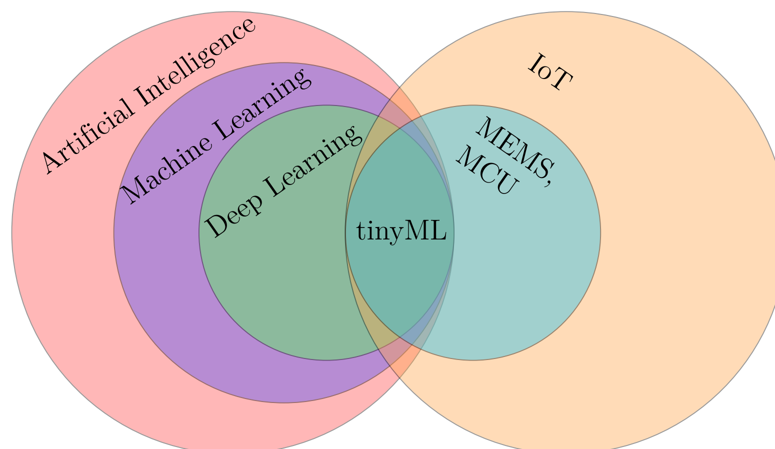


Fig. 1.1 TinyML as the intersection between artificial intelligence and embedded systems.

1.1 Neural networks

We introduce neural networks (Section 1.1.1), then we motivate how their theoretical properties (Section 1.1.2) and modern architectures (Section 1.1.3) are of interests in tinyML, and finally explain its implications for our work (Section 1.1.4).

1.1.1 Feedforward neural networks

The concept of artificial neural networks was introduced by McCulloch and Pitts (1943) as a mathematical model to simulate the human biological neural system but was limited in its ability to learn. This laid the foundation for the perceptron model, which was the first neural model capable of learning and classifying linearly separable data (Rosenblatt, 1958, Sakib et al., 2018). In turn, the backpropagation (Rumelhart et al., 1986) and gradient descent algorithms (Baldi, 1995, Lecun et al., 1998) were developed to allow efficient training of multi-layer perceptron (MLP) that is capable of classifying non-linear inputs. The MLP is a type of feedforward neural network that consists of alternatively stacking multiple layers L of neurons and non-linear functions ϕ (Rumelhart et al., 1986, Huang, 2009) as represented in Figure 1.2. These layers include an input layer, one or more hidden layers, and an output layer. Stochastic gradient descent (SGD) and backpropagation algorithms, and progress in hardware computation have enabled the revolution in the field of neural networks, leading to the modern era of deep learning algorithms (LeCun et al., 2015), for example capable of achieving state-of-the-art performance on ImageNet (Krizhevsky et al., 2012).

Formally, a neural network can be defined as a function f and a directed, weighted graph composed of nodes (neurons) and edges (connections between neurons) with associated weight parameters W, bias B, where inputs x are propagated forward in the graph to produce an output y. The objective of the neural network f defined as

$$\begin{aligned} y &= f(x) = h^{(L)} \\ h^{(l)} &= \phi^{(l)}\left(W^{(l)} h^{(l-1)} + B^{(l)}\right) \quad \text{for } l = 1, \ldots, L \\ h^{(0)} &= x \end{aligned} \tag{1.1}$$

is to approximate some function f^* mapping an input vector x to an output vector y by learning weights matrix W (Goodfellow et al., 2016).

Neural networks have interesting theoretical and practical properties, as we will see in the next sections.

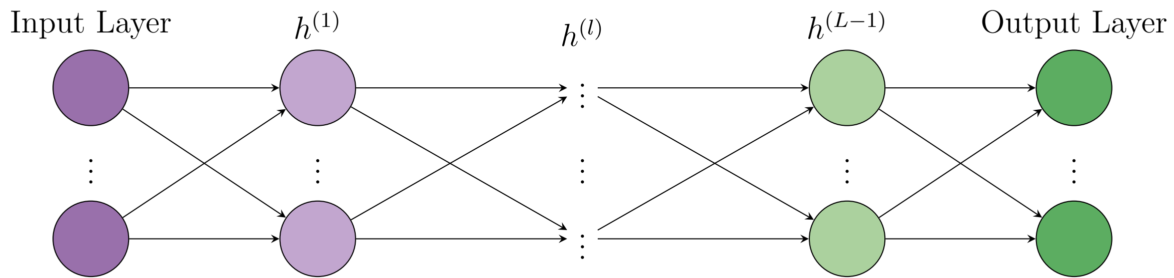


Fig. 1.2 Feedforward neural network.

1.1.2 Properties

Neural networks possess powerful theoretical properties that stand out from standard machine learning approaches, making them of great interest for a wide range of applications.

Expressiveness

Neural networks are universal approximators. Cybenko (1989), Hornik et al. (1989) have theorized that a sufficiently wide hidden layer is able to approximate any continuous function on a compact set to an arbitrary level of precision. More recent work by Lin and Jegelka (2018) has extended the universal approximation theorem to residual neural networks (ResNets) (He et al., 2016), proving that a sufficiently deep neural network with one-neuron hidden layers with residual connections has enough expressive power to approximate any continuous function.

The direct implication is that feature extraction can be done automatically without domain knowledge, unlike standard machine learning. Thus, this allows for a uniform algorithm design process across a wide range of applications, facilitating the creation of algorithms from a set of labeled data, and their use in the industry and research fields.

Although these theorems prove that neural networks are able to learn “by heart” any function, given enough input samples, they do not tell anything about *how* to reach generalization ability to new samples.

Generalization

Neural network models have shown that it is possible to generalize to new data with fewer examples than parameters with very large models (Li and Liang, 2018, Kawaguchi and Huang, 2019), and are even capable of labeling random data (Zhang et al., 2021a). This overparametrization results in a highly-dimensional non-convex space and redundancy, but results in higher quality and quantity of local minima (Choromańska et al., 2014). This implies that the optimization function has a higher chance of not getting stuck

in a bad local minimum compared to small-size networks. These results differentiate neural networks from standard ML models where the overparametrization is usually detrimental to generalization, especially if there are more parameters than needed. Thus, the learnable capacity of neural networks makes them of great interest for performing various tasks in tinyML, using a uniform approach.

We saw an overview of the theoretical properties of deep learning, we will now explore which modern deep learning architectures are commonly used in practice and why.

1.1.3 Modern deep learning

Although in the modern deep learning era, the hardware progress can allow supporting the given high volume of computation and data, the design of the architecture is critical to the final performance and depends on the applications.

Developing and finding new neural network architectures is of great interest in research to surpass the state-of-the-art. Most of these state-of-the-art architectures are variations and combinations of the one we present below. Table 1.1 provides a summary of standard architectures used in modern deep learning, and their strengths and weaknesses.

Fully-connected layers

Fully-connected (FC) layers, also known as dense layers were the first type of layers used in neural networks, specifically in MLP as presented in Section 1.1.3 and depicted in Figure 1.2. They connect each neuron of a layer to all the neurons in the next layer and process each input independently. by applying a non-linear transformation. They are often used toward the end of the model to aggregate the higher-level features from the previous layer and make the final predictions. The simplest form of a fully-connected layer is a weighted sum, which makes them very general and not specialized to any particular application. Thus, they are building blocks of modern deep learning architectures. However, they are prone to overfitting, and may poorly perform on spatial or temporal data.

Convolutional neural networks

Convolutional neural networks (CNNs) are commonly used as feature extractors, showing their strength in processing spatial structures, such as images (Krizhevsky et al., 2012), videos (Simonyan and Zisserman, 2014), or signal processing (Alnaim and Abbod, 2019, Gong and Poellabauer, 2018). As they suggest, they consist of applying convolutional operations using filters, also called kernels on the input in 1D, 2D or 3D, and are shared across the spatial dimensions. They are often stacked all together with max-pooling, to

summarize a group of values by their maximum, (Krizhevsky et al., 2012), batchnorm to normalize activations and facilitate training (Ioffe and Szegedy, 2015), and ReLU activation for non-linearity. Compared to FC layers, this design allows CNNs to efficiently learn spatial hierarchical structures and detect local to global patterns, such as edges, shapes, and textures. In addition, the weight sharing aspect reduces the number of parameters and makes them more robust to spatial translations and distortions. Some classic CNN architectures are AlexNet (Krizhevsky et al., 2012), VGGNet (Simonyan and Zisserman, 2015), or GoogleLeNet (Szegedy et al., 2015), each using increasing network depths, thereby large model size.

Thus, in modern deep learning architectures, CNNs are often found in the early stages of the network serving as powerful feature extractors, but they have shown limitations in learning with sequential data structure or modeling long-range dependencies (Shorten and Khoshgoftaar, 2019, Liu et al., 2020a).

Recurrent neural network

Recurrent neural networks (RNNs) are specialized layers for modeling sequential data (Rumelhart et al., 1986, Elman, 1990), such as signals (Graves and Jaitly, 2014, Alnaim and Abbod, 2019), speech (Zhang et al., 2018) or text (Bahdanau et al., 2015). Compared to CNNs, they are able to model longer temporal contexts by keeping a description of previous contexts because each output directly depends on previous inputs. This is of particular interest for sensor-based applications that inherently deal with sequential data.

The building block of an RNN can be defined as a simple RNN (Elman, 1990):

$$\begin{aligned} h_t &= \phi_h(W_h[h_{t-1}, x_t] + b_h), \\ y_t &= \phi_y(W_y h_t + b_y), \end{aligned} \tag{1.2}$$

where x_t is the input, h_t is a shared internal state, serving as a *memory* at time t, b_h and b_y are bias terms. However, they are difficult to train because of the effects of the vanishing or exploding gradient when the sequence is long (Bengio et al., 1994). Then long-short term memory (LSTM) (Hochreiter and Schmidhuber, 1997, Gers et al., 1999, 2003) and gated-recurrent units (GRU) (Chung et al., 2014) layers were designed to alleviate the limitations of the simple RNN.

They are based on two forms of memory updates:

- "Leak": *Progressive* update of the current memory: $h_{t+1} = h_t + \phi(h_t, x_t)$
- "Gate": *Context-dependent* updates of the memory: $h_{t+1} = \alpha h_t + (1 - \alpha)\phi(h_t, x_t)$,

where α can be a scalar or the output of a gated function $g(h_t, x_t) \in [0, 1]$ as in GRU or LSTM. Note that the "gated" mechanism is a specific form of the attention mechanism (Vaswani et al., 2017), allowing it to focus its attention on specific inputs depending on the context.

In particular, LSTM has three gates (input, forget, and output) and has two hidden temporal streams c_t and h_t where c_t corresponds to h_t in the previous definition of the RNN (Equation (1.2)) and h_t is an auxiliary stream used to compute α thus controlling the quantity of updates.

GRU is a simplified version of LSTM (update and forget) as well as one hidden temporal stream h_t, which has shown performance close to LSTM with a lower power footprint (Cahuantzi et al., 2021).

However, RNNs are limited in handling spatially structured data and processing sequences in parallel. This is because RNNs process input one time step at a time (Equation 1.2).

Residual neural networks

Residual neural networks (ResNets) were introduced in He et al. (2016). They provide each layer with direct feedback from distant previous layers to minimize the loss of gradient information during the backpropagation in deep networks. Although ResNets has shown state-of-the-art performance in computer vision (Khan et al., 2020), they are typically on the scale of millions of parameters (Menghani, 2023) and are more commonly applied on deep networks, which is not suitable for tinyML hardware.

Transformers

Transformers are attention-based models introduced in Vaswani et al. (2017) that surpass state-of-the-art performance on large-scale natural language processing tasks or computer vision tasks (Lin et al., 2022). They allow the model to focus their attention on each token of the input sequence (local) with respect to other tokens (global). This design addresses the limitations of CNNs and RNNs as stated previously because Transformers can process long-term dependencies and sequences in parallel. Although they have encountered great success and interest, they require a large amount of data, and a power footprint for both training and inference, even more than ResNets, which makes them bad candidates for tinyML.

Table 1.1 Summary of standard architectures used in modern deep learning.

Layer	Definition	Strength	Weakness
FC	Connects all neurons in-between layers	High-level aggregations	Overfitting, not specialized
CNN	Convolutional operations with shared parameters	Local and global spatial patterns	Struggles with sequences
RNN	Processes sequences with a hidden state	Temporal dependencies	Struggles with spatial patterns
ResNets	Deep nets with residual connections	Eases training deep networks	Large model size, expensive
Transformers	Self-attention for input relationships	Long-term local and global patterns	Large training data and power footprint

Activation functions

Activation functions in deep learning introduce non-linearity to the model, enabling deep learning models to achieve higher levels of expressiveness and create more complex decision boundaries. This non-linearity is essential for processing real-world data, characterized by diverse and often non-linear features, effectively capturing intricate relationships within the data. Table 1.2 references standard activations used in modern deep learning.

Table 1.2 Reference table of standard activation functions.

Name	Definition	Notes
ReLU	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$	Returns identity if positive, else 0
Leaky ReLU	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{otherwise} \end{cases}$	Allows small negative values
PReLU	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha_i x, & \text{otherwise} \end{cases}$	Per-neuron learnable α_i values
Tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	Returns value in range $[-1, 1]$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	Returns value in range $[0, 1]$
Softmax	$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$	Returns class probabilities

Regularization

In Section 1.1.2, we have seen that neural networks possess interesting generalization properties. We will now explore popular regularization choices that help with generalization in practice.

As in standard machine learning, regularization can help neural networks to generalize better to unseen data, and make them less complex. Regularization techniques can either be of two forms, based on whether or not they directly alter the objective function:

- Explicit:

  - L_1 penalizes the absolute values of the weights, encouraging sparsity, and thus simpler models,
  - L_2 penalizes the squared values of the weights, constraining their magnitude, and thus encourages smoother and simpler models.

- Implicit:
  - Dropout (Srivastava et al., 2014) as an average of probabilistic architectures where each dropout-realization results in a different sub-network (Gal and Ghahramani, 2016),
  - Batch normalization limits the range of values and adds noise to the activation, preventing the model from memorizing the training data too well (Ioffe and Szegedy, 2015, Bjorck et al., 2018),
  - Early-stopping prevents the model from becoming too specialized during training (Sjöberg and Ljung, 1992, Bishop, 1995),
  - Data augmentation increases the size and diversity of the training set, which helps the model learn more robust features (Shorten and Khoshgoftaar, 2019),
  - Random noise injected into the input (also a form of data augmentation) (Goodfellow et al., 2016),
  - Noise introduced by SGD optimization (Poggio et al., 2020a,b).

Most of these regularization methods add negligible computation costs and help with generalization performance.

In this section, we provided a brief overview of the layers used in modern deep learning and which have the most potential for low-power hardware applications.

1.1.4 From large deep learning models to tinyML

In this section, we give an overview of the recent trends of deep learning model sizes, then we explicit the challenges of tinyML based on the neural network theory (Section 1.1.2) and practices (Section 1.1.3), and motivate our interest to apply them for tinyML.

Trend in deep learning models. Since the first AlexNet model was trained on a graphic processor unit (GPU) (Krizhevsky et al., 2012), we entered the modern era of deep learning where the limits of the state-of-the-art are regularly pushed on numerous complex tasks. Meanwhile, deep learning models are geared towards exponential increases in model size. As of 2023, the GPT4 model (OpenAI, 2023) is said to be even larger than

the GPT3 model with 175 billion parameters ($\approx$ 800GB) (Brown et al., 2020), marking a growth of 179900% parameters size in just over 5 years. In general cases, model sizes are at least in the order of 10^7 parameters.

Although model performance can benefit from overparameterization, large neural networks have been shown to have high redundancy (Han et al., 2016, Frankle and Carbin, 2019). Denil et al. (2013) estimated that in some cases only $\approx$ 5% of the total parameters are critical to the final output decision. Thus, we can see that these models fail in terms of *algorithm efficiency*, where the objective is to achieve a task with minimal effort.

This raises questions on how to train more efficient models and also suggests the existence of smaller but viable models.

Trend in efficient deep learning models. A new wave of efficient deep learning models emerged, such as SqueezeNet (Iandola et al., 2016), MobileNet V1, V2, and V3 (Howard et al., 2017, Sandler et al., 2018, Howard et al., 2019), or EfficientNet (Tan and Le, 2019), ranging in one to five million parameters, entering the scale of the feasibility on mobile devices. These new models can achieve up to a 510 time model size reduction compared to AlexNet (Tan and Le, 2019) with equal performance. In general cases, model sizes are in the order of at least 10^6.

Trend in ultra-low power deep learning models. Although mobile-sized models show a great shift toward efficient deep learning architectures, they are still too large for deployment on microcontrollers (Liberis and Lane, 2020, Lin et al., 2020, Banbury et al., 2021a). Deep learning on microcontrollers (Unlu, 2020) is an alternative paradigm that is still at an earlier stage compared to mobile-size research, where the term tinyML has been first appearing in 2019 (Han and Siebert, 2022). However, there has been a success in the deployment of neural networks on MCUs on audio classification tasks (Zhang et al., 2018, Lin et al., 2020, Fedorov et al., 2020) by using efficient CNNs, RNNs, or NAS (Banbury et al., 2021a). In Lin et al. (2020), they succeeded in deploying a person detection model with less than 1MB memory. In general cases, model sizes must be in the order of less than 10^6 and less than 1MB. These models reach a memory size of under 512 kB or even 256 kB, entering the scale of microcontroller hardware. The high resource limitations of MCUs present unique requirements and need the design of dedicated workflow and tools to enable end-to-end deep learning pipelines. Table 1.3 provides a summary of example model sizes for each platform we reviewed.

Motivations. Neural networks are powerful algorithms that can operate with a uniform approach in terms of algorithm design: labeled data, automated feature extraction and

Table 1.3 Comparison of representative deep learning model sizes across cloud, mobile, and MCU platforms.

Platform	Model	Parameters	Model size
Cloud	Inception-v3	$\geq 10^7$	$\geq 10^2$MB
Mobile	MobileNet-v3	10^6	$\geq 10^0$MB
MCU	MCUNet	$< 10^6$	$< 10^0$MB

modeling, and deployment, for a wide range of applications. This makes them a great class of algorithm candidates for MEMS-based applications relying on signal processing.

Unfortunately, the expressiveness and generalization ability of neural networks is dependent on their size, which makes them inherently complex and "black box" functions that are analytically difficult to interpret and design. However, they are mostly composed of very primitive operations (Equation (1.1)): *multiplications and additions*, which are accessible to any microcontrollers. Concerning the non-linear activations, some are very straightforward, such as ReLU (Fukushima, 1975, Nair and Hinton, 2010) or LeakyReLU (Maas et al., 2013), while other activations like tanh or sigmoid pose more challenges due to their computational complexity.

Moreover, prior literature has shown that it is *possible*, albeit *challenging*, to design and deploy small enough neural networks on resource-constrained microcontrollers. Therefore, following the trend of efficient deep learning models to reduce their inherent power footprint, we are interested in pushing the state-of-the-art of low-power footprint models to make them viable to microcontrollers, without degrading performance. Additionally, deep learning models in practice are commonly overparametrized (Denil et al., 2013), so the field of deep learning will benefit from more contributions to designing and deploying more efficient and accessible neural networks.

To summarize, we provided background on neural network theory and practices, their limitations and challenges, and why they are of great research interest for MEMS-based applications running in ultra-low power settings.

Next, we explore the literature on specialized methods to design efficient deep learning models for tinyML in Section 1.3, but we must first provide the necessary background on embedded hardware, which we will reference throughout our work in Section 1.2.

1.2 MEMS-based applications on ultra-low power microcontrollers

We provide a brief overview of MEMS and MCU hardware technology (1.2.1) to understand which specific applications we will focus on in this work (1.2.2) and their intrinsic challenges for deep learning (1.2.3).

1.2.1 Overview

MEMS and MCUs. MEMS are miniaturized (microscale dimensions) sensors and actuators omnipresent in a wide range of electronic devices, as they convert physical and analog information into digital inputs about their local environment (Lammel, 2015, Zhu et al., 2020), that can be processed by MCUs in real-time. Some examples of MEMS are accelerometers, microphones, or pressure sensors. Table 1.5 provides examples of different sensor types and their applications. Thus, they provide an interface to sense real-world information from hardware to software.

MCUs are miniaturized computers that are non-invasive (~1 mm^2 silicon area), cheap (~1$), low-power ($\leq$ 0.5 W), and are dedicated to performing one task for months or even years within a device (Banbury et al., 2021a, Garbay et al., 2022). MCUs are composed of connectors, input/output interface, on-chip storage (ROM), volatile memory (SRAM) for intermediate data, and a CPU with a frequency usually below the 10^3 MHz range (Banbury et al., 2021a). With over 250 billion MCUs already in use, forecasts predict a volume of 38.2 billion in 2023 alone (Lin et al., 2020). In this context, we emphasize that even a small difference in the power footprint between low-power hardware targets can translate to several billions of dollars in savings for the consumer market. This is exemplified by the 2$ difference observed between the low-end of MCUs in Table 1.4. Even between MCUs, there are several orders of magnitude in terms of low-power (Table 1.4). For example, the Cortex-M4 only consumes 0.1W, yet it still represents a target that is 1500 times more power-hungry and 20 times more memory (SRAM) capacity compared to the Cortex-M0+. Additionally, it is three times more costly for consumers. Consequently, it is important to highlight the strong industrial incentive to target the low-cost and low-power consumer market as much as possible with tiny hardware targets. By focusing on the power scale between these targets, we can realize billions in cost savings and other benefits that low-power MCUs offer for the consumer market.

Applicability. Sensing data at the edge allows for offline operations, as opposed to using online cloud computing, always-on and real-time processing, no network latency,

Table 1.4 Comparison of hardware for cloud, mobile, and TinyML, including the MCU targeted in this work, provided by TDK InvenSense (Banbury et al., 2021a, Saha et al., 2022).

Platform	Architecture	Memory	Storage	Frequency	Power	FLOPS	Price
Cloud	GPU	HBM	SSD/Disk				
Nvidia V100S	NVIDIA Volta	32GB	TB~PB	1.2GHz-1.3GHz	250W	~16.4G	14500$
Mobile	CPU	DRAM	Flash				
Galaxy Note 20	Kryo 585	8GB	128GB	1.8GHz-3.1GHz	~8W	1.2T	550$
TinyML	MCU	SRAM	eFlash/ROM				
SAME70Q21B	Cortex-M7	384kB	2048kB	300MHz	0.3W	~432M	5$
SAMG55J19	Cortex-M4	160kB	512kB	120MHz	0.1W	~180M	3$
Newport	Cortex-M0+	8kB	16kB	6.14MHz	70µW	N/A	1$
Newport	eDMPv1	4kB	16kB	6.14MHz	66µW	N/A	1$

limited energy overhead, and inherent privacy. MCUs are ubiquitous in modern electronic devices, including cars, mobiles, TVs, and cameras. Their high volume in the consumer market and wide applicability reinforce the significance of research and industry efforts in tinyML applications.

In this work, we target the most *extreme low-end range* of MCUs, with less than 8kB of RAM and 10MHz processing speed for extreme low-power deep learning inference. Therefore, we aim to push the hardware limit that is currently not considered in the state-of-the-art for embedded deep learning. In particular, we focus on the common ARM Cortex-M series microcontrollers (Yiu, 2019), and particularly the Cortex-M0+ and M4 (Table 1.4), or the eDMPv1 depending on the application. Hardware is provided by TDK InvenSense.

1.2.2 Scope of applications

As previously stated, the ability to embed neural networks at the edge can already benefit a wide variety of applications and can potentially lead to completely new types of products (Kanjo, 2022).

In this work, we will focus on motion detection, gesture recognition, such as human activity recognition (HAR), and keyword spotting. Note that these are all wireless applications, that must operate in real-time and are always-on. In this context, the device returns a decision at all times, so it is expected to provide a seamless user experience (e.g., not missing any user intention (false negatives) or over-triggering (false positives)). Their sensor types and target devices are specified in Table 1.5. Data is provided by TDK InvenSense.

Table 1.5 Example of sensor applications and their target MCU devices. We indicate our focus applications in italics.

Sensor types	Applications	Target devices
Accelerometer, Gyroscope, Magnetometer	*Human activity recognition*, *gesture recognition*, *motion detection*, voice detection, predictive maintenance	Arm Cortex-M0+, eDMPv1
Pressure	Fingerprint detection	Arm Cortex-M0+, M4
Microphone	Sound classification, *keyword spotting*	Arm Cortex-M4, M7

1.2.3 Challenges of ultra-low power hardware

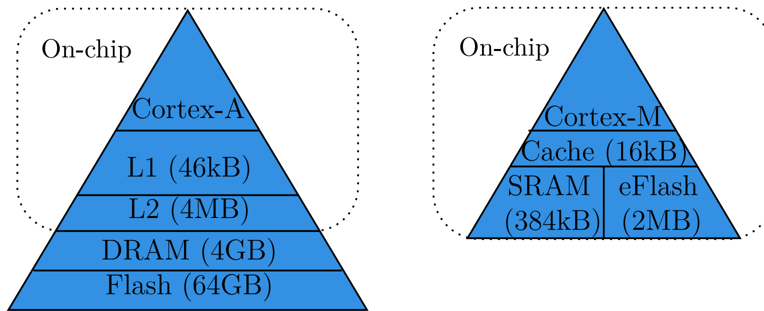


Fig. 1.3 Illustration of memory hierarchies for a mobile processor (left) and an Arm Cortex-M7 microcontroller (right). The microcontrollers process all computation and data transfer on-chip.

Compared to mobile devices, the all-on-chip design, as shown in Figure 1.3, allows the processing of data at the closest location to the source, resulting in lower communication latency and lower power consumption. Thus, this is ideal for real-time and low-power constraints. However, it also makes them inherently constrained because additional memory cannot be extended with an SD card for example.

Moreover, Table 1.4 highlights that the Cortex-M0+ and M4 are among the most resource-constrained devices, with the Cortex-M0+ lacking support for floating-point operations. Consequently, we restrict to fixed-point (in contrast to floating point) values (Figure 1.4) and arithmetic which approximates real-values and computations (Menard et al., 2006), to comply with the inherent hardware and energy constraints of MCUs. Floating-point to fixed-point conversion requires a scaling factor of a power of two, which can be inferred as a simple bit shift and rounding as follows:

$$\begin{aligned} Q(F,n) &= \lfloor F * 2^n \rceil \\ F(Q,n) &= Q * 2^{-n} \end{aligned} \tag{1.3}$$

where Q and F are the fixed point and floating point numbers, respectively, and n is the number of bits. In practice, this means that we are limited to *integer-only operations*. Thus, only primitive operations like bit-manipulation, boolean operators, and basic additions or multiplications are supported in contrast to computationally intensive operations, such as explicit division or exponentiation. Additionally, the memory is typically the first bottleneck, so we seek lower-bit precision parameters than 32-bits, but this may increase the risk of overflow, or numerical precision loss and thus erroneous inference. From a hardware point-of-view, restricting to integer-only inference removes the need for a floating-point unit, which saves silicon area for each embedded chip, and thus billions of dollars of annual savings.

31	30 29 28 27 26 25 24 23	22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sign (1-bit)	exponent (8-bits)	significand (23-bits)

(a) Single precision floating-point 32-bit representation from IEEE754 (Rajaraman, 2016).

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
integer part (m-bits)	fractional part (n-bits)

(b) Fixed-point Q16.16 (Q$m.n$) on 32-bit representation.

Fig. 1.4 Floating-point and fixed-point 32-bit representations. Floating-point allows a dynamic range (minimal to maximal possible value) of roughly $[-10^{38}, 10^{38}]$, compared to fixed-point $[-2^{m-1}, 2^{m-1} - 2^{-n}] \approx [-2^{15}, 2^{16}]$, which is approximately a 10^{33} smaller range (Novac et al., 2021). The smallest resolution (step between each consecutive representable value) of floating-point is $\approx 10^{-38}$ while it is $[2^{-n}] \approx 10^{-5}$ for $n = 16$ for fixed-point.

After a comprehensive review of the literature, the Cortex-M0+ and eDMPv1 appear to be one of the most resource-constrained platforms on which successful implementation of state-of-the-art deep learning has been reported (Zhang et al., 2018, Banbury et al., 2021a, Saha et al., 2022). Zhang et al. (2018) deployed a 70kB keyword spotting application on an Arm Cortex M7, while Banbury et al. (2021a) deployed the same application on an Arm Cortex-M4 with a higher accuracy.

Furthermore, embedded hardware has a very heterogeneous ecosystem because specifications may differ from one manufacturer to another, and even between new series of the same brand, making it challenging to find common tools and approaches that are widely supported.

Therefore, the ultra-low power hardware context presents a unique set of challenges due to their inherent resource limitations. Addressing these challenges pose high research

and industry potential value and can lead to transformative advancements in real-time and low-power applications across numerous domains.

To summarize, in Section 1.1 and Section 1.2 we provided background on neural networks, and low-power sensors and motivated the challenges and objectives of our work. We will now examine the literature on methods (Section 1.3) and tools (Section 1.4) to design and deploy efficient neural networks for MEMS-based applications.

1.3 Efficient neural networks for tinyML

Building upon the concepts and motivations surrounding neural networks and embedded systems introduced in the previous sections, we now turn our attention to their intersection: tinyML.

This emerging field aims to combine the powerful benefits of neural networks with the cost-effectiveness of ultra-low power devices with limited power, memory, and processing capabilities. Given the constraints of tinyML, developing efficient neural network architectures and algorithms is essential. In light of the growing efforts in this area, there is an increasing need for methods that can effectively scale to the most challenging embedded hardware, particularly in the context of MEMS-based applications.

In this section, we explore the methods to train and design efficient neural networks for deployment on MCUs, enabling the deployment of intelligent applications on low-cost devices.

1.3.1 Efficient RNNs

Sensor applications mainly process time-related data continuously, so we are naturally interested in standard RNNs layers, such as RNN (Rumelhart et al., 1986, Elman, 1990), GRU (Chung et al., 2014), LSTM (Hochreiter and Schmidhuber, 1997). Arik et al. (2017), Bhardwaj et al. (2022), Lu et al. (2022) have used convolutional recurrent neural networks (CRNNs) with a GRU or LSTM as the recurrent layer for keyword spotting or motion recognition applications for low-power and real-time inference, which matches our target applications and environment. The CRNN architecture offers strengths both in feature extraction, and time sequence processing, as well as compatible size for our target hardware (Bhardwaj et al., 2022).

In particular, Arik et al. (2017) empirically showed that GRU layers offer better size-performance tradeoff over LSTM in keyword spotting applications, which is our most demanding use case.

Moreover, there have been research efforts to find efficient alternatives to standard RNNs, such as minimal RNN (Chen, 2018), minimal gated unit (MGU) (Zhou et al., 2016), MGU1, MGU2, MGU3 (Heck and Salem, 2017). The MGUs differ from GRU by reusing the gates, removing the bias term or the weight matrix completely, or a combination, detailed as follows:

$$\begin{aligned} \text{MGU}: f_t &= \sigma\left(U_f h_{t-1} + W_f x_t + b_f\right) \\ \text{MGU1}: f_t &= \sigma\left(U_f h_{t-1} + b_f\right) \\ \text{MGU2}: f_t &= \sigma\left(U_f h_{t-1}\right) \\ \text{MGU3}: f_t &= \sigma\left(b_f\right), \end{aligned} \tag{1.4}$$

where f_t is the unique gate of the recurrent unit.

We notice that the MGU1, MGU2, and MGU3 variants do not directly gate the current input x_t, but instead, they indirectly gate the previous input x_{t-1} by gating the previous state h_{t-1}, that has processed the previous input x_{t-1}.

Zhou et al. (2016), Heck and Salem (2017) suggest that these alternatives are competitive with GRU in terms of accuracy with a smaller parameter budget and thus should be more low-power friendly.

Next, we explore the methods that apply directly to models in order to reduce their power footprints.

1.3.2 Overview of model compression techniques

Model compression is a set of methods aiming to address the growing power footprint and costs associated with the deployment of neural networks in terms of size and computation on resource-constrained devices (Neill, 2020, Hoefler et al., 2021), such as MCUs. In this section, we will provide an overview of the most commonly used techniques, which essentially encompass five methods: knowledge distillation, pruning, quantization, weight sharing, and low-rank matrix decomposition (Neill, 2020).

Knowledge distillation

Knowledge distillation is a high-level approach to model compression, first explored in Buciluă et al. (2006) to reduce the model size by learning a small (student) model from an ensemble of models (teacher). Then Hinton et al. (2015) popularized knowledge distillation for neural networks where a small model (student) is trained from the supervision of a larger and overparameterized trained model (teacher) that has learned

"dark knowledge". The idea is to leverage the latent knowledge the large teacher has captured and transfer it to the student during the training process. The loss encompasses both the original student loss (e.g., cross-entropy) and the difference between the teacher and student distribution, expressed as follows:

$$L_{\mathrm{KD}}(x, y) = \alpha L_{\mathrm{S}}(x, y) + (1 - \alpha)\mathrm{D}_{\mathrm{KL}}\left(\mathrm{softmax}\left(\frac{T(x, y)}{\mathrm{temp}}\right), \mathrm{softmax}\left(\frac{S(x, y)}{\mathrm{temp}}\right)\right), \quad (1.5)$$

where L_S is the student loss function, $S(x, y)$ is the output of the student model, $T(x, y)$ is the output of the teacher model, D_{KL} is the Kullback–Leibler (KL) divergence, $\alpha \in [0, 1]$ is a hyperparameter that controls the amount of distillation given by the teacher to the student, and temp is another hyperparameter that softens the probability distributions of the output models.

In practice, we must choose and train one teacher and one student architecture. Hinton et al. (2015) showed promising results across general computer vision tasks and sequential data. However, the disadvantages are that it requires empirical knowledge to find good teacher and student models, as well as additional computations to train the teacher and the forward pass of the teacher during the student's training. Although the design of the teacher would consist of training an overparameterized model, which works well in practice, the student should be the size of our target model. Moreover, we can bypass the additional forward pass of the teacher by storing its output along with the training set.

Therefore, the general framework design of knowledge distillation is flexible for our case and has proven promising performance in a wide range of applications.

Model pruning

While knowledge distillation involves training a new smaller model, pruning focuses on removing less important parts of a model. From a neuroscience perspective, the human brain has a pruning mechanism that removes redundant connections or irrelevant information from past experiences (Walsh, 2013, Neill, 2020). In the case of deep learning models, they are notoriously overparameterized (Section 1.1.2), which provides them with a large degree of freedom. In fact, it has been found that only a small fraction of the total parameters are critical (Denil et al., 2013). Model pruning is a very active research area at the intersection of promoting efficient deep learning and understanding neural network training and generalization ability, where new methods emerge continuously (Alqahtani et al., 2021, Hoefler et al., 2021, Freire et al., 2023).

Han et al. (2016), Ullrich et al. (2017) made a major breakthrough for model compression in the modern deep learning era, where they combined pruning, quantization (Section 1.3.2) and Huffman encoding (Huffman, 2006) to reduce a CNN model by 49 times its size with less than 0.5% accuracy loss on the ImageNet dataset.

Seminal work by Frankle and Carbin (2019), Liu et al. (2019) provided more theoretical understanding; the lottery ticket hypothesis (LTH) states that there exists a sparse subnetwork (winning ticket) that can be trained from scratch with the same initialized weights and reach the performance of the original network (10 times larger). In this view, a large model has a greater chance of containing a good subnetwork. They suggest that the network architecture itself is more critical than keeping the values of the weights in the original trained network. In practice, Frankle and Carbin (2019) requires iterative pruning trials of subnetworks to find the winning ticket, which is computationally expensive. Further work extended the LTH, showing that universal tickets could be reused across other applications (Burkholz et al., 2021, Fischer and Burkholz, 2022). In particular, Ramanujan et al. (2020) generalized the LTH to the strong lottery ticket hypothesis (SLTH) where the subnetwork performs well with the randomly initialized parameters and thus does not require retraining. Additionally, Burkholz et al. (2021) demonstrates that SLTH can also yield universal tickets across other applications. Consequently, the SLTH promises that training deep learning models could be replaced by efficient neural network pruning (Fischer and Burkholz, 2022).

Alternatively, pruning can be seen as a form of neural architecture search (NAS) (Elsken et al., 2019), aiming to find Pareto-optimal architectures (Liu et al., 2019). Moreover, it is also a form of regularization because it reduces the complexity of the model, similar to dropout, but the effect remains permanent.

There are essentially two types of pruning: unstructured and structured pruning, referring to how the pruning is performed in a weight matrix of a model, as illustrated in Figure 1.5.

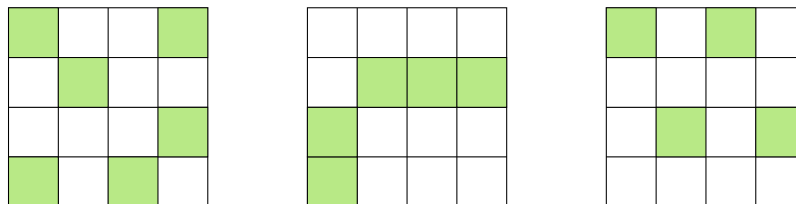


Fig. 1.5 Unstructured pruning (left panel) versus structured pruning (middle and right panels).

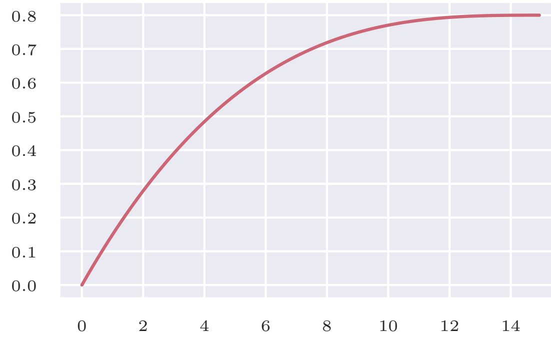


Fig. 1.6 Pruning rate over epochs with a polynomial schedule function (Zhu and Gupta, 2017) with $s_f = 0.8$, $s_0 = 0$, $t_0 = 0$, $n = 13260$, $\Delta t = 100$ (Equation (1.6)).

Unstructured pruning. Unstructured pruning refers to the removal of fine-grained weights in contrast to a group of weights. It is the simplest and most sparsity-inducing type of pruning because trained neural networks are less sensitive to one weight than a specific block.

The most intuitive pruning scheme is to remove weights based on their absolute values, which is the simplest form of magnitude-based pruning, so it does not require any data. This simple approach has been studied early (Hagiwara, 1993) and is very effective (Han et al., 2016, Zhu and Gupta, 2017, Gale et al., 2019, Hoefler et al., 2021). In general, it involves re-training to adapt the model to its new architecture.

While there are a plethora of pruning algorithms, Gale et al. (2019) suggested that magnitude-based pruning provides state-of-the-art or comparable performance to other pruning methods (Thakker et al., 2020, Louizos et al., 2018).

In particular, Zhu and Gupta (2017) introduced a gradual sparsity technique using a polynomial during the training schedule as follows

$$s_t = s_f + (s_0 - s_f)\left(1 - \frac{t - t_0}{n\Delta t}\right)^3, \text{for } t \in \{t_0, t_0 + \Delta t, \ldots, t_0 + n\Delta t\}, \tag{1.6}$$

where s_t and t are the current sparsity and step, s_f is the target sparsity, s_0 and t_0 are the initial sparsity and training step (usually 0), n is the number of pruning steps, and Δt is the pruning step frequency. In other words, at every Δt, a gradual number of weights is set to zero based on their magnitude until we reach the desired sparsity level. The objective of the polynomial schedule is to prune quickly and early when there is the most redundancy, and then slow down the pruning rate as there is little remaining redundancy (Zhu and Gupta, 2017).

Noting that pruning is a form of regularization, Golatkar et al. (2019) found that the early regularization phase is the most critical to performance and that late regularization can even worsen the results, thus supporting the effectiveness of polynomial schedule.

The advantage of magnitude-based pruning is that it is model and task agnostic, can seamlessly incorporate within training, and is easy to implement. Moreover, progressive pruning (Zhu and Gupta, 2017) is natively supported by the TensorFlow framework (Abadi et al., 2015). Additionally, they demonstrate a 90% sparsity rate with acceptable accuracy loss and found that their approach on large-sparse networks performs better than their smaller-dense counterpart. An explanation of this is that larger models are easier to prune because the magnitude of single weights becomes smaller as the model grows larger when the model has converged (Neill, 2020). However, the biggest disadvantage is that unstructured pruning results in sporadically induced weights, which may be difficult to efficiently leverage on embedded hardware, but previous work demonstrated that it is possible to leverage high sparsity with practical encoding (Han et al., 2016).

Structured pruning. Structured pruning alters the architecture of the neural network in blocks, such as neurons, filters, or an entire row or column of a weight matrix. Structured pruning can be induced by using a systematic criterion based on redundancy, as in Srinivas and Babu (2015), where neurons were removed in neural networks by identifying duplicate pairs of neurons, performing a recovery step to compensate for removal. Another common approach is to use regularization penalty to encourage pruning at the channel level in CNN models (He et al., 2017, Liu et al., 2017), by neurons (Alvarez and Salzmann, 2016), or layers (Wen et al., 2016), resulting in models with 60% sparsity without significant loss. The clear advantage of structured pruning is that it is hardware efficient because it may allow skipping entire filters or rows during a matrix multiplication, as suggested in Figure 1.5. Although, block-based pruning techniques have strict compression rules that make them more difficult to achieve without degrading performance and require a certain amount of block sparsity to obtain a faster run time than baseline (Thakker et al., 2020). However, recent research suggests that wider and sparser networks generalize better than their smaller dense counterparts designed by structured pruning (Zhu and Gupta, 2017, Li et al., 2020, Golubeva et al., 2021, Timpl et al., 2021, Ballas, 2022).

In summary, pruning has strong theoretical and practical incentives that make it a high-potential and relevant choice. Unstructured pruning approaches are more flexible across diverse architectures and yield the highest sparsity rate, while structured pruning is more hardware efficient.

Moreover, multiple works have shown that combining pruning with other model compression methods, such as quantization, can produce a high compression rate without significant performance loss (Han et al., 2016, Van Baalen et al., 2020, Zhang et al., 2021b).

Quantization

A different perspective on model compression is quantization. It is a method that maps input values from a larger set (often continuous) to a smaller set (often discrete) (Gholami et al., 2021) to find lossless approximations of numerical input values, and can be seen in related work dating back to the 1800s in the foundations of calculus (e.g., least-squares, approximation of integrals) (Gray and Neuhoff, 1998, Gholami et al., 2021).

In particular, fixed-point attempts to represent continuous values (larger set) with a fixed amount of precision (smaller set); thus, quantization is a mandatory method to meet the low-power requirements of fixed-point arithmetic inference on MCUs, as stated in Section 1.2.3.

Recent work on neural network quantization builds upon prior work but presents unique challenges due to the high power footprint and overparameterized nature of deep learning models. The inherent redundancy in deep learning models allows for some leniency in quantization errors, limiting accuracy loss (Guo, 2018, Gholami et al., 2021). Consequently, very small models, that can be found in tinyML, should be more sensitive to quantization.

Minimizing quantization performance loss can be seen as an optimization problem, where the objective is to find a discrete distribution (quantized weights) that is closest to the original distribution (real weights, activation, or data). In practice, this translates by rounding or truncating the model's parameters (weights, activations) and data from floating points (e.g., 32-bits) to integer values (e.g., 8-bits).

Compared to pruning, quantization often results in less accuracy loss because weights lose precision but are not removed, hence a lower level of information loss (Saha et al., 2022).

Quantization approaches can be characterized by several factors: the stage of the quantization process as quantization-aware training (QAT) or post-training quantization (PTQ), the type of quantization steps as uniform or non-uniform, and the arrangement of quantization levels around the zero-point Z as symmetric or asymmetric (Equation (1.7)).

Quantization-aware training (QAT). QAT involves integrating quantization into the training process or fine-tuning the model by simulating the effects of quantization during the forward or backward pass. However, the quantized function is not differentiable (Equation (1.7)) and can result in zero-gradients in low bit-precisions, making it difficult to train the model. Prior works have quantized values in the forward pass and used real values during the backward pass such as the straight-through estimator (STE) (Bengio et al., 2013, Courbariaux et al., 2015, Jacob et al., 2017), or other approximations (Yin et al., 2018, Louizos et al., 2019). In addition, Choi et al. (2018) learns to optimize the range of activation clipping values and then linearly quantize both weights and activations to 4-bits, while Bhalgat et al. (2020) uses a gradient estimate to learn scaling factors of weights and activations. The objective of QAT is to obtain a stabilized quantized model by the end of training. These methods enable below 8-bit quantization and even down to 1-bit weights or activations (Courbariaux et al., 2015, Rastegari et al., 2016, Hubara et al., 2016, Liu et al., 2018, Qin et al., 2020) with competitive results compared to full precision networks and PTQ. Additionally, AskariHemmat et al. (2022) found that quantization is a form of regularization, where the induced quantization noise can help improve generalization, and particularly to 8-bits on several computer vision tasks. However, QAT often requires a lot of tuning, additional computation, and access to the dataset to re-train the model, especially for low-bit quantization.

Post-training quantization (PTQ). PTQ is the simplest and fastest approach, where quantization can be applied to any trained model without re-training or access to the dataset (Han et al., 2016, Choukroun et al., 2019, Banner et al., 2019, Cai et al., 2020b, Fang et al., 2020). Previous work corrected the mean and variance of quantized weights (Banner et al., 2019, Gholami et al., 2021), or minimized the mean squared error between the quantized and full-precision distributions (Choukroun et al., 2019), allowing 4-bit quantization with acceptable performance. Another approach used piecewise linear functions to partition the quantization range into non-overlapping regions for each weight in order to minimize the quantization error (Fang et al., 2020).

The most widely used quantization method for MCUs is uniform affine PTQ to int8 because it is straightforward and supported by MCUs (Krishnamoorthi, 2018, Gholami et al., 2021, Saha et al., 2022). Moreover, uniform PTQ with int8 provides sufficient performance compared to the original full-precision 32-bits (FP32) model for a wide variety of NNs (Krishnamoorthi, 2018, Lee et al., 2018, Fang et al., 2020). However, PTQ may lead to a more significant loss in accuracy, especially for quantization below 8 bits (Banner et al., 2019, Gholami et al., 2021).

(Non-)uniform quantization. In uniform quantization, the quantization steps are evenly spaced, so it is the most straightforward type of quantization while being natively supported in all embedded hardware (Saha et al., 2022).

In contrast, non-uniform quantization may better capture the original distribution, thus yielding higher accuracy (Gholami et al., 2021). For example, Miyashita et al. (2016), Zhou et al. (2017) uses a logarithmic distribution with exponential quantized steps instead of linear steps. Alternatively, Fu et al. (2020) quantize activations and gradients by finding optimal quantization points that fit their full-precision distributions based on their Weibull prior properties (Vladimirova et al., 2019, 2021), and obtained competitive results compared to the full precision training using less bits than their uniform-based counterpart (Fu et al., 2020).

However, non-uniform quantization schemes are challenging to deploy on embedded hardware because they require a custom implementation to efficiently exploit their specific distribution, in contrast to uniform quantization which is deployable out of the box. Therefore, we restrict the scope of our review to uniform quantization schemes for a wide hardware support.

(A-)symmetric quantization. In symmetric quantization, the lower and upper bounds of the quantization range are equidistant from the zero-point, and $Z = 0$, which simplifies as follows

$$Q(r) = \text{int}(r/S) - Z, \tag{1.7}$$

where Q is the quantization function, r the value to quantize, S a scaling factor, Z represents the zero-point value in the integer discrete space, α, β denote the lower and upper bounds ($\alpha < \beta$), respectively, of the clipping range where we constrain r, and b is the bit-width.

The scaling factor for symmetric and asymmetric quantization is computed as follows:

$$\begin{aligned} S_{\text{sym}} &= \frac{\max(|\alpha|, |\beta|)}{2^{b-1} - 1} \\ S_{\text{asym}} &= \frac{\max(|\alpha|, |\beta|)}{(2^b - 1)/2}, \end{aligned} \tag{1.8}$$

Asymmetric quantization schemes consider the full range of quantized values, e.g., $[-128, +127]$, in contrast to $[-127, +127]$. This provides a slightly larger range to minimize quantization error but is a more complicated implementation due to the zero point $Z \neq 0$ in Equation (1.7), and may lead to more computational overhead (Wu et al., 2020).

In summary, quantization methods have a long history and exist in many flavors to achieve lossless approximations in the most constrained settings. QAT emerges as a superior option in below 8-bit settings, but is more complex and requires more computations than PTQ.

However, uniform PTQ with lower bit quantization is more sensitive due to the distributional properties of weight, which are clustered around zero (Gaussian or Laplacian) (Han et al., 2016, Lin et al., 2016), and few of them are in a long tail (Sub-Weibull) (Vladimirova et al., 2019, 2021). Consequently, uniform quantization maps too few quantization levels to small weights and too many to large ones, leading to performance loss (Fang et al., 2020). However, overparameterized models are less sensitive to PTQ due to having more degrees of freedom (Neill, 2020) in contrast to smaller models.

Thus, we would favor uniform 8-bit PTQ due to its simplicity and acceptable results until we need lower-bit precision for more power footprint reduction.

Weight sharing

Weight sharing is the simplest form of model compression, involving sharing weights values in different parts of the model, so it imposes a model architecture prior to training (Neill, 2020). We could set the amount and location of weight sharing in a strategic way in the model, such as in rows or columns of the weight matrix, for efficient inference. However, manual weight sharing design may be difficult because we cannot predict the final performance, even if redundancy is part of the design of deep learning models.

Prior works have used an automated approach, such as clustering weights with K-means that shares the centroid value among weight clusters with re-training (Wu et al., 2018), where they compressed a CNN model by a factor of three without significant loss, or by using a penalty term to encourage grouping weight (Nowlan and Hinton, 1992, Ullrich et al., 2017).

In particular, quantization is a form of weight sharing because lowering the bit-precision of parameters forces them to be aggregated into a common set of values.

Low-rank matrix and tensor decompositions

Since neural network weight parameters are essentially matrix or tensors, we can apply approximation methods from linear algebra such as single value decomposition (SVD) or its generalization to tensor decomposition (TD) (Neill, 2020). The weight matrix is then replaced by a product of two lower-rank matrices (Xue et al., 2013, Sainath et al., 2013, Novikov et al., 2015, Alvarez and Salzmann, 2017). In particular, Alvarez and Salzmann (2017) obtained a compression rate of up to 96% compared to the original model.

However, these methods require additional hyperparameter tuning (Lebedev et al., 2015), as well as trial and error to find the optimal rank, which may not generalize between applications. Furthermore, for MCUs, it is crucial to consider that the incorporation of additional products from the lower rank matrix may not always lead to increased efficiency and reduced power consumption, so further evaluation of the device is required.

1.3.3 Summary

In summary, we have provided a comprehensive overview of the key methods to design and train efficient tinyML models, accompanied by their related theoretical concepts and practical implications. These methods have generated growing interest, as they bridge the gap between deep learning theory and the deployment of efficient neural networks.

Specifically, model pruning, knowledge distillation, and quantization have demonstrated very promising compression rates, particularly in larger-scaled networks (Mobile or Cloud size) that are more robust to model adjustments. Furthermore, some model compression methods are also forms of regularization that can even help the model to generalize better. Thus, these approaches show high potential to meet the ultra-low power requirements MCUs.

In practice, since tinyML is at an early stage, tools and processes are not mature enough yet to evaluate and truly leverage the high compression rate of existing methods for ultra-low power MCUs, so we will review practical tinyML tools and aspects of the deployment of compressed neural networks in the next section.

1.4 Deployment of deep learning models on ultra-low power MCUs

In this section, we define and review existing tools for the end-to-end deployment of efficient neural networks on ultra-low power MCUs.

TinyMLOps. The first framework for training deep learning models was developed in 2008 (Al-Rfou et al., 2016), with TensorFlow (Abadi et al., 2015) and PyTorch (Paszke et al., 2019) following suit in 2015 and 2016, respectively. These frameworks enabled the large-scale development and deployment of deep learning models, which in turn led to the emergence of Machine Learning Operations (MLOps) (Kreuzberger et al., 2022). MLOps consolidates best practices and outlines steps for mitigating technical debt (Sculley et al., 2015) during the development and deployment of machine learning systems.

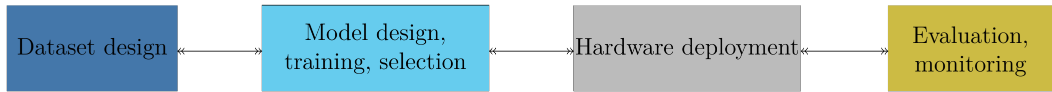


Fig. 1.7 TinyMLOps pipeline.

In contrast, the earliest known publication on tinyML dates back to 2019 (Han and Siebert, 2022), and the first dedicated deep learning framework for microcontrollers, TensorFlow Lite for Microcontrollers (TFLM), was also released in 2019 (Warden and Situnayake, 2020, David et al., 2021). As tinyML gained traction in the industry, MLOps naturally expanded to include tinyMLOps as a subset (Sah et al., 2022, Leroux et al., 2022, Antonini et al., 2022), focusing on refining the process of deploying machine learning on embedded devices, as depicted in Figure 1.7. In the context of tinyML, deployment refers to the process of taking a trained model and enabling it to run on an embedded system, such as compiling the model, firmware integration, and verification of the solution on the target device.

Consequently, the tinyMLOps ecosystem is still in an earlier stage than MLOps, with challenges yet to be fully addressed. In the next sections, we detail the challenges for tinyMLOps tools, and then we review existing solutions.

Challenges. The fundamental characteristic of tinyML is the tight dependency between software and hardware components. In fact, failure to adapt the delivered machine learning software to the constraints of particular hardware renders it unusable, resulting in wasted efforts in previous tinyMLOps steps. Additionally, the diverse landscape of embedded hardware further complicates the task of developing a versatile software base capable of supporting a wide range of embedded hardware platforms (Sah et al., 2022, Leroux et al., 2022), resulting in a manual and iterative approach to the design of new models. As a result, designing new models that work on different hardware remains a manual and iterative approach (different firmware, debugging interfaces, ...). The challenge of tinyMLOps is to improve the entire pipeline, from design to deployment, from data to computation.

Even though tinyML shares some tools with traditional ML (e.g., TensorFlow, PyTorch, Tensorboard), its more recent emergence means that specialized tools are not yet created or are less mature in providing comprehensive solutions. As the tinyML community continues to grow, greater awareness and adoption of tools will lead to faster innovation and the development of comprehensive solutions.

1.4.1 TinyML tools

We restrict tinyML frameworks to the one that supports TensorFlow models as input due to wide adoption within TDK InvenSense, and also targets Arm Cortex-M MCUs for inference.

We essentially consider these two common approaches to tinyML frameworks (Sipola et al., 2022):

1. Using a runtime that loads the model from read-only device memory at runtime (e.g., TensorFlow Lite Micro),
2. Using a transcompiler that converts and compiles models to C or C++ code that then can be built within a project (NNoM, Edge Impulse, µTVM).

Low-level library

CMSIS-NN. CMSIS-NN (Cortex Microcontroller Software Interface Standard for Neural Networks) is a low-level library specifically developed by Arm (Lai et al., 2018) for the Cortex-M microcontroller ecosystem (Table 1.4). It provides a collection of efficient neural network core functions for low-level acceleration. These functions include optimized operations for common neural network operations, such as fully-connected (FC) layers, convolutions, and activation functions (ReLU, sigmoid, tanh, ...). CMSIS-NN has been shown to provide a 4.6x speedup and 4.9x energy savings over non-optimized convolutional models (Lai et al., 2018, Saha et al., 2022).

TinyML frameworks

TensorFlow Lite Micro. TensorFlow Lite Micro (TFLM) is an extension of the TensorFlow ecosystem, specifically designed for deploying neural networks on low-power MCUs such as ARM Cortex-M (David et al., 2021, Warden and Situnayake, 2020). (Sipola et al., 2022, Ray, 2022, Saha et al., 2022). TFLM emphasizes portability by discarding uncommon features, data types, and operations and avoids reliance on specialized libraries or operating systems, thereby achieving memory efficiency and support for a wide range of hardware. It converts and quantizes a 32-bit floating-point TensorFlow model to a compressed flat buffer file (.tflite) using 8-bit integers for weights and 32-bit integers for activations and data. TFLM uses an interpreter-based approach to process the neural network graph at runtime and consists of three primary components: operator resolver, memory stack pre-allocation, and interpreter (Sponner et al., 2021, Schizas et al., 2022). The operator resolver links only essential operations to the model binary file, and the

memory stack is used for initialization and storing runtime variables. The interpreter resolves the network graph at runtime, allocates the memory stack, and performs runtime calculations. More technical details are provided in David et al. (2021), Schizas et al. (2022).

However, TFLM has limitations, such as missing support of some layers or operations (GRU, Conv1D, some important activation functions, ...), arbitrary bit-widths of weights, and activations. Moreover, TFLM lacks target-specific optimizations during compilation because it relies on a graph-level representation that does not include device-specific function kernels and execution details (Sponner et al., 2021, Schizas et al., 2022), and can result in larger memory usage, so it may not meet our extreme memory requirements. Moreover, it does not provide built-in tools to measure power footprint metrics such as inference time or memory usage. Moreover, the interpreter-based approach at runtime makes it difficult to debug and extend, compared to standard compiled code, which hinders research efforts. Despite these limitations, TFLM remains the most popular choice for microcontroller-based deep learning applications.

NNoM. Neural Network on Microcontrollers (NNoM) (Ma, 2020) is an open source project that relies on a C code generation approach with a set of function calls. It is flexible, easy to debug, and supports a wide range of MCUs, but only supports models created using TensorFlow. The project includes a compiler that converts and quantizes a TensorFlow model to plain C code with 8-bit weights and 32-bit activations and data. Additionally, the NNoM compiler supports the CMSIS-NN to generate optimized code for ARM Cortex-M processors (Sipola et al., 2022). It does support all RNN layers including GRU, in contrast to TensorFlow. However, it does not support lower bit-width quantization and has a smaller community and adoption compared to TFLM, so this hinders the development of new features.

Edge Impulse. Edge Impulse (Hymel et al., 2023) is a closed-source cloud service that develops TinyML machine learning models for edge devices and supports AutoML for mobile and microcontrollers (Saha et al., 2022, Ray, 2022). Edge Impulse provides a complete end-to-end model deployment solution, including data collection, feature extraction, training, and deployment (Saha et al., 2022), with an intuitive graphical interface and a friendly no-code approach. The training is carried out in the cloud and the learned model can be exported to an edge device using a data-forwarding capable connection (Schizas et al., 2022).

For model deployment, Edge Impulse uses an interpreter-less edge-optimized neural compiler, which directly compiles the model into C++ source code. This approach eliminates the need to store unused ML operators, resulting in reduced memory requirements at the expense of portability compared to TFLM. Studies have shown that the EON compiler can run the same model with 25%–55% less SRAM and 35% less flash memory than TFLM (Saha et al., 2022).

In conclusion, tinyML brings together the embedded systems and machine learning communities, which have traditionally operated independently. Both academia and industry have developed several software frameworks for TinyML to streamline the deployment of machine learning models on microcontrollers. In particular, we are interested in TFLM because it integrates with TensorFlow and provides a complete toolchain for deploying low-power models MCUs. We are also interested in NNoM because it provides a flexible and simple approach to quantizing and deploying models from plain C code and CMSIS-NN support for Arm Cortex-M MCUs. Moreover, these two frameworks are open-source, which makes them accessible as well as potentially extendable. However, these frameworks are still in the early stages of development, with some missing features and functionality. Despite their limitations, the current first generation of tinyML tools can transition the state-of-the-art machine learning models to ultra-low power environments.

1.5 Summary and contributions

In summary, in Section 1.1 we presented the state of neural networks and motivate our interest in them for our applications, then we provided an overview of MEMS-based applications, emphasized the opportunities and challenges of our extremely low-power constraints, that reinforce the need for more tinyML research efforts in Section 1.2, then in Section 1.3, we presented the existing methods to design efficient neural networks on ultra-low power MCUs, and finally, we provided an overview of existing tools to deploy neural networks to enable for tinyML applications in Section 1.4.

Contributions

Here, we outline the rest of the manuscript and highlight the main contributions of our work.

- Chapter 2 investigates methods for designing efficient neural networks, such as efficient RNNs, knowledge distillation, and pruning.

- Chapter 3 introduces our tinyMLOps solution to deploy 8-bit neural networks on ultra-low power MCUs for MEMS-based applications:

> Lê, M. T. and Arbel, J. (2023). TinyMLOps for real-time ultra-low power MCUs applied to frame-based event classification. In *EuroMLSys '23: Proceedings of the 3rd European Workshop on Machine Learning and Systems, Rome, Italy, May 08-12, 2023*. ACM, New York, NY.
>
> Ponçot, R., Lê, M. T., de Foras, E., Ataya, A., and Hartwell, P. G. (2022). Method for improved keyword spotting. *Pending patent.*

- Chapter 4 presents a novel approach to generalize the quantization to N-bits, and extend to extreme low-precision levels, such as 1-bit:

> Lê, M. T. and de Foras, E. (2022). Towards Universal 1-Bit Weight Quantization of Neural Networks with end-to-end deployment on ultra-low power sensors. *tinyML EMEA Innovation Forum 2022.*
>
> Lê, M. T., de Foras, E., and Arbel, J. (2023). Regularization for hybrid N-bit weight quantization of neural networks on ultra-low power microcontrollers. In Iliadis, L., Papaleonidas, A., Angelov, P., and Jayne, C., editors, *Artificial Neural Networks and Machine Learning – ICANN 2023*, Cham. Springer Nature Switzerland.
>
> de Foras, E. and Lê, M. T. (2022). One bit quantization for embedded systems. *Pending patent.*

- Chapter 5 discusses the significance and contributions of this thesis.

Chapter 2

Methods for design of efficient neural networks

Abstract

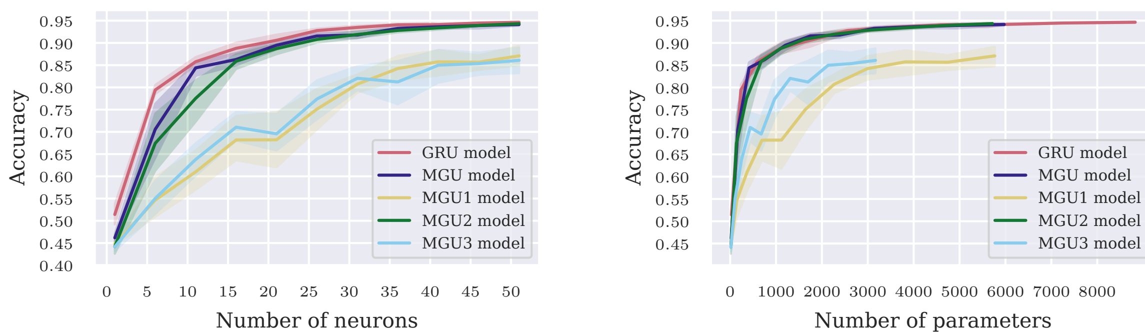
In this chapter, we focus on methods to design efficient neural networks to make them as small as possible before on-device deployment (Chapter 3). In particular, we explore efficient alternatives of RNN layers (Section 2.1) for MEMS-based applications, then we experiment with the higher-level compression: knowledge distillation (Section 2.2), then we investigate model pruning (Section 2.3), and finally we present results, limitations, and directions for future work (Section 2.4).

2.1 Efficient RNNs

Our applications are sequential signals from sensors, in rather short context, compared to full sentences or paragraphs in NLP, so we choose to focus on RNNs layers that can capture time-dependent patterns with moderate size. In particular, we found that GRU-based models have a good size-performance tradeoff, as stated in Section 1.1.3. So we chose to experiment with GRU alternatives, like MGU (Zhou et al., 2016), and MGUs alternatives (Heck and Salem, 2017).

We implemented and compared the size with the performance between GRU and MGU in Figure 2.1 on an HAR dataset. The results show that MGU and MGU2 perform close to GRU above 35 neurons for the same number of neurons. If we look at the corresponding parameter budget, MGU and MGU2 perform similarly to GRU from around 600 to 700 parameters. However, MGU1 performs as poorly as MGU3, while it has more parameters and complexity than MGU2 (Equation 1.4). The results obtained by MGU, MGU2, and MGU3 are on par with the original paper (Heck and Salem, 2017) on the MNIST and Reuters Newswire Topics (text classification) datasets. However, we obtained contrasting results with MGU1, which is surprising because MGU1 is a superset of MGU2, as it only has an additional bias term (Equation (1.4)).

Therefore, MGU2 appears to be the most efficient gated RNN. This suggests that (i) the forget and update gate of the GRU layer can share the same weights without performance loss, and (ii) the gate can omit to process the current input values if the bias of the gate is also removed. However, the same parameter budget as GRU is needed to obtain the same accuracy in MGU and MGU2.



(a) Accuracy over the number of neurons in GRU-based models.

(b) Accuracy over the number of parameters in GRU-based models.

Fig. 2.1 Comparison of test results of GRU baseline (Chung et al., 2014) versus MGU (Zhou et al., 2016) versus MGU1, MGU2, MGU3 (Heck and Salem, 2017) over the number of neurons (left) and parameters (right). Each training is repeated 10 independent times on an HAR dataset.

Next, we experimented with popular model compression methods, such as knowledge distillation and model pruning.

2.2 Knowledge distillation

Knowledge distillation is a high-level method to make smaller models learn from larger models (Section 1.3.2). We started to experiment with the knowledge distillation algorithm (Hinton et al., 2015) presented in Section 1.3.2, which has popularized distillation algorithms. They showed promising results across general computer vision tasks and sequential datasets.

We first experimented with the MNIST dataset (Section 2.2.1), and then with the HAR dataset (Section 2.2.2). We trained a teacher and a student of the same architecture: MLP to MLP (Figure 2.3), GRU to GRU (Figure 2.4), or between different architectures: LSTM to GRU (Figure 2.6), GRU to simple RNN (Figure 2.5).

For each experiment, we vary the temperature and α hyperparameter to tune the influence of the teacher on the student model. We also repeat each experiment independently five times.

2.2.1 MNIST dataset

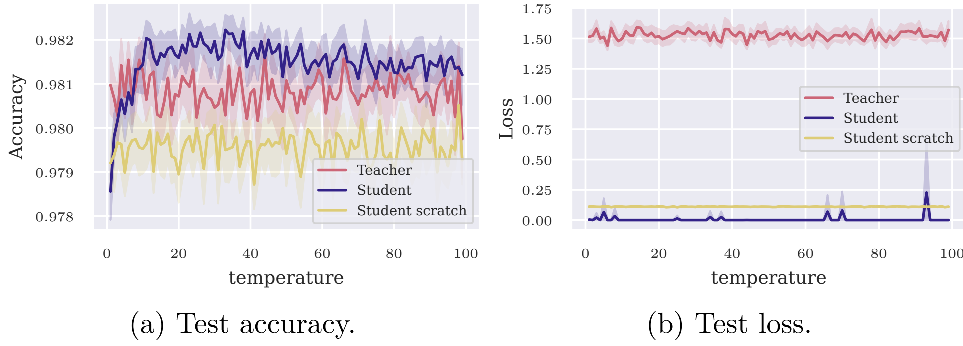
Figure 2.2 shows that the student model, which is 10 times smaller than the teacher, consistently outperforms the student model trained from scratch and even the teacher. A temperature of at least 10 to 40 would be recommended, while the best trade-off would be between 25 and 40.

The outcome is on par with the results obtained in Hinton et al. (2015), so the results are promising.

2.2.2 HAR dataset

Knowledge distillation with the same architectures

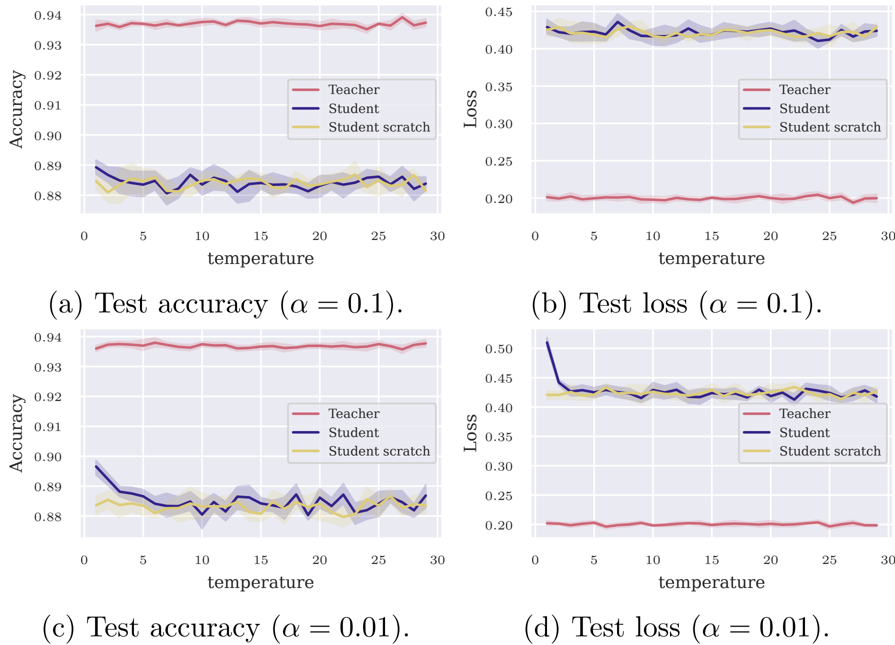
Figure 2.3 illustrates that MLP to MLP knowledge distillation yields a 1% accuracy gain when the teacher's influence increases to 99% of its distilled knowledge at lower temperature settings. In this case, we note that MLP students have a lower accuracy variance than a model trained from scratch. In other cases, distillation does not improve or degrade the student's performance. Similarly, GRU to GRU distillation (Figure 2.4) does not enhance the accuracy of the student, even with the high influence of the teacher.



(a) Test accuracy.

(b) Test loss.

Fig. 2.2 Knowledge distillation on MNIST dataset with CNN models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is 10 times larger than the student model.



(a) Test accuracy ($\alpha = 0.1$).

(b) Test loss ($\alpha = 0.1$).

(c) Test accuracy ($\alpha = 0.01$).

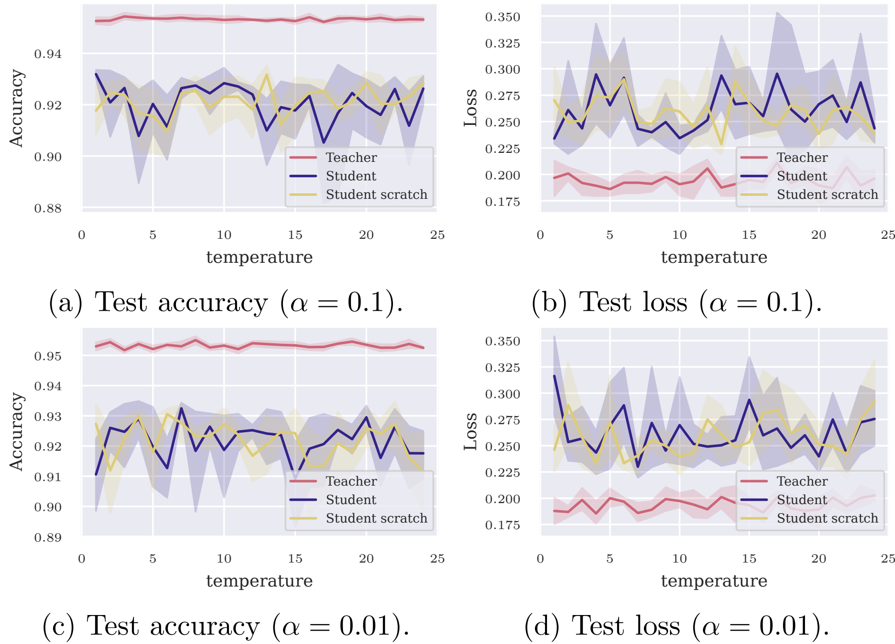
(d) Test loss ($\alpha = 0.01$).

Fig. 2.3 Knowledge distillation on HAR dataset with MLP models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is 5 times larger than the student model.

Different architectures

We also experimented between different types of models to try to reduce the size of the model with the distillation of LSTM to GRU (Figure 2.6) and GRU to RNN (Figure 2.5) distillation.

GRU to RNN and LSTM to GRU distillation do not improve the results compared to the student trained from scratch. The results on the RNN student may be due to the



(a) Test accuracy ($\alpha = 0.1$). (b) Test loss ($\alpha = 0.1$).

(c) Test accuracy ($\alpha = 0.01$). (d) Test loss ($\alpha = 0.01$).

Fig. 2.4 Knowledge distillation on HAR dataset with GRU(80) to GRU(30) models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is about 6 times larger than the student model.

larger size ratio between the student and the teacher, or to the more limited capacity of the RNN to process longer input sequences. The GRU student trained from scratch in Figure 2.6a loses robustness using a temperature between 10 and 13, overwise it still performs close to the other student model.

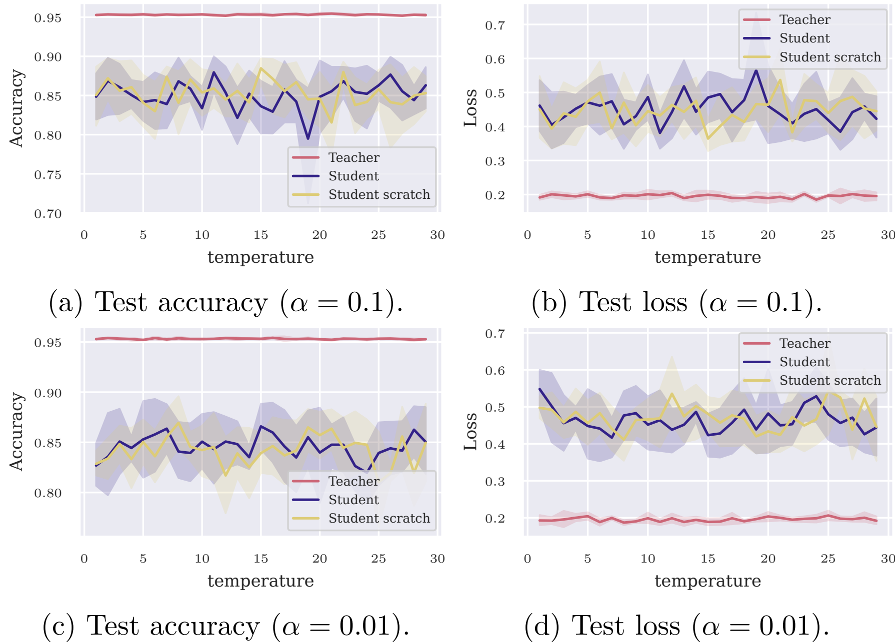
In summary, we succeeded in reproducing positive compression results on MNIST but obtained mixed results with this approach on the HAR dataset. This could be because RNNs need a more specialized distillation approach than MLP or CNNs, where MLP distillation showed some improvements with the HAR application.

In the next section, we present our experiments with model pruning.

2.3 Model pruning

Network pruning is a very common technique to reduce the number of parameters of a model. It has been shown to be able to significantly reduce the size of large networks with minimal performance losses (Section 1.3.2).

We chose to experiment with unstructured pruning, with a polynomial sparsity schedule (Zhu and Gupta, 2017). For the datasets, we continued experiments on MNIST and the HAR dataset. The models are trained to convergence, and then



(a) Test accuracy ($\alpha = 0.1$). (b) Test loss ($\alpha = 0.1$).

(c) Test accuracy ($\alpha = 0.01$). (d) Test loss ($\alpha = 0.01$).

Fig. 2.5 Knowledge distillation on HAR dataset with GRU(80) to RNN(40) models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is about 11 times larger than the student model.

they are progressively pruned and fine-tuned for two and 10 epochs on MNIST and HAR, respectively.

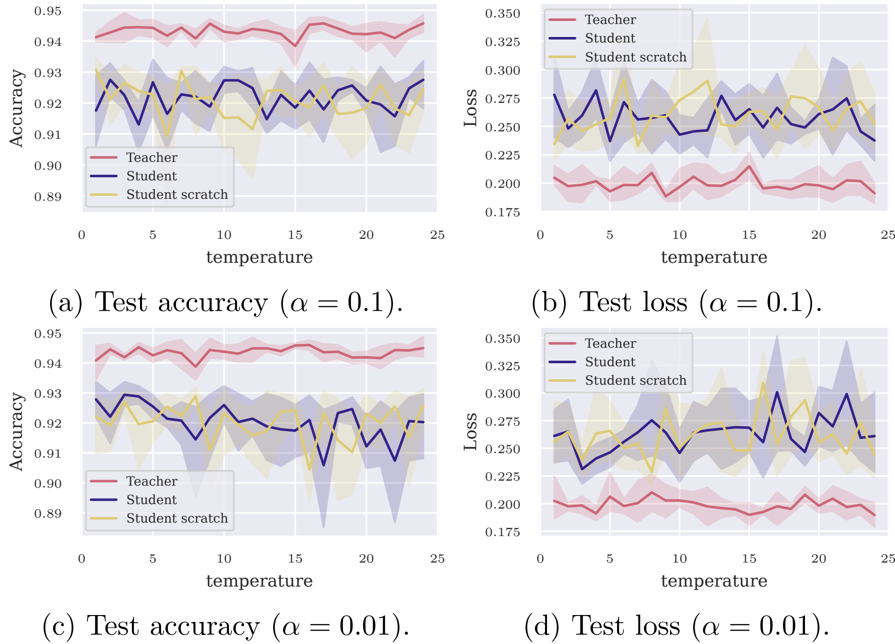
2.3.1 MNIST dataset

Figure 2.8 shows that we can easily reach 80-82% of sparsity and 3-3.5x smaller model (in bytes) without significant performance losses, which is on par with Zhu and Gupta (2017) and our overview of the pruning literature (Section 2.3).

2.3.2 HAR dataset

Similarly to the previous experiments on efficient RNNs, and knowledge distillation, where results were promising, we want to compare the effectiveness of pruning against compact models trained from scratch.

In order to do so, we train multiple GRU baseline models over different numbers of neurons from scratch (in five neuron steps). Then, we train additional large (120 neurons), medium (80 neurons), and small (40 neurons) GRU models and prune them to different levels of sparsity in 5% steps, and then restart the pruning process for each sparsity level.



(a) Test accuracy ($\alpha = 0.1$).

(b) Test loss ($\alpha = 0.1$).

(c) Test accuracy ($\alpha = 0.01$).

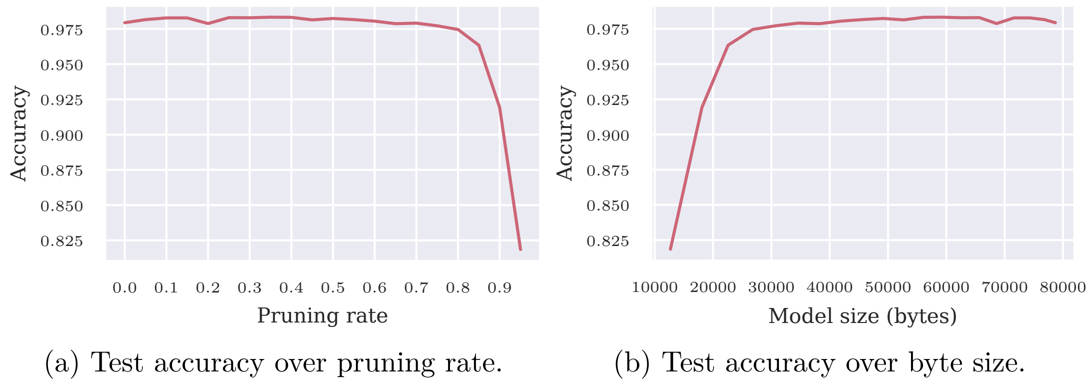
(d) Test loss ($\alpha = 0.01$).

Fig. 2.6 Knowledge distillation on HAR dataset with LSTM(80) to GRU(30) models. Comparing student models trained with the teacher, teacher, and student models trained from scratch. The teacher is about 8 times larger than the student model.

In Figure 2.8, we compare the accuracy of baseline models trained from scratch against pruned models with 40, 80, and 120 neurons in the GRU layer. We observe that pruning demonstrates the potential to reduce up to 80-82% parameters without incurring a significant loss in accuracy, such as previously in the MNIST dataset, highlighting the existence of a Pareto-optimal frontier. We notice a higher variance in smaller GRU models trained from scratch than larger ones, which corresponds to a smaller quantity of optimal local minima (as discussed in Section 1.1.2). Otherwise, the variance between all models remains minimal.

In contrast, the pruned GRU(40) model exhibits enhanced robustness compared to the baseline within the range of 1500 to 2500 parameters because it already started from a favorable local minimum rather than from random initialization. In this case, the pruned GRU(40) models perform similarly to the baseline above 3000 parameters. However, its performance falls below that of the baseline when operating with fewer than 1000 parameters.

The pruned GRU(80) models perform slightly better than the baseline above 1000 parameters with remarkably low variance, also related to the larger number of favorable local minima.



(a) Test accuracy over pruning rate.

(b) Test accuracy over byte size.

Fig. 2.7 Magnitude-based (unstructured) pruning with polynomial decay (Zhu and Gupta, 2017) from one base CNN model on MNIST, with two epochs for fine-tuning.

Furthermore, pruned GRU(120) underperforms compared to baseline when operating with less than 9000 parameters. Above this threshold, it can be hypothesized that both models would have comparable performance, with minimal variance, due to the abundance of optimal local minima.

2.4 Conclusion

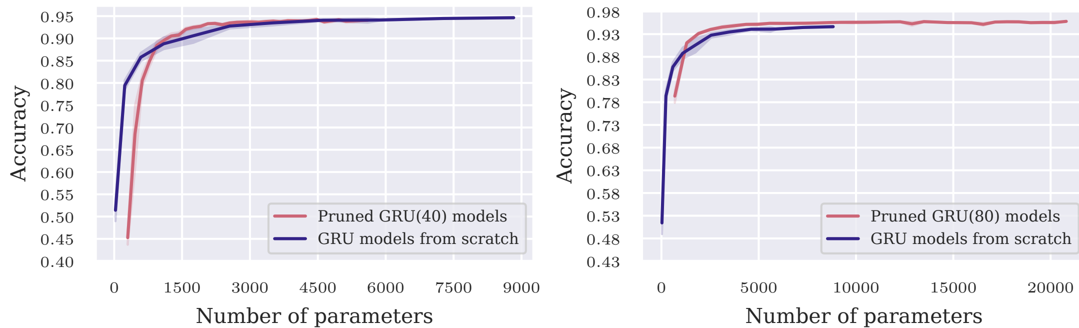
2.4.1 Discussion and limitations

In this chapter, we explored popular methods on both standard and sensor-based datasets to try to improve the efficiency of deep learning architectures. These methods are either model-agnostic or can be applied to a wide range of sequence-based applications. We found some cases where we can find smaller models than their baseline without performance loss, particularly when using MLP models and knowledge distillation, and pruning smaller GRU models to a certain extent.

Despite the promising results found in the literature on MGUs, knowledge distillation, and pruning, we also found that the obtained improvement did not align with our expectations of efficient neural network design.

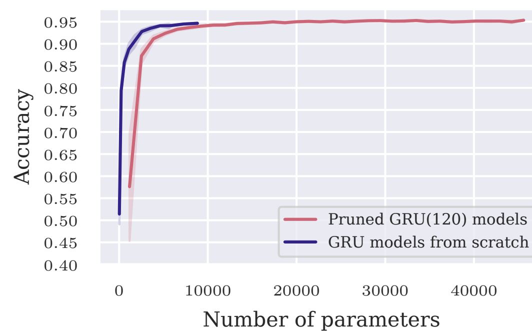
Efficient RNNs

Our experiments on HAR suggest that we could use GRU, MGU, or MGU2 interchangeably above 35 neurons but would not decrease the overall memory footprint given a target accuracy. We also found surprising results where MGU1 performed as poorly as the lightweight MGU3, whereas it has theoretically more expressiveness capacity.



(a) Test accuracy with pruned GRU(40) model.

(b) Test accuracy with pruned GRU(80) model.



(c) Test accuracy with pruned GRU(120) model.

Fig. 2.8 Magnitude-based (unstructured) pruning with polynomial decay (Zhu and Gupta, 2017) on GRU models applied to the HAR dataset. The blue curve is obtained from five independent training of GRU models at different sizes, acting as a baseline. This baseline is the same for all three plots. The red curve is obtained from re-pruning five GRU models (large, medium, and small) for each sparsity rate (5% pruning step rate).

Overall, this means that MGUs would not improve memory or computation footprint, so we decided not to follow this lead because it would not give a significant inference advantage on the embedded sensor in our case.

Knowledge distillation

Overall, we found that distillation in RNN models did not improve or degrade performance, unlike CNNs or MLP-based models, where we successfully found a smaller student in certain settings. This may suggest that RNNs need a more specialized distillation approach than MLP or CNNs.

Previous work found that using a teacher assistant as an intermediate-sized model between the teacher and the student network improves its performance over vanilla knowledge distillation (Mirzadeh et al., 2020, Ozerov and Duong, 2021, Mazlan et al.,

2022). Mirzadeh et al. (2020), Cho and Hariharan (2019) suggested that the student's performance can deteriorate when the knowledge gap between the teacher and student is too large to learn from, so a given student has limited capacity to learn from a teacher whose knowledge exceeds their individual learning threshold.

However, this teacher-assistant approach is not straightforward and requires additional training and iterations.

Pruning

For smaller models, our findings suggest that training from smaller and compact models under 1000 parameters from scratch would yield better results than pruning larger models, contradicting the findings of Zhu and Gupta (2017). This may be because compact networks are more robust than large sparse networks when the model is underfitting scenarios. In the case of small and medium models, the performance of pruned models is generally more robust and slightly superior to models trained from scratch, given the same parameter budget, so we could use pruning as a design strategy to find smaller models with a given parameter budget.

However, it raises the question of how embedded MCUs can exploit such sparsity efficiently for inference (without storing the zero values). Common approaches for sparse computations are look-up tables (table that maps input values to precomputed output values) or specialized hardware accelerators. Moreover, the accuracy gap between the pruned GRU models and the baseline is not substantial enough for future applications.

Finally, we have not tested these methods on more datasets than MNIST and HAR, so our results may not generalize to other applications, and further investigation could reveal additional insights.

2.4.2 Future work

We propose several directions for future research based on our results and limitations.

One potential direction concerns testing efficient RNNs on a wider variety of datasets to verify if our findings generalize to other sequence-based datasets.

Future work could focus on exploring specialized knowledge distillation techniques for RNN-based models, employing a finer level of knowledge distillation, such as layer-wise or group-wise distillation approaches, and considering multi-teacher strategies (Liu et al., 2020b) or teacher assistant for larger model size ratios (Mirzadeh et al., 2020, Mazlan et al., 2022). Moreover, we could also investigate the combination of distillation with other model compression methods, such as quantization (Qu et al., 2020).

A potential lead for pruning would be to investigate hardware-friendly sparsity methods such as structured pruning or hardware-aware pruning (Hawks et al., 2021, Shen et al., 2022). Structured pruning could provide a valuable tool for guiding compact model architecture design. Moreover, developing hardware capable of exploiting unstructured sparsity could really accelerate on-device inference.

Finally, our study did not experiment with other compression methods, such as weight sharing (Section 1.3.2) and tensor decomposition (Section 1.3.2), which could be combined with other approaches would require further investigation.

In this chapter, we focused on high-level methods to improve the algorithm efficiency of deep learning models. In the following chapters, we investigate the methods and process of deploying tinyML models on ultra-low power MCU through quantization.

Chapter 3

TinyML for 8-bit neural networks on ultra-low power MCUs

Abstract

In this chapter, we comprehensively examine the design and deployment of deep learning models on ultra-low power MCUs for MEMS-based applications. In particular, we first provide an overview of the challenges and solutions of the end-to-end tinyMLOps pipeline. We argue that our use case presents unique challenges and solutions that differentiate it from previous processes, as it requires overcoming state-of-the-art constraints through careful consideration and joint efforts from the machine learning and hardware communities in research and industry. Overcoming the challenges of tinyMLOps promises to automatize algorithm design for embedded MEMS-based applications and to democratize deep learning solutions at a large scale on low-cost devices.

Then, we present our own tinyMLOps solution that can train, quantize, and deploy efficient neural networks in 8-bit ultra-low power settings. Our contributions shed light on the deployment process and establish solid ground for the automation of large-scale deployment of tiny neural networks in the most constrained power settings for industrial applications. As a result, we effectively redefine the concept of ultra-low power inference to extreme low-power levels, pushing the boundaries of efficiency and performance in tinyML.

This chapter is based on a publication (Lê and Arbel, 2023) and pending patent (Ponçot et al., 2022):

Lê, M. T. and Arbel, J. (2023). TinyMLOps for real-time ultra-low power MCUs applied to frame-based event classification. In *EuroMLSys '23: Proceedings of the 3rd European Workshop on Machine Learning and Systems, Rome, Italy, May 08-12, 2023*. ACM, New York, NY.

Ponçot, R., Lê, M. T., de Foras, E., Ataya, A., and Hartwell, P. G. (2022). Method for improved keyword spotting. *Pending patent.*

3.1 Introduction

Over the course of the modern deep learning era, practitioners have developed and refined the process of designing and deploying neural networks for large-scale applications in high-end clusters through MLOps. Following this, the emergence of tinyML requires a novel approach to MLOps that incorporates specific considerations such as memory and power limitations, real-time inference, and on-device processing.

Furthermore, on the consumer end, sensor data is the interface between the physical and digital worlds. It enables efficient, always-on, low-cost processing of these data in real-time and directly at the source, presenting a multitude of high-potential applications. Furthermore, on the practitioner's end, the promise of deep learning to design efficient algorithms at will for these smart applications under said constraints is highly attractive.

In this section, we highlight our contributions that address the challenges of tinyML and enable the potential of smart and efficient MEMS-based applications and algorithm design. The contributions are listed as follows:

- an overview of the challenges and solutions of tinyMLOps, applied to event-based classification,
- a high-level data augmentation API for robust keyword spotting,
- a novel, and comprehensive tinyMLOps framework for 8-bit neural networks on ultra-low power microcontrollers for a wide range of MEMS-based application
- automation of the algorithm design process from end-to-end under state-of-the-art hardware constraints, facilitating wide adoption and deployment of efficient smart sensors continuously interfacing with their surrounding environment in real-time.

3.2 TinyMLOps for real-time ultra-low power MCUs applied to frame-based event classification

TinyML applications such as speech recognition, motion detection, or anomaly detection are attracting many industries and researchers because of their innovative and cost-effective potential. Since tinyMLOps is at an even earlier stage than MLOps, the best practices and tools of tinyML are yet to be found to deliver seamless production-ready applications. TinyMLOps has common challenges with MLOps, but it differs from it because of its hard footprint constraints. In this work, we analyze the steps of successful tinyMLOps with a highlight on challenges and solutions in the case of real-time frame-based event classification on low-power devices. We also report a comparative result of our tinyMLOps solution against tf.lite and NNoM.

3.2.1 Introduction

TinyML. The increased adoption of the Internet of Things (IoT) and machine learning (ML) has led to the emergence of tinyML (Han and Siebert, 2022, Doyu et al., 2021), in response to the strong demand to enable advanced applications running on embedded systems. Opening ML to a smaller scale of devices such as wearables or low-power microcontrollers (MCUs) at the tiniest scale, offers new opportunities for various applications including object detection, keyword spotting, face or activity recognition, and predictive maintenance.

However, as tinyML is in its early and rapidly-evolving stage, established best practices are still lacking for designing efficient and accurate tinyML solutions. This gap is hindering the widespread adoption and development of tinyML across the research and industry communities.

TinyMLOps. Similar to MLOps, tinyMLOps aims to automate production-ready applications by utilizing the most efficient tools and practices, from data to model inference on an embedded platform. Therefore, deploying and monitoring tinyML systems face many common objectives and challenges as standard machine learning such as: interfacing each step of the workflow (e.g., data format and model input, model training and delivery), model versioning, integration tests of data, model logging, and codes (Kreuzberger et al., 2022, Antonini et al., 2022).

However, tinyMLOps essentially differs from MLOps in the following aspects (Shafique et al., 2021):

- Models make inferences on very resource-constrained devices (e.g., FPGA or MCU) with limited operations (integer-only, no explicit division, small buffer, and slower clock) and memory as low as 32 KB storage, and up to 48 MHz processor frequency. However, some critical applications require low latency and/or high accuracy.
- Some devices, such as ultra-low power MCUs, may only operate off-grid and may be expected to run autonomously for months or even years at the 1 µW scale. Thus, power consumption, latency, and model size are critical metrics rather than raw accuracy.
- Model deployment must support very heterogeneous and specific hardware platforms, in contrast to standard Linux GPU servers. Additionally, monitoring and updating models after edge deployment is difficult.
- TinyML enables high privacy and responsiveness without the cost of communicating with another device.
- Industry and research communities face new challenges in creating meaningful benchmarks due to platform heterogeneity, but there is a growing effort towards finding a common ground (Sudharsan et al., 2021, Banbury et al., 2021b).
- Most tinyML tools are not mature enough yet because of the field's novelty, so they may not include essential features or be in the alpha stage.

Model compression. Recent works focus on model compression methods to reduce model inference costs (Cheng et al., 2020, Hoefler et al., 2021). These methods include weight sharing (Wu et al., 2018), model pruning (Alvarez and Salzmann, 2016, Zhu and Gupta, 2017, Frankle and Carbin, 2019), which can remove redundant weights up to 90% (Han et al., 2015) with decent results, knowledge distillation (Hinton et al., 2015), which learns smaller models from a large teacher model, neural architecture search (NAS) (Cai et al., 2020a, Liberis et al., 2021, Banbury et al., 2021a), that automates model design with user-defined constraints, and quantization (Han et al., 2016, Gholami et al., 2021), which can reduce bit-precision as low as 1-bit with acceptable performance (Tang et al., 2017, Tu et al., 2022) for computation and storage efficiency, and is a mandatory step for edge inference. In general cases, full 8-bit integer quantization is used, as it does not degrade accuracy at inference time (Jacob et al., 2017) and is natively supported by MCU platforms.

Frame-based event classification. We consider the case of frame-based event classification where each input generates one output (many-to-many), that is coming continuously at some frequency f. This corresponds to many real-world applications, such as audio detection (keyword spotting, speech detection...), motion detection (activity or gesture recognition...), or anomaly detection problems.

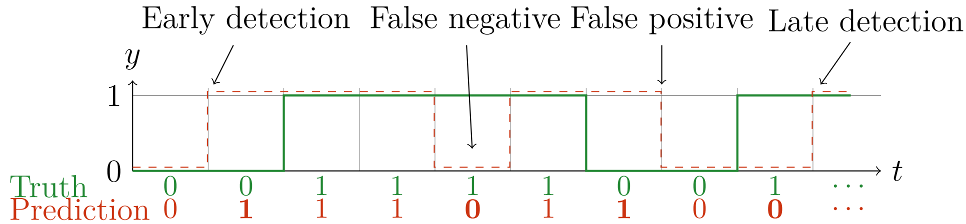


Fig. 3.1 Example of a frame-by-frame event-based classification encoded by binary vectors. We observe four possible misclassifications.

When dealing with such frame-based event classification, some ambiguity arises in model predictions, as Figure 3.1 shows. In particular, designing an appropriate custom metric is essential because the dataset may have a significant class imbalance between the presence or absence of an event. So using a plain accuracy metric comparing two binary vectors is not sufficiently relevant. However, designing such a metric adds more challenges since it requires some domain-specific knowledge and may involve multiple iterations before automating the tinyML workflow.

To address this challenge, we propose a novel tinyMLOps solution that can design datasets, deploy and evaluate models on ultra-low power sensors for both general and event-based classification.

In this work, we focus on the entire tinyMLOps process (Figure 3.2) rather than the technical implementation details. Specifically, we consider frame-based event classification on real-time sensors with ultra-low power MCUs, such as Arm Cortex-M0+ or M4. We also report some comparative results against tf.lite (David et al., 2021) and NNoM (Ma, 2020) for activity recognition (Table 3.1). Finally, we evaluate a model using our solution with a custom-designed metric for frame-based event detection on a head gesture dataset (Figure 3.3) that allows us to deliver the optimal user experience.

3.2.2 TinyMLOps for real-time sensors applied to frame-based event classification

Figure 3.2 illustrates the tinyMLOps workflow as an iterative process and shows some examples of indirect iteration causes. We discuss each tinyMLOps step and discuss its specific challenges and solutions for frame-based event classification on real-time sensors.

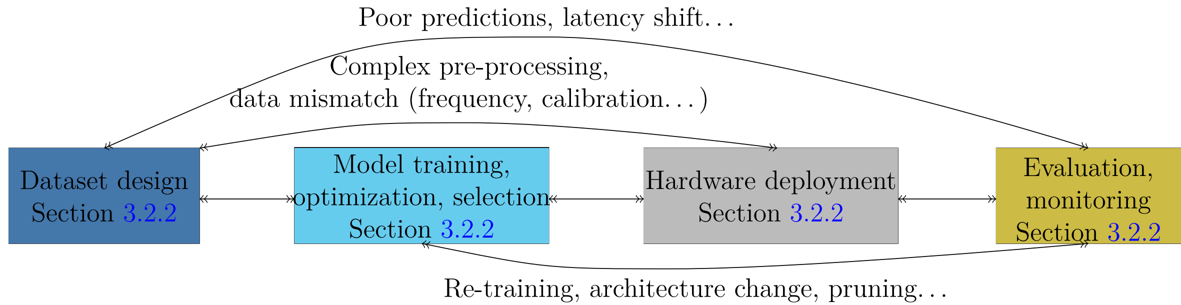


Fig. 3.2 TinyMLOps: Steps and examples of indirect iteration causes.

Dataset design

Building a good frame-based event dataset is difficult because it often requires some expert knowledge for preprocessing. For instance, in audio classification, domain knowledge is of great value to create realistic background noise and other data augmentation strategies (Park et al., 2019) which are crucial to the overall model performance.

In the case of a continuous signal, it may require domain knowledge to annotate the precise time window (start to finish) of an event. Moreover, the sampling frequency of the data must be set prior to proceeding with future steps, as it can impact the entire process. A low frequency is cost-effective, but may affect model accuracy due to the loss of details in the continuous inputs. While a high frequency may not be supported or will drain the power supply.

Figure 3.1 illustrates some issues encountered in frame-by-frame event-based classification. For the first event, the model predicts three ambiguous outputs: an early detection, a false negative in the middle, and a late classification at the end. These predictions raise the questions such as:

- Is early detection considered valid? If so, how much early?
- Same for late detection: how much is too late?
- Should the model be correct at all times during an event? If so, what is the accepted tolerance for classification errors? It may indicate a failed model or a noisy sample.

The tolerance for early and late detection can actually be considered as a hyperparameter of the dataset and model metric, requiring fine-tuning as well as going through the whole workflow to have feedback. We can design the expected truth vector y to account for this tolerance by padding an event of n frames in the y vector. However, a high latency tolerance may lead to a more lenient model, poor user experience, or higher power consumption. Determining the impacts of these factors during dataset design can be challenging.

Model training, optimization, and selection

The first challenges in model training for tinyML and event-based classification are to select a viable architecture in terms of size, and then ensure that it can contain operations (e.g., activations) that will be supported for MCU inference.

The most common model architectures for sequential data classifications essentially include these models or a variation of them: CNN, RNN, transformers (Vaswani et al., 2017), or residuals (He et al., 2016). However, due to their smaller size and performance, CNNs and RNNs are better candidates for ultra-low power MCUs. For less complex or shorter sequences, we may choose GRU over LSTM for a lower energy footprint (Cahuantzi et al., 2021).

The next challenge is to carefully select hardware-compatible activations or to design custom approximations such as piece-wise functions. Typically, exponentials, explicit divisions ($\neq 2^n$), or square roots are difficult to support on constrained hardware.

Hardware deployment and code generation

Once the model is trained, it must be quantized, converted, and deployed for the target hardware, which is a tremendous challenge for tinyMLOps because each hardware platform has its own specifications and requirements.

TinyML is still in its early stages, so suitable libraries for model conversion and inference on ultra-low power MCUs still remain scarce. Some important criteria for industry adoption are:

- high performance: accuracy, multiply-accumulate operations (MACs), latency, code size,
- flexibility, and support for custom layers/activations or some final optimization,
- hardware compatibility and ease of deployment,

- white-box solution to debug and understand the deployed model bottlenecks or issues for developers, appropriate metric,
- open source for commercial use if possible.

We tested both tf.lite (David et al., 2021) and NNoM (Ma, 2020) as framework candidates for hardware deployment. We found that tf.lite gives the best performance out of the two, but is heavy in size (Table 3.1) and the generated model is difficult to debug and does not allow for easy customization or optimization of layers and activations. In particular, it does not support GRU layers with a stateful option, which can be necessary for some real-time applications (e.g., speech or gesture recognition) and has a better performance tradeoff than LSTM for some event-detection problems (Cahuantzi et al., 2021).
In contrast, NNoM generates standard C code, hence it is more transparent and flexible. It also supports CMSIS-NN (Lai et al., 2018), which can reduce the inference cost by four on ARM Cortex-M MCUs, but outputs unstable results (Table 3.1).

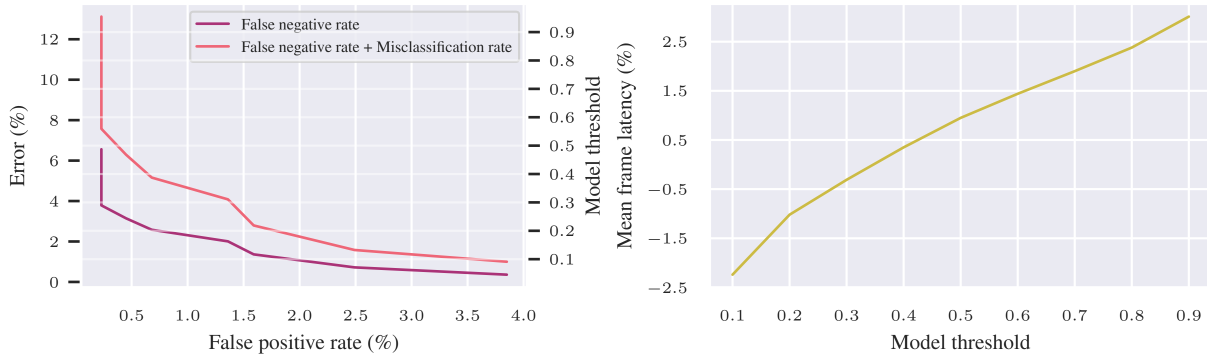
To overcome the aforementioned drawbacks, we created our own tinyMLOps solution. We chose to generate standard C code with CMSIS-NN from a TensorFlow model, like NNoM, to provide more usage and implementation flexibility, cost-efficient inference, debugging, wide hardware support, and lightweight deployment. This approach allows us to test and add custom activations and layers.

Table 3.1 Comparison of a model quantized to int8 deployed on an Arm Cortex-M4 MCU using tf.lite (David et al., 2021), NNoM (Ma, 2020) and our tinyMLOps solution on an activity recognition dataset.

Metric	tf.lite	NNoM	Our
Accuracy (%)	85.5	**68.24**	**86.95**
Model size (KB)	**6.72**	**0.29**	1.41
Stack size (KB)	**12**	6.1	**5.5**
Code memory (KB)	**303**	16.12	**5.5**

Evaluation and monitoring

TinyMLOps comparison. To evaluate our solution against tf.lite and NNoM, we quantized and deployed the same model to 8-bit integers on an activity recognition dataset using the same MCU platform, and measured the accuracy and memory sizes. We restricted the model selection by what was supported by all three solutions.



(a) False positive rate versus error rate for different error types (See Appendix 3.A listing errors in a confusion matrix): False negative rate and misclassification rate between positive classes are measured by varying the model output threshold.

(b) Model threshold versus average number of frames latency. Early accepted frames are defined as those captured 10 prior to the actual event, and late accepted frames are set to 20 after the actual event. Frame rate is at 20Hz.

Fig. 3.3 Custom metrics evaluated by varying the decision threshold and measuring event errors or latency of a GRU model on a four-class head gesture test set. The model is deployed on an Arm Cortex-M0+ MCU.

Based on the results presented in Table 3.1, using C code generation, as opposed to an interpreter-based approach, enables the deployment of lighter working models on ultra low-power MCUs. Our solution achieved an acceptable memory footprint, even though the NNoM model size was the lightest among the three, while tf.lite may not even fit in some ultra-low power settings. In addition, our solution obtained the best accuracy, with tf.lite coming in second place.

Custom event metric. In addition to padding the dataset for early/late event windows (Figure 3.1), we can also create a custom metric that accepts early/late detections and have some frame tolerance. This allows us to properly evaluate the model's prediction from a frame-by-frame vector to an event-based confusion matrix (Appendix 3.A) shows an example of a generalized confusion matrix for multiclass event-based classification.). We evaluate a stateful GRU model with return sequences for an always-on real-time multiclass head gesture detection problem, with a rejection class (i.e., no gestures). The model was quantized to 8-bit integer and deployed on an Arm Cortex-M0+.

As depicted in Figure 3.3a, there is a tradeoff between false positives versus false negatives and misclassification through the model threshold output. Setting a high threshold results in more false negatives, which adversely impacts user intent (e.g., missed a screen wake-up or wrong gestures) and also increases latency (Figure 3.3b). In contrast,

a low threshold results in more false positives, which wastes power supply (e.g., unwanted screen wake-up), but reduces latency. For example, if a user desires a false positive rate below 1.5%, then they must accept an overall classification error (excluding false positive) of at least ≈ 3%, which corresponds to a threshold of at most 0.25. This threshold also leads to an average frame latency of ≈ -0.5 frame, indicating early detection. The threshold sweet spot varies depending on the specific application: critical applications in cars may require accurate and quick detection (low false negative and low threshold), while wearable applications such as motion detection may tolerate less false positive for better battery life.

Thus, the evaluation and design of metrics for frame-based events require domain knowledge to fine-tune time-related hyperparameters to assess real-life user experience. It can also serve as a model selection criterion for specific applications. In addition, deploying on edge devices requires consideration of more hardware-related metrics, such as MACs and power usage or data movement costs (sRAM and flash) (Svoboda et al., 2022).

3.2.3 Conclusion

Our work highlights unique key challenges at each direct and indirect iteration of the tinyMLOps workflow and demonstrates the potential to deliver working neural network applications on the tiniest constrained hardware with care and given the early stage of the field. Specifically, we showed that frame-based events classification requires domain knowledge and careful fine-tuning to design appropriate datasets and metrics to deploy optimal models for real-time inference on ultra-low power MCUs.

The successful advancement and potential of the tinyML field critically depend on the continual innovation and implementation of efficient tools and methods by both the industry and researchers to deliver impactful and practical applications at the edge.

3.3 ML2MCU: Towards automated large-scale deployment of tiny neural networks

In the following sections, we introduce our tinyMLOps solution called *ML2MCU* (Ataya, 2022) to train, quantize, and deploy deep learning systems for MEMS-based applications in the most constrained settings: under 8 kB RAM, 16 kB ROM memory, and 70 µW power usage (Table 1.4). In particular, the 4MHz clock frequency is the lowest found in 32-bit microcontrollers, effectively achieving extreme-low power levels, compared to

other MCUs that use 120 to 300MHz. This extreme-low power setup is applied to motion-related applications. Audio-related applications, which are more demanding, target the Arm Cortex-M4 or M7. Our solution enables the automatization of the design process of creating efficient algorithms for MEMS-based applications for non-practitioners in machine learning through a uniform approach.

ML2MCU provides comprehensive features for tinyMLOps, the main contributions of this work are emphasized by the following features:

- *data augmentation*, specialized for sensor signals (audio, motion),
- *model training*,
- *8-bit weight quantizer, and model conversion to lightweight and portable fixed-point C code library,*
- *integer-only inference with fixed-point arithmetic for extreme-low power always-on and real-time inference,*
- *wide hardware support*,
- *specialized metrics for event-based classification to match user experience*
- data visualization per sample,
- meta-optimization, and model selection,
- on-device evaluation.

3.3.1 Dataset design

Overview of the MEMS-based dataset

Table 3.2 provides a reference to the datasets on MEMS-based applications we benchmark in this work, provided by TDK InvenSense. They essentially consist of motion-based applications, which are typically low-frequency (up to 100Hz), and audio applications with higher frequency (up to 16kHz).

All of these applications are expected to run always-on, in real-time, on battery, and attempt to fit the lowest end possible of target hardware platforms to reduce production, consumption, and consumer costs (Table 1.4).

Table 3.2 Dataset reference for TDK InvenSense's MEMS-based applications.

Dataset	Data type	Description	Classs number	Class	Frequency (Hz)
Bring-to-see	Accelerometer	Wake up a smartwatch from wrist motion	2	Wake motion, other	20
Hand gesture	Accelerometer gyrosccope	Wrist gestures for a smartwatch	6	Wrist in, out, up, down, shake, other	50
Head gesture	Accelerometer gyrosccope	Head gestures for an earbud	4	Yes, no, nod, other	50
Finger gesture	Accelerometer gyrosccope	Finger gesture from a smartwatch	9	Clench, (1x, 2x) pinch (1x, 2x, 3x, 4x) tap, other	50
Voice activity detection (VAD)	Accelerometer	Wake up an earbud on voice from motion detection	2	Voice activity, other	800
Keyword spotting: wake word	Microphone	Wake a device on spoken keyword (e.g., "Siri")	2	Keyword spotted, other	16k
Keyword spotting: command words	Microphone	Spoken keyword command action	12	Google Speech commands v2-12 (Warden, 2018)	16k

Data augmentation

Among the various applications listed in Table 3.2, keyword spotting is the most challenging in terms of dataset design and training a model to generalize. Audio data have a high variance in voices, pitch, accents, and background noise; plus, some words may sound similar or be a subset of other words, such as "Alexa" and "Alexandra". Thus, keyword spotting requires capturing a large quantity and diversity of human voices in various contexts.

Therefore, we need to use data augmentation to design a dataset that is robust for real-world usage and against adversarial contexts, such as music background or distorted voices.

We propose a high-level algorithm (Algorithm 1) and an `AudioSample` API class provided in Appendix 3.B to design a robust keyword spotting dataset. This work is part of the contribution of the patent (Ponçot et al., 2022), under review, and laid the foundation for further work by Rémi Ponçot, who improved the data augmentation scheme at a lower level with specialized audio processing (e.g., time distortion). From a high-level point of view, a robust keyword spotting classification can differentiate between target keywords (***positives***) and non-keywords (***negatives***) in any given sound environment (***background***). We define an audio sample by a random background noise with a certain volume level ***target_decibel*** of a fixed ***duration***, which is overlaid with a random number of positive and negative samples in a random position.

Figure 3.4 illustrates what an audio sample is and the API is provided in Appendix 3.B.

We hypothesize that data augmentation can help design smaller and more robust models because higher quality training data results in better and easier convergence, so a smaller expressiveness capacity would be required to reach generalization.

Algorithm 1 High-level data augmentation for robust audio samples for keyword spotting.

```
1: Input: duration, positive_samples, negative_samples,
2: background_samples, target_decibel
3: Output: audio_sample
4: n_positives ← random_integer(min = 0, max = 3)
5: n_negatives ← random_integer(min = 0, max = 3)
6: background ← sample_N_audio(background_samples, n = 1)
7: list_samples ← sample_N_audio(positive_samples, n = n_positives)
8: list_samples ← sample_N_audio(negative_samples, n = n_negatives)
9: audio_sample ← match_target_amplitude(background,
   target_decibel, duration)
10: for sample ∈ list_samples do
11:     if not is_overlapping_previous_samples(audio_sample, sample) then
12:         audio_sample ← overlay(audio_sample, sample)
13:     else
14:         Repeat
15:     end if
16: end for
17: Return audio_sample
```

3.3.2 Model design and training

As mentioned in Section 2.1, GRU-based models have been shown to provide a good size-performance tradeoff. Adding a 1D convolution to the GRU model can provide more efficient feature extraction and improve accuracy, with reasonable additional computation. Our experiments empirically support Arik et al. (2017), Cahuantzi et al. (2021) that GRU or 1D convolutional GRU (CGRU) models consistently provide acceptable performance across all of our MEMS-based applications for always-on, real-time inference listed in Table 3.2 with reasonable size. Thus, we found a polyvalent hardware-friendly template model that contributes toward an automated and uniform algorithm design process for future MEMS-based applications. Therefore, the remaining most critical hyperparameters are only the number of filters f of the CNN and cells c of the GRU layer, so the hyperparameter space is reduced and easier to explore and exploit.

ML2MCU provides a meta-optimizer interface to probe the best size-performance tradeoff between f, and c, as illustrated in Figure 3.5, and supports other hyperparameters such as activations, boolean addition of layers (dropout, batchnorm, ...).

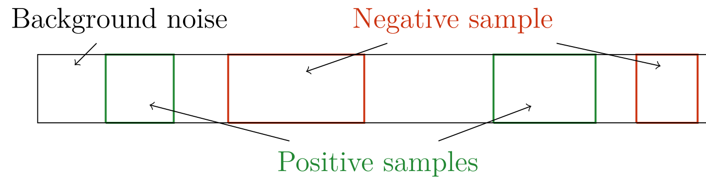


Fig. 3.4 Illustration of an audio sample, with random samples, position, and a number of positive and negative keywords.

3.3.3 Deployment

In this section, we first motivate our choice to create a quantization framework from scratch due to the gap in the existing literature, then present our objectives, requirements, and methods for our proposed solution, and demonstrate that our deployed neural networks successfully deliver robust and competitive results under the strictest power footprint constraints as per state-of-the-art and industry standards.

Comparison of quantization framework candidates

Based on our review of quantization frameworks in Section 1.4.1, TFLM (David et al., 2021) and NNoM (Ma, 2020) seem the most appropriate framework candidates for deploying neural networks.

However, TFLM has unsupported operations, such as GRU or 1D convolution, which we rely on in all of our applications. TFLM uses an interpreter-based approach at runtime, which increases its complexity and makes it difficult to develop missing features. Moreover, this prevents it from providing hardware-specific optimizations.

Moreover, as stated in Lê and Arbel (2023), we found supporting results that TFLM is on the heavier side, and NNoM is on the lighter side of the quantization frameworks (Zingg and Rosenthal, 2020). When deploying a CNN model on a microcontroller, Zingg and Rosenthal (2020) measured a 25 times RAM usage ratio and a 75% increase in flash size between TFLM and NNoM. Consequently, this means that in some of our applications, using TFLM would require upgrading the MCU model to a more power-hungry and costly target, such as Arm-Cortex M4 instead of M0+, which could increase production costs by a factor of 10, and power usage by a factor of approximately 1500 (Table 1.4). In addition, they also show that on-device inference is at least five times slower in TFLM than NNoM, and has lower quantized accuracy.

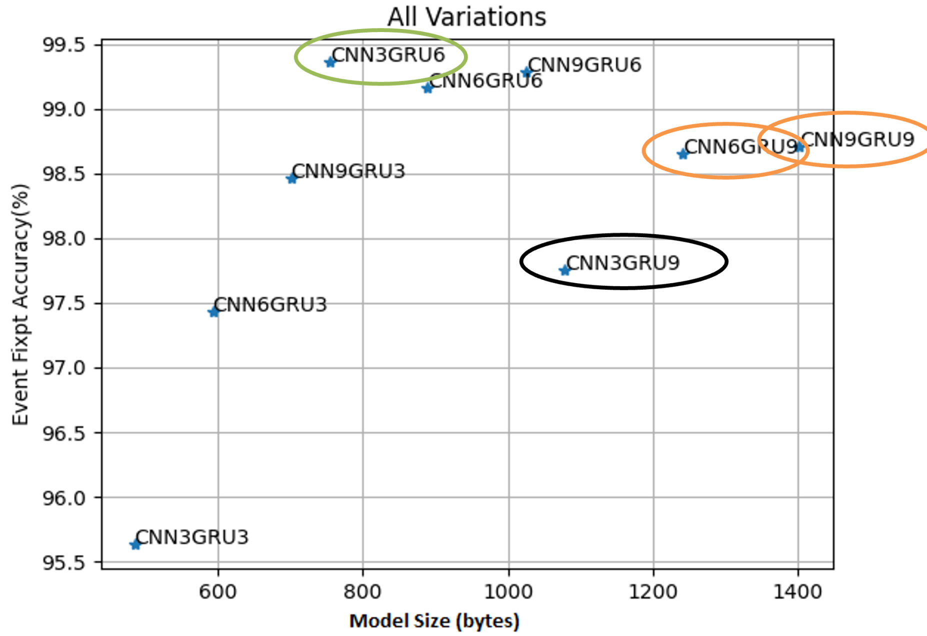


Fig. 3.5 Tradeoff between size and on-device accuracy for a CGRU model applied on the bring-to-see dataset, using ML2MCU's meta-optimizer. Optimal tradeoffs are highlighted in green, suboptimal tradeoffs in red, and manually designed models in black.

Thus, we argue that while TFLM is a popular option due to its wide support and ecosystem, it is not a viable option for our extreme-low power requirements.

Regarding NNoM, while it offers an accessible and flexible approach based on C code generation, we found unstable results in the HAR application, as previously mentioned (Table 3.1).

Therefore, this required us to create and develop our own deployment framework (Ataya, 2022) to quantize and deploy neural networks on ultra-low power embedded devices: *ML2MCU*.

Proposed framework for quantization and deployment of neural networks

Our proposed quantization framework is based on C code generation, similar to NNoM, but includes the following features that differentiate it from previous work:

- added full support for RNNs (simple RNN, GRU, LSTM), 1D CNN, with standard options (e.g., `return_sequence` for GRU), and other missing features from TFLM,
- full integer-only inference of neural networks, with 8-bit weights and 32-bit activations and data,

- re-implementation of more efficient activations in fixed-point arithmetic, using only basic operations, such as additions, multiplications, bit-shift, power of twos, missing in NNoM,
- an API to simulate fixed-point inference in Python for fast prototyping and verification,
- minimal and plain C code generation of neural networks for the most constrained power footprint requirements in the state-of-the-art, not previously achievable through TFLM,
- flexibility for implementing custom layers or activations, difficult in TFLM,
- wide hardware support for lower integration efforts,
- stable inference and bug fixes, encountered in NNoM,
- causal inference for real-time predictions.

The deployment process takes as input a trained model and outputs portable C code as `model.c`, and `model.h` files and consists of the following steps:

1. Parsing the model, layer by layer,
   (a) converting the layer's weights to 8-bit integers, and remaining parameters to 32-bit integers (bias, batch normalization, activations) using min/max uniform symmetric quantization (Algorithm 2),
   (b) computing the scale S (Equation (1.7)) for the current layer,
   (c) generating the corresponding C code for this layer, which makes calls to the inference library and writes the weights to a `model.c` file.
2. Compile and build the C code source.
3. Integrate and deploy the built source to the microcontroller.

Quantization and fixed-point arithmetic

In this section, we provide a step-by-step computation of some standard neural network operations in fixed-point. As stated in Section 1.1.1, neural networks are complex functions composed of a chain of compound operations, such as multiplications, additions or non-linear functions.

We first detail how to quantize a matrix to N-bits, then we present a side-by-side comparison of the computation of an FC layer in floating-point and 8-bit fixed-point. Finally, we apply the same principle to a GRU layer, based on the TensorFlow implementation.

Symmetric quantization. Algorithm 2 quantizes a matrix to N-bits. On line four, the maximum value of W becomes mapped to the lower or upper bound in 2^{N-1}, e.g., -127 if the absolute maximum value of W is the minimum. Then, the clipping function in line five forces for symmetric quantization. Note that line four and five are where the numerical precision of values is reduced.

Algorithm 2 Symmetric quantization of a floating-point matrix to N-bits : NBitQuantize.

1: **Input:** $\boldsymbol{W}, \boldsymbol{N}$
2: $\mathbf{max}_{\boldsymbol{w}} \leftarrow \max(|\boldsymbol{W}|)$
3: $\boldsymbol{S} \leftarrow \lfloor \mathbf{max}_{\boldsymbol{w}} * 2^N \rceil$
4: $\boldsymbol{W}_{\mathrm{q}} \leftarrow \lfloor \boldsymbol{W} * 2^{N-1} / \mathbf{max}_{\boldsymbol{w}} \rceil$
5: $\boldsymbol{W}_{\mathrm{q}} \leftarrow \mathrm{clip}(\boldsymbol{W}_{\mathrm{q}}, -2^{N-1}, 2^{N-1})$
6: **Output:** $\boldsymbol{W}_{\mathrm{q}}, \boldsymbol{S}$
7: **Return** $\boldsymbol{W}_{\mathrm{q}}, \boldsymbol{S}$

FC layer. We compute a step-by-step floating-point versus fixed-point inference of an FC layer with 8-bit weights and 32-bit activations. Algorithm 4 is highly efficient because it only uses hardware-friendly operations, such as additions, multiplication, and bit shifts.

Algorithm 3 Floating-point fully-connected: FloatFC.

Input: $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}$

Output: $\boldsymbol{y}$

$\boldsymbol{y} \leftarrow \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$

Return $\boldsymbol{y}$

Algorithm 4 Integer-only fixed-point fully-connected inference: QuantizedFC.

Input: $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}$

Output: $\boldsymbol{y}_{\mathrm{q}}$

$\boldsymbol{W}_{\mathrm{q}}, \boldsymbol{S} \leftarrow \mathrm{NBitQuantize}(W, 8)$

$\boldsymbol{b}_{\mathrm{q}}, _ \leftarrow \mathrm{NBitQuantize}(W, 32)$

$\boldsymbol{x}_{\mathrm{q}}, _ \leftarrow \mathrm{NBitQuantize}(x, 32)$

$\boldsymbol{y}_{\mathrm{q}} \leftarrow \boldsymbol{S}\boldsymbol{x}_{\mathrm{q}} + 2^7$

$\boldsymbol{y}_{\mathrm{q}} \leftarrow (\boldsymbol{y}_{\mathrm{q}} \gg 8)$

$\boldsymbol{y}_{\mathrm{q}} \leftarrow \boldsymbol{y}_{\mathrm{q}}\boldsymbol{W}_{\mathrm{q}} + 2^7/2$

$\boldsymbol{y}_{\mathrm{q}} \leftarrow \boldsymbol{y}_{\mathrm{q}} \gg 7$

$\boldsymbol{y}_{\mathrm{q}} \leftarrow \boldsymbol{y}_{\mathrm{q}} + b_{\mathrm{q}}$

Return $\boldsymbol{y_q}$

Quantized activations. For each commonly use activations (ReLU-based, tanh, sigmoid, softmax, ...) we implemented an optimized quantized alternative that only relies on hardware-efficient operations, such as bit-shifts, simple multiplications, additions, ...

Currently, these quantized activation functions are proprietary and cannot be disclosed. Nonetheless, we can provide an instance of such functions to illustrate the concept.

We introduce the hard Elu function, a hardware-friendly approximation of the Elu function for $\alpha = 1$, which removes the expensive exponential computation, as an example of an efficient activation as follows:

$$\text{hard Elu}(x) = \begin{cases} -1 & \text{if } x < -2, \\ -0.5x & \text{if } -2 \leq x \leq 0, \\ x & \text{otherwise.} \end{cases} \tag{3.1}$$

The L^1 distance between the original Elu and our approximation is equal to 0.32, as illustrated in Figure 3.6 and explained in Algorithm 6.

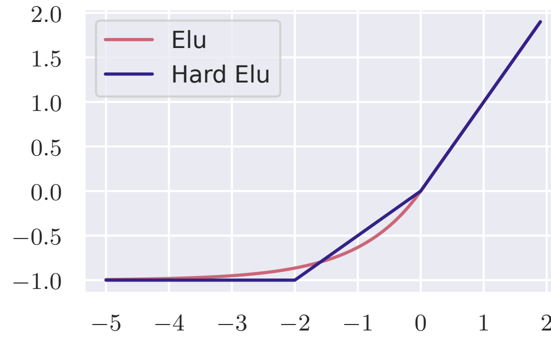


Fig. 3.6 Elu versus hard Elu function. A L^1 distance between the original Elu and our approximation ≈ 0.32 for x ∈ ℝ.

Algorithm 5 Floating-point hard Elu: FloatHardElu.

1: **Input:** $\boldsymbol{x}$
2: **Output:** $\boldsymbol{y}$

3: **if** $x < -2$ **then**
4: $\quad \boldsymbol{y} \leftarrow -1$
5: **end if**
6: **if** $x \in [-2, 0]$ **then**
7: $\quad \boldsymbol{y} \leftarrow x/2$
8: **else**
9: $\quad \boldsymbol{y} \leftarrow x$
10: **end if**
11: **Return** $\boldsymbol{y}$ =0

Algorithm 6 Fixed-point hard Elu: QuantizedHardElu.

1: **Input:** $\boldsymbol{x_q}$
2: **Output:** $\boldsymbol{y_q}$

3: **if** $x < -2$ **then**
4: $\quad \boldsymbol{y}_\text{q} \leftarrow -1$
5: **end if**
6: **if** $x \in [-2, 0]$ **then**
7: $\quad \boldsymbol{y}_\text{q} \leftarrow x \gg 2$
8: **else**
9: $\quad \boldsymbol{y}_\text{q} \leftarrow x$
10: **end if**
11: **Return** $\boldsymbol{y}_\text{q}$

Other quantized layers, such as RNNs or CNNs, are built on the same principle and call the $QuantizedFC$ function.

We implemented all standard layers in Python, which allows us to simulate the inference and accuracy and provides an interface for debugging each quantized operation before deployment. We chose to quantize weights with lower precision than activations because the weights are more robust to low-bit quantization than activations, as confirmed in previous studies (Gong et al., 2019, Liang et al., 2021). In particular, Liang et al. (2021) recommends keeping the biases, first and last layers, at higher precision to maintain performance. We empirically confirmed that activation precision is more critical than

weights (Liang et al., 2021), so weights can be converted to lower-bit precision than weights.

3.3.4 Evaluation

As presented in Lê and Arbel (2023), event-based classification requires custom metrics that can closely represent the user experience in the real world. However, our contribution in Ponçot et al. (2022) demonstrates that relying solely on plain frame-by-frame accuracy for evaluating event-based classification is insufficient due to the high class imbalance of non-events (class 0), and the low frame granularity that results in biased accuracy results. To more accurately reflect the user experience, we propose a higher-level metric that evaluates complete events as a list of continuous frames.

Additionally, we revisit the standard labeling approach because labeling each sample to the precise frame can be challenging. For example, in gesture recognition, the sensor may detect small variations in hand position throughout the movement, making it difficult to define the clear start and end boundaries of an event. Therefore, we introduce tolerance hyperparameters for early and late frame accepts, which correspond to the maximum number of frames around the labeled events. These hyperparameters affect the evaluation of the user experience. A high late frame tolerance may result in higher benchmark performance, but it can overestimate accuracy and underestimate latency in real settings. Conversely, a high early frame tolerance may result in underestimated accuracy but lower latency.

We can also adjust the model output threshold as presented in Lê and Arbel (2023), which directly impacts the measured false positive and negative tradeoff, in addition to the latency.

Careful consideration of these hyperparameters promises to accelerate the algorithm design feedback loop to iterate in the tinyMLOps process (Figure 3.2), and deliver high-quality specialized models that match user experiences in real-life.

In the next section, we present our results on a portfolio of MEMS-based applications (Table 3.2) that showcase such high-quality user experience while respecting our tight constraints.

3.4 Results

Table 3.3 presents our benchmark results for motion detection datasets, using accelerometers or gyroscope sensors. We keep the model size under 8kB, except for the finger gestures

applications, enabling to use the low-cost Arm Cortex M0+ MCU. Moreover, neural networks on-device are highly responsive; we obtain a latency under 60ms and an inference time per sample under 10ms. Furthermore, we obtain energy-efficient models that require under 40 µA of power consumption.

Table 3.3 Benchmark results of 8-bit neural networks deployed on ultra-low power MCUs for motion detection datasets.

Specification	Bring-to-see	Hand gestures	Head gestures	Finger gestures	HAR
MCU	M0+ (4 MHz)	M0+ (4 MHz)	M0+ (4 MHz)	M4 (120 MHz)	M0+(4 MHz)
Accuracy (%)	98.59	99.15	99.05	97.77	89.3/ 86.95
Data frequency (Hz)	20	50	50	400	100/20
Flash (kB)	2.2	3.61	2.8	6.1	9.9 /1.41
Data RAM (kB)	0.4	0.8	0.7	3.6	2.3 /5.5
Power (µA)	27 (29nJ)	39	35	—	—
Latency (ms)	17	26	55	25	—
Inference (ms)	3.2	8	4.33	2.9	2.9
Cycle count	14693	—	—	—	—

Table 3.4 presents our benchmark results for audio applications, using only an accelerometer for VAD, and a microphone for KWS. We achieve competitive accuracy for keyword spotting under 40kB, with again responsive and low-energy on-device inference.

We also compared the embedded inference of our keyword spotting system with a cloud server, where only the wake word model is embedded, and the more demanding command word model is deployed on the cloud, when the wake word model triggers. In this scenario, the cloud-based solution adds an inference overhead of 2.5 seconds to the wake word model due to additional network connection, resulting in approximately 15 times slower inference and inherently larger power footprint overall, compared to a self-contained fully embedded model.

Therefore, we successfully meet the constraints of ultra-low power hardware on motion and audio applications. In particular, in motion applications (except finger gestures), we reach extreme-low power inference levels, which redefines the state-of-the-art notion of tiny neural network models while maintaining high accuracy above 97%, except for HAR.

3.5 Discussion

3.5.1 Conclusion

TinyML poses novel and unique challenges for the machine learning and hardware communities to fit expensive functions on constrained devices, which requires careful

Table 3.4 Benchmark results of 8-bit neural networks deployed on ultra-low power MCUs for audio applications.

Specification	VAD	KWS: wake word with command words (MCU)
MCU	M0+ (4 MHz)	M7 (300 MHz)
Accuracy (%)	99	99.43
Data frequency (Hz)	800	16k
Flash (kB)	1.16	37.50
Data RAM (kB)	0.59	9.80
Stack (kB)	—	1.3
Power (µA)	175	39
Latency (ms)	130	—
Inference (ms)	—	160
Cycle count	1318	—

consideration and specialized solutions. Its corresponding high potential for large-scale intelligent sensing applications motivates our interest. We demonstrated that existing state-of-the-art solutions lack performance, power footprint, flexibility, and production readiness. This motivated us to create our own tinyMLOps solution that addresses previous issues, to automatically train, quantize to 8-bits, deploy, and evaluate deep learning-based systems that can run on the most constrained platforms found in research and industry.

Our contribution laid the foundation for a platform that enables cost-efficient deployment of neural networks on ultra-low power devices for MEMS-based applications. In particular, we dedicated special attention to designing and evaluating event-based models to ensure that the reported inference closely aligns with real-world user experiences. We successfully deployed responsive and ultra-low power neural network models that utilize as little as 1kB of memory, or 30 µA of power, to adhere to the always-on, real-time inference requirements. These figures represent a reduction in power footprint by hundreds to billions of orders of magnitude compared to models and platforms designed for cloud or mobile applications (Table 1.3), redefining the notion of tiny neural networks and extreme-low power inference levels. This billion order of magnitude also effectively translates into billions of savings in the microcontroller consumer market. Additionally, our work allowed us to apply a uniform algorithm design process that can be generalized to all of our MEMS-based applications. Consequently, this empowers anyone to easily create new algorithms for MEMS-based applications, paving the way for large-scale adoption and deployment of intelligent embedded devices capable of interacting with their local environment in a low-cost and low-power manner.

3.5.2 Limitations

We restricted our benchmark to TFLM and NNoM as they appeared to be the most popular or promising framework candidates for our context. However, other quantization tools initiatives exist, such as µTVM (Chen et al., 2018, Liu et al., 2023,?), N2D2 (Bichler et al., 2017), which have shown promising results in the deployment of neural networks on embedded devices. However, the first µTVM release was not available[1] before we finalized our first release[2], and is still in the early development stage (Sipola et al., 2022). N2D2 (Bichler et al., 2017) is another European research initiative for end-to-end design and deployment of neural networks that supports a wide range of software solutions (TensorFLow, PyTorch, ONNX, C++, TensorRT, ...), and hardware, from GPUs to microcontrollers, and is open source.

3.5.3 Future work

While we laid a strong foundation for developing and deploying efficient neural networks on the most limited embedded devices, the field is still in its early stages. There remain numerous exciting challenges and opportunities for improvement.

We identify several future avenues of exploration for our work. We can aim to speed up the time-to-market by improving each step of the tinyMLOps workflow. We can work on bug fixes and improvements for a more seamless user experience and deployment process for non-practitioners in machine learning. In particular, exploring new data augmentation strategies can help reduce manual data collection and labeling efforts, train more robust models, and also smaller architectures (due to less expressive power required). Additionally, we seek to extend our portfolio of MEMS-based applications to refine our deployment pipeline for efficient neural networks on MCUs. Finally, we can focus on researching new methods to further reduce our models' power footprint, such as structured pruning, experimenting with new architectures for sequence classification, or lower-bit quantization of activations or weights.

In Chapter 4, we introduce a generalized N-bit quantization scheme that can enable extreme low-bit precision, such as 1-bit.

[1] commit hash: 7b3a22e, November 12th, 2021.

[2] This thesis started June 15th, 2020, and our first proof of concept was operational in April 2021.

Appendix

3.A Generalized confusion matrix for multiclass event classifications

Truth \ Prediction	Rejection	Class 1	... Class N
Rejection	True positive	False positive	False positive
Class 1	False negative	True positive	Mis-classification
... Class N	False negative	Mis-classification	True positive

Fig. 3.A.1 Error types of a generalized confusion matrix for multiclass event classifications with a rejection class (i.e., non-event). There is an additional error type compared to binary event classification: misclassification between positive classes.

3.B AudioSample API class documentation

```
NAME
    AudioSample - Define the AudioSample class to synthesize audio samples.

CLASSES
        AudioSample

    class AudioSample(builtins.object)
```

```
8         AudioSample(st: str, positives: List[pydub.audio_segment.AudioSegment],
          ↪  negatives: List[pydub.audio_segment.AudioSegment], backgrounds:
          ↪  List[pydub.audio_segment.AudioSegment], target_dBFS: int, db_pos_path:
          ↪  str, subfolder_to_create: str = '')
9
10        This class defines how one audio sample data is synthesized.
11                - It has a fixed duration of 10s.
12                - It has 1 random background sample.
13                - It has a random number (or none) of random overlaid positive
                  ↪  samples.
14                - It has a random number (or none) of random overlaid negative
                  ↪  samples.
15
16        The constructor takes as input arguments:
17                - Sample type in [train, val, test]
18                - All the audio databases (positives, negatives, and background
                  ↪  audio),
19                - Target dBFS: target dB for the background noise,
20                - db_pos_path: positive database folder to use,
21                - new database folder (if not exists) to create and write the
                  ↪  AudioSample in.
22
23        Any successful instantiation of this object results in the creation of:
24                - A new audio sample .wav,
25                - Its corresponding spectrogram (using PSD) X of Tx timestep and
                  ↪  n_freq frequencies (shape=(101, 1998)),
26                - Its corresponding Y label vector (binary) of length Ty (shape=(1,
                  ↪  498)) where a positive sample is identified by nb_ones_label '1'
                  ↪  continuous values,
27                - An index file if not created or new index lines appended to this
                  ↪  file,
28                - A new database folder containing 3 nested folders among
                  ↪  'train'/'val'/'test' if not created.
29
30        Methods defined here:
31
32        __init__(self, st: str, positives: List[pydub.audio_segment.AudioSegment],
          ↪  negatives: List[pydub.audio_segment.AudioSegment], backgrounds:
          ↪  List[pydub.audio_segment.AudioSegment], target_dBFS: int, db_pos_path:
          ↪  str, subfolder_to_create: str = '')
33            Args:
34                    st (str): dataset type
35                    positives (List[AudioSegment]): list of the positive samples
36                    negatives (List[AudioSegment]): list of the negative samples
```

```
37                  backgrounds (List[AudioSegment]): list of the background samples
38                  target_dBFS (int): target dB of the background
39                  db_pos_path (str): positive database folder to use
40                  subfolder_to_create (str): database folder to write the
                    ↪  AudioSample in
41              This function creates a new synthesized AudioSample.
42
43          create_audio_sample(self) -> Tuple[pydub.audio_segment.AudioSegment,
            ↪  numpy.ndarray, numpy.ndarray]
44              Creates a new synthesized audio file from the whole database as well as
                ↪  the corresponding spectrogram data and y labels. Also creates the
                ↪  necessary new folders if not created
45              Returns: the new synthesized audio, x data (spectrogram), and label y.
46
47          generate_audio_filename(self) -> str
48              Generate a unique filename to give to the AudioSample to be created
49              Returns: New filename
50
51          get_all_self_attributes_in_list(self) -> List
52              Construct the index file of one folder to log the AudioSample creation
                ↪  process (which/where/what composes the synthesized audio)
53              Returns: List of dictionaries (1 line of the index file)
54
55          get_last_filename_index(self) -> int
56              Get the index number of the last .wav file created to find a unique
                ↪  filename for the current AudioSample to be created
57              Returns: Last index file number of a .wav, else -1 if no audio files are
                ↪  found.
58
59          get_random_negative_samples(self, negatives:
            ↪  List[pydub.audio_segment.AudioSegment]) -> Tuple[List[int],
            ↪  List[Tuple[pydub.audio_segment.AudioSegment, int, int]]]
60              Select self.nb_neg random positive samples from the positive database
61              Args:
62                  negatives (List[AudioSegment]): negative database
63              Returns: Return the index list of the selected samples, and List of the
                ↪  extracted audio background, starting timestamp of the extracted
                ↪  audio, ending timestamp of the extracted audio
64
65          get_random_positive_samples(self, positives:
            ↪  List[pydub.audio_segment.AudioSegment]) -> Tuple[List[int],
            ↪  List[pydub.audio_segment.AudioSegment]]
66              Select self.nb_pos random positive samples from the positive database
67              Args:
```

```
68                      positives (List[AudioSegment]): positive database
69              Returns: Return the index list of the selected samples and the list of
                ↪   samples
70
71          insert_ones(self, y, segment_end_ms) -> Tuple[numpy.ndarray, int]
72              Update the label vector y. The labels of the 50 output steps strictly
                ↪   after the end of the segment should be set to 1. By strictly we mean
                ↪   that the label of segment_end_y should be 0 while the 50 following
                ↪   labels should be one.
73
74              Arguments:
75              y -- numpy array of shape (1, Ty), the labels of the training example
76              segment_end_ms -- the end time of the segment in ms
77
78              Returns:
79              y -- updated labels
80              segment_end_y -- Ending position of last '1' to insert in vector y
81
82          save_all_to_csv(self) -> None
83              Write X data, Y data, and index_origin to file
84              Returns: None
85
86          -------------------------------
87          Static methods defined here:
88
89          extract_random_audio_sample_from_original(original_audio:
            ↪   pydub.audio_segment.AudioSegment, is_bg: bool) ->
            ↪   Tuple[pydub.audio_segment.AudioSegment, int, int]
90              Extract an audio subsample from an original audio
91              Args:
92                      original_audio (AudioSegment): original audio to extract a
                        ↪   subsample from
93                      is_bg (bool): if it is a background sample
94              If original_audio is a background then extract 10s else, extract a random
                ↪   duration (~near 1s).
95              Returns: Tuple of the extracted audio sample, starting timestamp origin
                ↪   of the extracted audio,
96              ending timestamp origin of the extracted audio
97
98          get_random_background(backgrounds: List[pydub.audio_segment.AudioSegment]) ->
            ↪   Tuple[int, pydub.audio_segment.AudioSegment, int, int]
99              Args:
100                     backgrounds (List[AudioSegment]): background audio database
```

```
101             Returns: index of the selected sample, the extracted audio background,
                ↪  starting timestamp of the extracted audio,
102             ending timestamp of the extracted audio
103
104         get_random_time_segment(segment_ms) -> Tuple[int, int]
105             Gets a random time segment of duration segment_ms in a 10,000 ms audio
                ↪  clip.
106             Arguments:
107             segment_ms -- the duration of the audio clip in ms ("ms" stands for
                ↪  "milliseconds")
108             Returns:
109             segment_time -- a tuple of (segment_start, segment_end) in ms
110
111         insert_audio_clip(background, audio_clip, previous_segments) ->
            ↪  Tuple[pydub.audio_segment.AudioSegment, Tuple[int, int]]
112             Insert a new audio segment over the background noise at a random time
                ↪  step, ensuring that the audio segment does not overlap with existing
                ↪  segments.
113
114             Arguments:
115             background -- a 10 second background audio recording.
116             audio_clip -- the audio clip to be inserted/overlaid.
117             previous_segments -- times where audio segments have already been placed
118
119             Returns:
120             new_background -- the updated background audio
121
122         is_overlapping(segment_time, previous_segments) -> bool
123             Checks if the time of a segment overlaps with the times of existing
                ↪  segments.
124
125             Arguments:
126             segment_time -- a tuple of (segment_start, segment_end) for the new
                ↪  segment
127             previous_segments -- a list of tuples of (segment_start, segment_end) for
                ↪  the existing segments
128
129             Returns:
130             True if the time segment overlaps with any of the existing segments,
                ↪  False otherwise
131         -------------------------------
132         Data and other attributes defined here:
133         Tx = 1998
134         Ty = 498
```

```
135         audio_type_dict = {'bg': 3, 'neg': 1, 'pos': 0}
136         db_root_path = '/shared/db_train_val_test/'
137         duration = 10000
138         n_freq = 101
139         nb_max_neg = 3
140         nb_max_pos = 3
141         nb_ones_label = 18
142         ratio_reference_Tx_Ty = 4.008
143         ratio_reference_Ty_ones = 27.5
144         seed = 369
145         valid_set_type = ['train', 'val', 'test']
146         db_root_path = '/shared/db_train_val_test/'
```

Chapter 4

Regularization for hybrid N-Bit weight quantization of neural networks on ultra-low power microcontrollers

Abstract

In this chapter, we introduce a novel approach that generalizes quantization to N-bits, down to extreme low-bit precision, such as 1-bit quantization. This approach builds upon our tinyMLOps solution introduced in Chapter 3, as it is designed to integrate into our established tinyML end-to-end process seamlessly.

We propose a novel regularization method for hybrid quantization of neural networks, enabling efficient deployment on ultra-low power microcontrollers in embedded systems. Our approach introduces alternative regularization functions and a uniform hybrid quantization scheme targeting {1, 2, 4, 8}-bits. The method offers flexibility to the weight matrix level, negligible costs, and seamless integration into existing 8-bit post-training quantization pipelines. Additionally, we propose novel schedule functions for regularization, addressing the critical yet often-overlooked timing aspect and providing new insights into pacing quantization. Our method achieves a substantial reduction in model byte size, nearly halving it with less than 1% accuracy loss, effectively minimizing power and memory footprints on microcontrollers. Our contributions advance resource-efficient models in resource-constrained devices and the emerging field of tinyML, overcoming limitations of existing approaches and providing new

perspectives on the quantization process. The practical implications of our work span diverse real-world applications, including IoT, wearables, and autonomous systems.

This chapter is based on a pending patent (de Foras and Lê, 2022), which was presented as a poster at tinyML EMEA Innovation Forum 2022 (Lê and de Foras, 2022), and a paper (Lê et al., 2023):

Lê, M. T. and de Foras, E. (2022). Towards Universal 1-Bit Weight Quantization of Neural Networks with end-to-end deployment on ultra-low power sensors. *tinyML EMEA Innovation Forum 2022*.

Lê, M. T., de Foras, E., and Arbel, J. (2023). Regularization for hybrid N-bit weight quantization of neural networks on ultra-low power microcontrollers. In Iliadis, L., Papaleonidas, A., Angelov, P., and Jayne, C., editors, *Artificial Neural Networks and Machine Learning – ICANN 2023*, Cham. Springer Nature Switzerland.

de Foras, E. and Lê, M. T. (2022). One bit quantization for embedded systems. *Pending patent*.

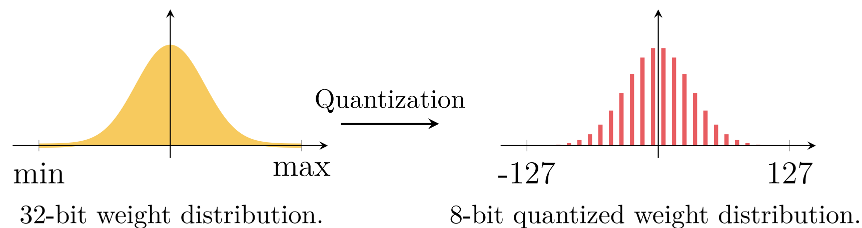
4.1 Introduction

Neural networks and resource-constrained devices

As neural networks (NNs) continue to deliver impressive results across a wide range of domains, they are primarily powered by highly over-parameterized models, which come with a substantial power consumption cost. This energy barrier inadvertently restricts their deployment in resource-constrained applications, such as ultra-low power microcontroller units (MCUs), like Arm Cortex-M0+ and M4 (Unlu, 2020). MCUs are integral components of embedded systems commonly found in wearables and IoT devices. These systems demand efficient neural network models that can operate within their limited memory (as low as 8 KB), computational capacity (as low as to 6 MHz), and strict power constraints that must run always-on, in real-time, for months or even years. The need for innovative solutions for compressing and deploying neural networks on resource-constrained embedded devices emerged into the field of tinyML (Lê and Arbel, 2023).

Quantization

TinyML necessitates the development of novel quantization methods that retain the flexibility of existing approaches while achieving better trade-offs between model size, accuracy, and efficiency for resource-constrained applications. Quantization is a widely-used technique for compressing deep neural network models by converting their floating-point parameters into lower-bit precision integers, as illustrated in the following diagram. It has become an essential method for meeting the requirements of MCUs that perform inference based on fixed-point arithmetic (Gholami et al., 2021). Uniform symmetric post-training quantization (PTQ) at 8-bits is a prevalent approach due to its model-agnosticism, simplicity, and broad embedded hardware support with acceptable performance loss (Zmora et al., 2019, Gholami et al., 2021). However, it suffers from sensitivity to outliers and performs poorly at low-bit precision, such as binary or ternary networks (Zmora et al., 2019).



32-bit weight distribution. 8-bit quantized weight distribution.

Contributions

In this paper, we propose a novel regularized quantization method that enables hybrid quantization at the weight matrix level across various combinations of bit precision: $\{1, 2, 4, 8\}$-bits. Our approach introduces alternative 1-bit regularization functions for quantization based on the work of Darabi et al. (2020), and generalizes them to $\{2, 4, 8\}$-bits regularization functions.

By building upon PTQ, our method maintains its flexibility and compatibility with existing end-to-end pipelines already using 8-bit quantization, specifically targeting resource-constrained devices. Additionally, our approach offers a uniform hybrid quantization scheme across $\{1, 2, 4, 8\}$-bits, extendable to any bit precision, thereby reducing the model inference power footprint on MCU platforms. Furthermore, we introduce a scheduled regularization technique for progressive quantization, offering crucial insights on the often-overlooked but critical timing aspect of regularization (Golatkar et al., 2019) in the case of quantization. Our primary contributions are summarized as follows:

1. We propose new regularization functions for 1-bit quantization and their generalization to $\{2, 4, 8\}$-bits , extendable to any other bit-precision;

2. We introduce a hybrid quantization scheme at the weight matrix level, designed to reduce the model power footprint for ultra-low power MCUs;

3. We present a novel scheduled regularization approach for progressive weight quantization to $\{1, 2, 4, 8\}$-bits;

4. We conduct comprehensive experiments demonstrating the effectiveness of our method across various settings, including activation, regularization, schedule, layer combination, and size. These experiments provide new insights on how to use and pace regularization with schedule functions for quantization.

Related work

Prior works have focused on quantization-aware training approaches (QAT) using the straight-through estimator for binary or ternary networks (Courbariaux et al., 2015, Zhu et al., 2017), or other approximations (Yin et al., 2018, Louizos et al., 2019) to simulate the quantization effect during training. Other approaches introduce an additional term to the objective function (Tang et al., 2017, Darabi et al., 2020) of the neural network to encourage weights to converge to a set of points during training, minimizing the loss of accuracy during PTQ. Alternative approaches have used knowledge distillation (Alemdar

et al., 2017) or Bayesian methods (Van Baalen et al., 2020) to learn lower-bit precision networks. Moreover, previous studies have shown that lower bit-precision models can achieve better performance by utilizing PReLU activation functions in CNN models (Bulat et al., 2019, Kim et al., 2021).

Regularization techniques are widely used to improve the generalization performance of neural networks. Various implicit methods include dropout, early stopping, data augmentation (Shorten and Khoshgoftaar, 2019) or SGD (Zhang et al., 2017), while explicit regularization involves the L_2 penalty. Additionally, the L_1 regularization can be used to encourage weight pruning. Regularization can also be employed for quantization, as demonstrated in Darabi et al. (2020), where regularization functions are used to quantize models down to 1-bit using the following regularization functions

$$\mathrm{R}_1(w) = |\alpha - |w||, \mathrm{R}_1^2(w) = (\alpha - |w|)^2, \tag{4.1}$$

with $\alpha = 1$, depicted in Figure 4.2.2. They achieve acceptable results on ImageNet close to the full precision model.

The early training stages are the most critical for convergence, so introducing an explicit high-penalty regularization term for quantization at the beginning can negatively impact and disrupt the gradient optimization (Achille et al., 2019, Frankle et al., 2020). Moreover, Golatkar et al. (2019) discovered that when using regularization, the initial phase is the most crucial and sensitive period for performance. They also show that starting with late regularization does not improve generalization, and can even worsen the outcome in some cases. They conclude that the role of regularization is to steer the initial transient towards regions of the loss with multiple equivalent solutions, rather than bias the final solution towards critical points. Our study fills a gap in the literature by exploring the effect of transient dynamics on regularization for quantization, an aspect that has not been previously investigated.

In the following sections, we present our proposed method encompassing generalized regularization functions, a hybrid quantization scheme, and a scheduled regularization approach (Section 4.2), experimental setup (Section 4.3), and results demonstrating the efficacy of our approach (Section 4.4).

4.2 Proposed method

In this section, we introduce our regularization approach for hybrid {1, 2, 4, 8}-bits quantization of neural networks at the weight matrix level, illustrated in Figure 4.2.1,

using a convolutional neural network (CNN), a gated recurrent unit (GRU) model, with a fully-connected (FC) layer as an example.

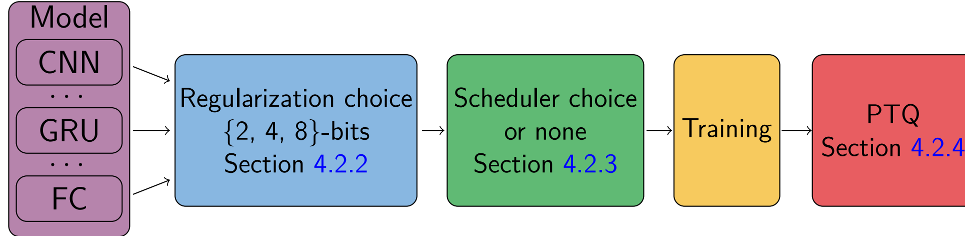


Fig. 4.2.1 Regularized quantization pipeline using hybrid {1, 2, 4, 8}-bits quantization with schedule option using a convolutional (CNN) GRU model with a fully-connected (FC) layer as example.

The explicit regularization adds a term to the loss function L in the objective function J as follows (Darabi et al., 2020):

$$J(W, x, y) = L(f(x, W), y) + \lambda \sum_{i=1}^{M} \text{Reg}(W_i), \tag{4.2}$$

where, f is the neural network with weights W, λ controls how much the weights are penalized by the regularizer function Reg, and M is the number of weight matrices to quantize. Notice that in the notation $\text{Reg}(W_i)$, the function Reg is applied element-wise to the weight entries of W_i.

The proposed following regularizations in Section 4.2.1 and 4.2.2 are fully differentiable as well as model-agnostic and enable full or hybrid quantization at the weight matrix level, by selecting the number of bits N. Thus, it can easily be integrated into any training procedure and will be transparent to any embedded deployment process. Moreover, the regularization optimization is executed in conjunction with the training process, resulting in minimal computational overhead.

4.2.1 1-bit regularization

We propose three new regularizers for 1-bit quantization having global minima at -1 and $+1$ and local maximum in zero as illustrated in Figure 4.2.2:

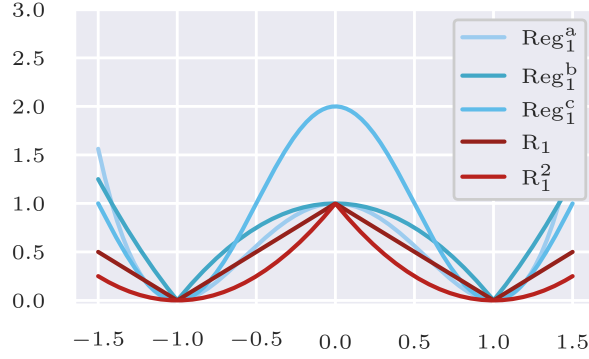


Fig. 4.2.2 1-bit regularization functions of Equations (4.1), (4.3), (4.4), and (4.5).

$$\mathrm{Reg}_1^a(w) = (w^2 - 1)^2, \tag{4.3}$$

$$\mathrm{Reg}_1^b(w) = |w^2 - 1|, \tag{4.4}$$

$$\mathrm{Reg}_1^c(w) = \begin{cases} \cos(\pi w) + 1 & \text{if } |w| \leq 3/2, \\ \pi(|w| - 3/2) + 1 & \text{if } |w| > 3/2. \end{cases} \tag{4.5}$$

Function Reg_1^c is defined in such a way that it is continuously differentiable ($\mathcal{C}^1$) on $\mathbb{R}$. Such a regularization forces the weights to converge to a discrete set of values during training while minimizing the function J. We can characterize each minima pits by its angle around the global minimum and the curvature (convex, concave) between the minima and 0. A shallower angle or a concave curve gives more room for the model to learn since it has a gentler impact on the quantization penalty around the minima than a steeper angle. This results in a greater variance around the minima and a higher chance of experiencing PTQ accuracy loss. On the contrary, a steeper angle or a convex curve may impede the network's learning ability due to a higher penalty but leads to better quantization convergence, so there is a tradeoff between minimizing PTQ accuracy loss and learning performance.

We can then rank the regularization by their angle from shallower to steeper angle: $\mathrm{R}_1^2 < \mathrm{Reg}_1^a < \mathrm{Reg}_1^c < \mathrm{R}_1 < \mathrm{Reg}_1^b$.

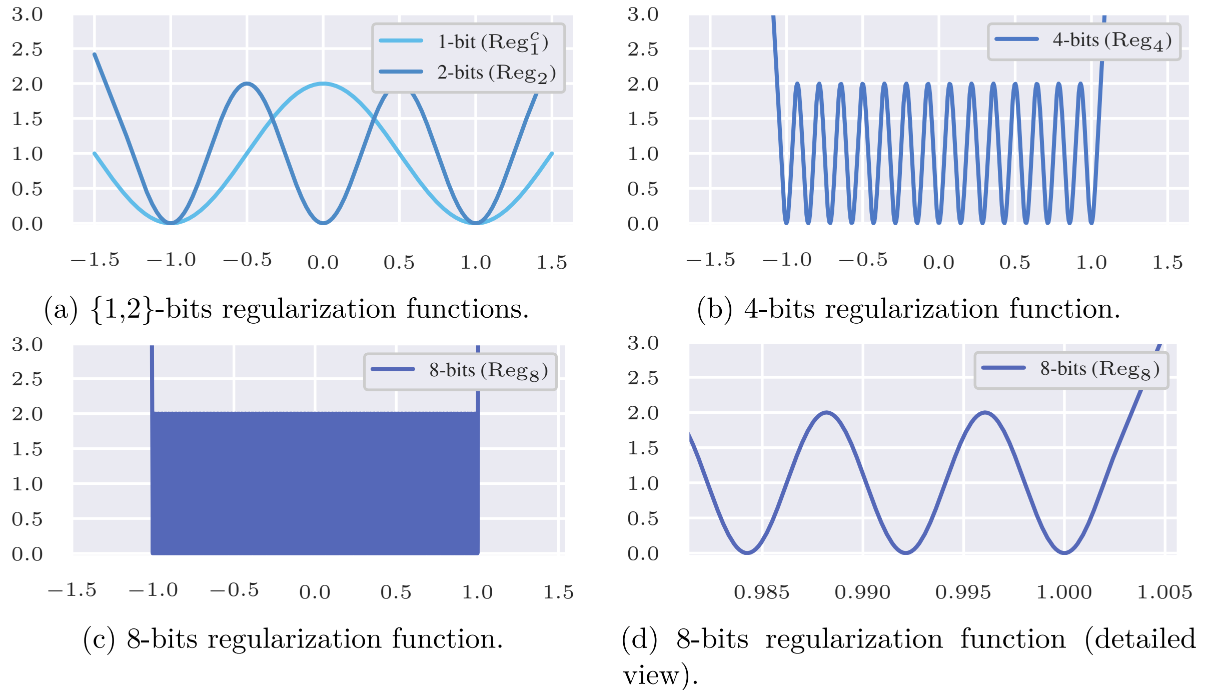
4.2.2 N-bit generalization

We then generalize our regularization functions for quantization Reg_N for $N \in \{2, 4, 8\}$ as follows:

$$\text{Reg}_N(w) = \begin{cases} 1 - \cos\left(2\pi(2^{N-1} - 1)w\right) & \text{if } |w| \leq 1 + \frac{1}{4(2^{N-1}-1)} \\ 1 + 2\pi\left((|w| - 1)(2^{N-1} - 1) - \frac{1}{4}\right) & \text{if } |w| > 1 + \frac{1}{4(2^{N-1}-1)} \end{cases} \tag{4.6}$$

These functions are depicted in Figure 4.2.3. They are all based on the cosine function with a frequency that matches the cardinality of unique values in N-bits between $[-1; +1]$ and also has a minimum in 0. As observed in Figure 4.2.3, a higher frequency leads to steeper minima angles. All Reg_N are continuously differentiable ($\mathcal{C}^1$) on $\mathbb{R}$. For any $N \in \{2, 4, 8\}$, function Reg_N has $2^N - 1$ global minima defined by:

$$\left\{\frac{k}{2^{N-1} - 1}, k \in \mathbb{Z}, |k| \leq 2^{N-1} - 1\right\}. \tag{4.7}$$



(a) {1,2}-bits regularization functions.

(b) 4-bits regularization function.

(c) 8-bits regularization function.

(d) 8-bits regularization function (detailed view).

Fig. 4.2.3 N-bit regularization functions of Equation (4.5) and Equation (4.6), $N \in \{1, 2, 4, 8\}$.

4.2.3 Scheduled regularized quantization

The high density of global minima, especially in the 8-bit regularization in Figure 4.2.3c, may cause high gradient spikes throughout training and prevent convergence, or drive the optimizer toward a suboptimal region of the loss early on.

To overcome this issue, we employ a scheduled regularization scheme (Golatkar et al., 2019) that progressively increases the quantization penalty over time, as depicted in Figure 4.2.4. Thus, this allows more time for the weights to properly converge before being trapped by incremental regularization in the global minima, as shown in Figure 4.2.5.

A fast schedule function has similar effects to a steep minima angle: weights should converge faster towards their quantized values, but it may hinder the learning process if the penalty is excessive. Conversely, starting with a slower pace allows more time for the model to learn; however, if the last period is too accelerated, it might cause abrupt gradient surges, negatively impacting the performance.

The schedule function S is scaled to the quantized regularization over time during training as

$$\text{Reg}(W, t) = \text{Reg}(W)S(t). \tag{4.8}$$

This also has the effect of increasing the angle steepness over time, which traps weights to a minimum. Schedule functions $S(\cdot)$ need to satisfy the following properties:

$$S(t = 1) \approx 0 \text{ and } \geq 0, \quad S(t = t_{\text{final}}) \leq 1, \quad S'(t) \geq 0.$$

We then define eight schedule functions in Table 4.2.1 depicted in Figure 4.2.4.

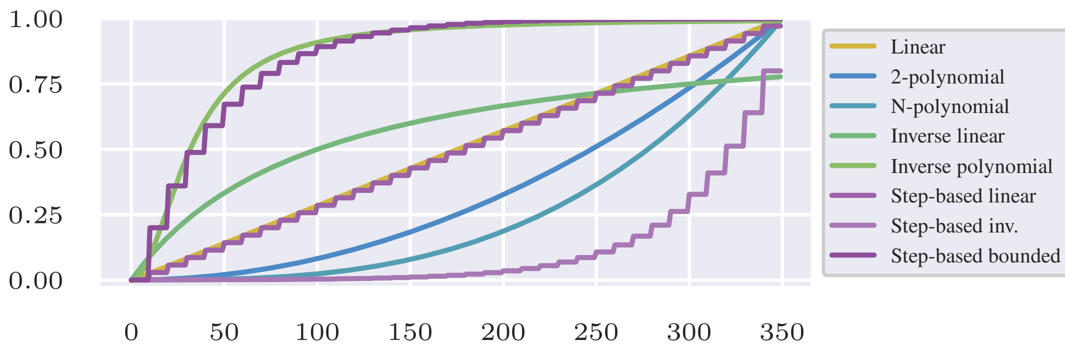


Fig. 4.2.4 Schedule functions for quantized regularization for $t_{\text{final}} = 350$.

Figure 4.2.5 demonstrates the quantization convergence of weights over epochs using a linear schedule. We notice that the weight distribution becomes more and more discrete and converges towards Reg_4 minima, so we can minimize the loss of PTQ accuracy after rounding the weights.

Table 4.2.1 Schedule functions for regularization. Here, d is set to 0.01 and 0.001 for schedule inverse linear and inverse polynomial respectively, and rate is set to 0.8 by default. For step-based functions, step is set to 10 by default. For step-based inverse, a higher rate leads to a faster schedule, contrary to step-based bounded.

Scheduler name	Function
Linear	$S(t, t_{\text{final}}) = \frac{t-1}{t_{\text{final}}-1}$
2-polynomial	$S(t, t_{\text{final}}) = \frac{t^2-1}{t_{\text{final}}^2-1}$
N-polynomial	$S(t, t_{\text{final}}, N) = \left(\frac{t}{t_{\text{final}}}\right)^N$
Inverse linear	$S(t, d) = \frac{1}{1+\frac{1}{dt}}$
Inverse polynomial	$S(t, d) = \frac{1}{1+\frac{1}{dt^2}}$
Step-based linear	$S(t, t_{\text{final}}, \text{step}) = \frac{\lfloor \frac{t}{\text{step}} \rfloor}{\lfloor \frac{t_{\text{final}}}{\text{step}} \rfloor}$
Step-based inverse	$\text{scale}(t_{\text{final}}, \text{step}, \text{rate}) = \frac{1}{\text{rate}^{\lfloor \frac{t_{\text{final}}}{\text{step}} \rfloor}}$
	$S(t, \text{step}, \text{rate}) = \frac{1}{\text{scale} \times \text{rate}^{\lfloor \frac{t}{\text{step}} \rfloor}}$
Step-based bounded	$S(t, \text{step}, \text{rate}) = 1 - \text{rate}^{\lfloor \frac{t}{\text{step}} \rfloor}$

4.2.4 Post-training quantization

We choose the simple symmetric linear min-max as the PTQ algorithm for its ease of implementation and seamless deployment to low-power embedded devices, as well as reasonable performance results in most scenarios (Zmora et al., 2019, Gholami et al., 2021).

The model weights are quantized in two straightforward steps. First, we round the regularized weights to their nearest global minima with respect to their regularization function ({1, 2, 4, 8}-bits), and quantize them to int-N integers. Finally, we quantize the remaining weights to 8-bit integers, and bias and activations are quantized to 32-bit integers.

4.3 Experiments

This section details the experimental design and choices to extensively evaluate our proposed method for regularized quantization on two classification tasks.

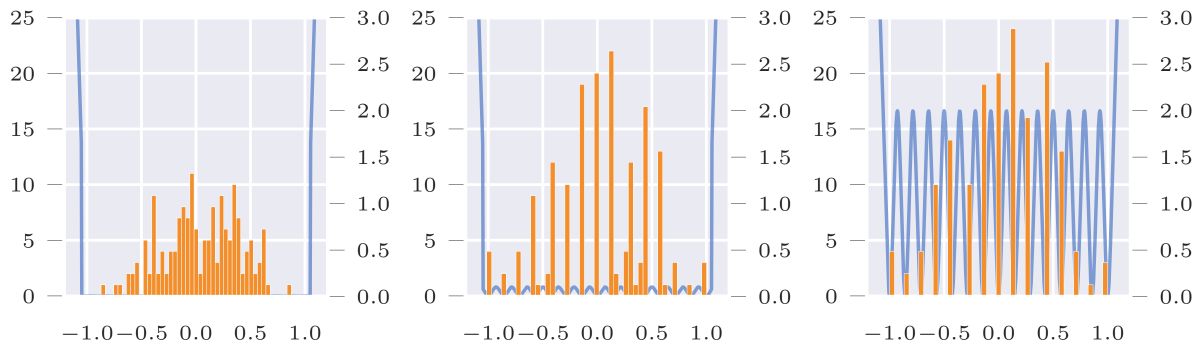


Fig. 4.2.5 Scheduled regularization of GRU weights at epoch 1, 25, and 500 (left to right) to 4-bits with Reg_4 regularization and linear schedule in the background, on the BTS dataset.

4.3.1 Datasets

Bring-to-see

We experimented with a bring-to-see (BTS) dataset, which refers to the binary detection of the motion of a user's wrist toward their face as a means of waking a watch or mobile device. We have raw data from a 3-axis accelerometer at 20 Hz as input, and each gesture sample is 2 seconds. Thus, each input is shaped as 40 sequence length, 3 features (x, y, z accelerometer axis).

Google Speech Commands v2-12

We also tested the Google speech commands v2-12 (GSCv2-12) dataset (Warden, 2018) consisting of 12 keyword spotting words. The audio samples are pre-processed with MFCC and have a shape of 94 sequence length, 13 features.

4.3.2 Model

We use a convolutional RNN and specifically convolutional GRU because of their size-performance tradeoff (Arik et al., 2017) and standard usage for sequence classification. They are also compatible with causal architectures, which is critical for real-time inference (Takeuchi et al., 2020). We also use batch normalization, dropout, and spatial dropout (Tompson et al., 2015) layers. Additionally, we evaluate the PReLU activation over the ReLU (Bulat et al., 2019, Kim et al., 2021). We refer to the baseline model as the configuration in Table 4.3.1. For the BTS dataset and GSCv2-12 dataset, each model is trained for 500 and 350 epochs, respectively using the Nadam optimizer with default parameter on TensorFlow.

Table 4.3.1 Baseline model configuration size and parameters where (c, k) refers to the number of channels and the kernel size of the CNN1D. Weights are stored in 8-bits and activations in 32-bits integers.

Layer	Configuration		Parameters		Size (B)	
	BTS	GSCv2-12	BTS	GSCv2-12	BTS	GSCv2-12
CNN1D (c, k)	(6, 5)	(128, 10)	96	16,768	114	17,512
Batchnorm	✓		24	512	96	2,048
Activation	ReLU/PReLU		c if PReLU else 0		$c \times 4$ if PReLU else 0	
SpatialDropout	✓		✗		✗	
GRU	5	196	195	191,688	285	195,216
Dropout	✓		✗		✗	
Batchnorm	✗	✓	✗	784	✗	3,136
FC	2	12	12	2,364	18	2,400
Total			327(+6)	212,116(+128)	513(+24)	219,952(+512)

4.3.3 Experimental design

We aim to address the following questions through our experiments:

Q1. Which regularization functions are optimal for 1-bit, and when?

Q2. Where to apply {1, 2, 4, 8}-bits quantization, and when?

Q3. Is PReLU superior to ReLU for N-bit quantization?

Q4. Is scheduling beneficial to regularization? When is scheduling versus non-scheduling most effective? Is a faster schedule better than a slower one?

Q5. Can training a larger model for 1-bit quantization compensates for the performance gap between the 8-bit baseline and its 1-bit counterpart? Which layer sizes are most appropriate?

We experiment with our proposed method and provide answers to the previous questions through making two key components vary: the regularization design and the model configuration. The next sections elaborate on the details of the extensive experimental scenarios. Each training is replicated five times independently to test the robustness of our approach.

Regularization configuration

Regularization choice. We compare our alternative 1-bit regularization function (Equation (4.3), (4.4), and (4.5)) in Section 4.2.1, against the regularizations in Equation

(4.1) from Darabi et al. (2020) with $\alpha = 1$. Similarly, we also test our {2, 4, 8}-bits regularizations in the same hybrid settings.

Scheduler choice. We also assess the effectiveness of the schedule functions on generalization and quantization convergence by comparing scenarios with and without a schedule for each regularization choice.

Model configuration

Hybrid quantization. We evaluate the effects of quantization on various layer combinations: convolution, GRU, both, or both, and the output layer, which we refer to as "all layers". The remaining layers are quantized to 8-bits, as described in Section 4.2.4, enabling hybrid quantization. The hybrid configuration is annotated as {N, 8}-bits. To minimize the number of combinations, we decided to regularize the two weight matrices of the GRU layer at the same bit precision, even if regularization allows for weight-wise quantization granularity. We also limit the hybrid configuration to two different bit precisions for each model, even though our method allows models to be quantized to any mix of bit precision, e.g., {1,4,8}-bit model.

Activation choice. In addition, we test the ReLU versus PReLU performance as the convolutional activation as stated in Section 4.3.2.

Layer size choice. Finally, for 1-bit quantization and each hybrid quantization configuration listed above, we sample 4 to 7 model configurations that have a smaller memory footprint compared to the 8-bit baseline, but a larger GRU or CNN size than the baseline configuration. E.g., a binary-only 1D CNN model with 11 filters, kernel size of five, ReLU activation, and a two-unit GRU is smaller in memory than the baseline but has more CNN channels, so it might be able to compensate for having 1-bit weights. Note that, we choose to only perform this experiment on the BTS dataset as it is smaller. Model configuration details can be found in Appendix 4.3.1.

4.3.4 Deployment and evaluation

The neural networks are deployed and evaluated by accuracy on low-power embedded sensors for real-time inference using Arm Cortex-M0+ and M4 MCU (Unlu, 2020) for the BTS and GSCv2-12 datasets, respectively. The accuracy of the model was evaluated by selecting the best model on the validation set from five independent training repetitions for all experiments.

4.4 Results

We present our results and findings in the following sections, in the same order as presented above in Section 4.3.

4.4.1 Regularization configuration

Regularization choice

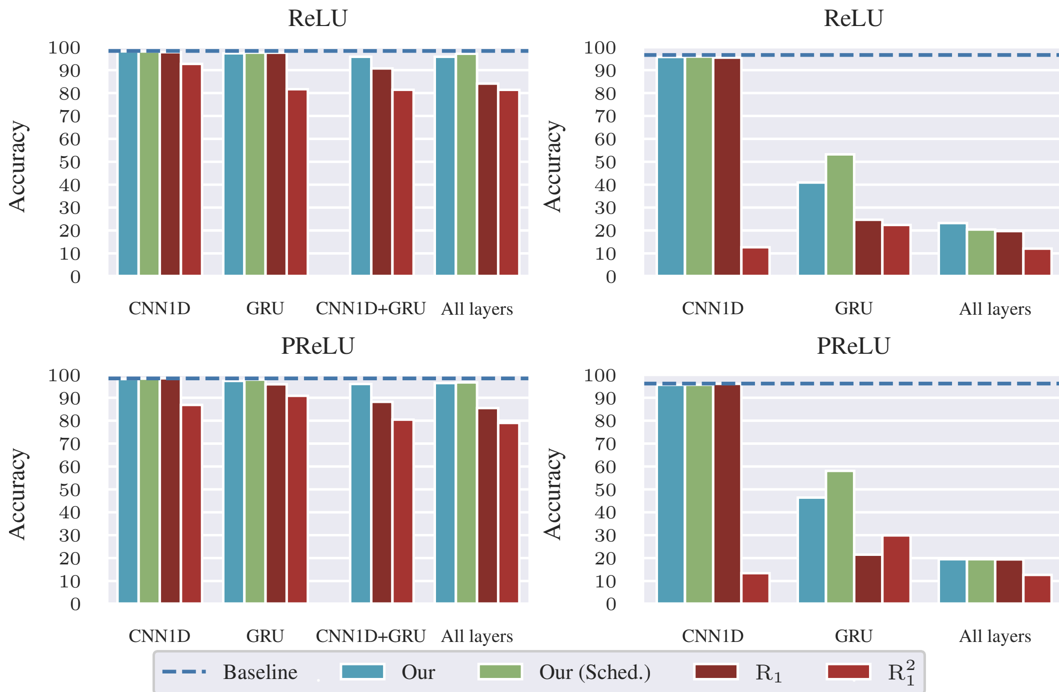


Fig. 4.4.1 Hybrid {1, 8}-bits quantization test accuracy results on the BTS (left) and GSCv2-12 dataset (right), without schedule (blue) or with schedule (green) (Section 4.2.3). The baseline is trained normally without regularization and quantized to 8-bits. For BTS, we exclude training with schedulers for CNN1D+GRU. For GSCv2-12, we exclude training with CNN1D+GRU.

1-bit. In Figure 4.4.1, our regularization consistently surpasses other alternatives in the majority of scenarios, except when considering CNN1D with PReLU activation, where R_1 achieves the best results. However, our solution still delivers competitive results in that scenario. R_1^2 frequently displays the weakest performance across all scenarios. It may be attributed to R_1^2 having the shallowest curve angle, leading to more lenient regularization for quantization convergence, and consequently suffering from greater

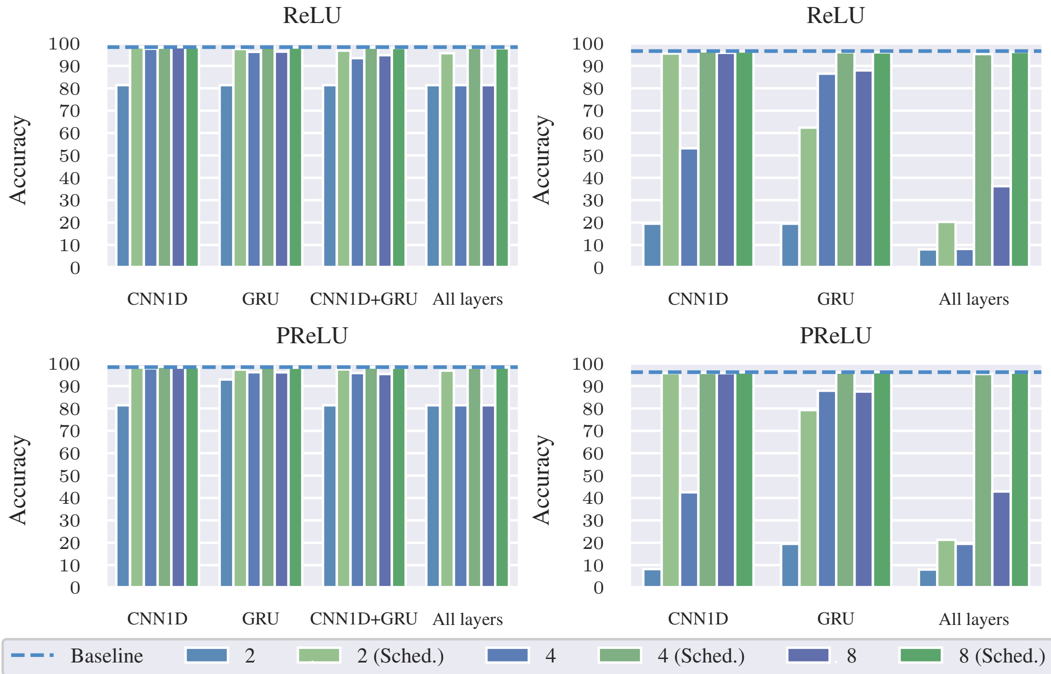


Fig. 4.4.2 Hybrid {2, 4, 8}-bit quantization test accuracy results on the BTS (left) and GSCv2-12 dataset (right), without schedule (blue) or with schedule (green). The baseline is trained without regularization and quantized to 8-bits.

PTQ performance degradation. In contrast, all steeper-angled regularization functions outperform R_1^2 in nearly all scenarios, addressing Q1. Overall, we observe that utilizing a scheduler contributes to slightly improved results compared to its absence, as observed in both datasets, shedding light on Q4.

***N*-bit.** Figure 4.4.2 demonstrates the effectiveness of our N-bits regularization approach for model quantization on {4,8}-bits, especially in the CNN1D case, addressing Q2. In particular, the 4-bit and 8-bit with scheduling outperform other configurations across all scenarios. However, in most cases, 2-bit performance falls behind that of 4-bits and 8-bits, except for CNN1D with scheduling, offering some insights for Q2 and Q4.

Scheduler choice

To schedule or not to schedule? As discussed in Section 4.4.1, incorporating scheduling with regularization improves the quantization outcomes compared to not using scheduling in most cases, particularly in the most demanding scenario where all layers are quantized, and with higher bit precision like {4,8}-bits, across both datasets.

This indicates that progressive quantization does foster a more favorable environment for optimization, addressing Q4.

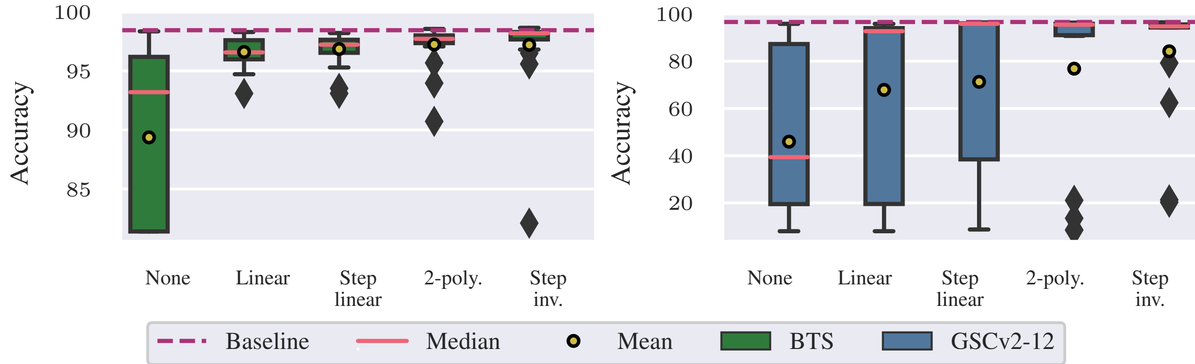


Fig. 4.4.3 Scheduler comparison of our 1-bit quantization results using our 1-bit (top) and {2, 4, 8}-bits (bottom) regularization, on the BTS (left) and GSCv2-12 (right) dataset, across all scenarios. Schedulers are ordered from fastest to slowest at mid-training time (as shown in Figure 4.2.4). The baseline is trained without regularization and quantized to 8-bits. For {2, 4, 8}-bits, inverse polynomial and step-based bounded are omitted from experiments.

Finding the right pace. In Figure 4.4.3, when considering median and mean accuracy, for 1-bit quantization, faster or linear-paced schedules are generally more effective than slower schedules on both datasets. Notably, the case without a schedule is the fastest, as it entirely omits progressive regularization. In contrast, slower-paced schedules demonstrate better performance for {2, 4, 8}-bits quantization. This implies that higher bit precision requires lower regularization penalties over time for effective PTQ.

These insights respond to Q4 and emphasize the importance of selecting a suitable scheduler for different quantization settings and highlight the need for further investigation into customizing scheduling approaches based on the dataset and model characteristics to achieve optimal results.

4.4.2 Model configuration

Hybrid quantization

In relation to Q2, quantizing the CNN1D layer is robust in all cases, even as low as 1-bit. However, the GRU layer is more sensitive to quantization as it is larger. 1-bit quantization yields a minor loss on BTS but is damaging on GSCv2-12. Scheduled quantization on the GRU layer at {2, 4, 8}-bits provides acceptable results on BTS.

However, only {4, 8}-bits is suitable for GSCv2-12 and a 2-bit quantization without a scheduler is overly detrimental in all cases.

Applying quantization to both CNN1D and GRU layers at 1-bit yields less favorable results, although {4, 8}-bits with a schedule provide acceptable performance. Quantizing all layers to 1-bit leads to a slight decline in performance on BTS but a more severe impact on GSCv2-12, while applying {4, 8}-bits quantization with a schedule yields less than a 1% loss for both datasets.

Activation choice: ReLU vs PReLU

In response to Q3, we observe that in the {2, 4, 8}-bit range, PReLU generally performs better than ReLU (Figure 4.4.2), while ReLU still achieves close results under the best regularization scenarios. In contrast, at 1-bit, the performance of PReLU is comparable to ReLU in our regularization.

Layer size choice

Figure 4.4.4 demonstrates that it is consistently possible to compensate for the performance loss of 1-bit quantization by training a larger model with a smaller byte size than the 8-bit baseline (6 CNN channels, 5 cells), except when quantizing GRU with ReLU, responding to Q5.

Specifically, adding both more CNN channels and GRU cells (i.e., the diagonal cell relative to the baseline) generally yields better results than increasing only CNN channels or GRU cells. However, when quantizing both CNN1D and GRU layers, adding more GRU cells is slightly more advantageous. Furthermore, having minimum GRU cells and maximum CNN filters result in poor performance. Lastly, increasing GRU cells is more beneficial than adding more CNN filters when quantizing all layers.

4.4.3 Key results

Our findings reveal that employing 4-bit quantization in conjunction with an appropriate scheduler results in the most advantageous size-performance tradeoff across all cases (Q2, Q3). In particular, the PReLU activation emerges as a superior option when considering {2, 4, 8}-bit precision (Q3). Additionally, by increasing the size of critical layers collectively, we can quantize models with reduced byte size without sacrificing performance (Q5).

Our method achieves substantial byte size reductions in baseline models (Table 4.3.1) when all layers are quantized to 4-bits: approximately 25.8% reduction with ReLU on

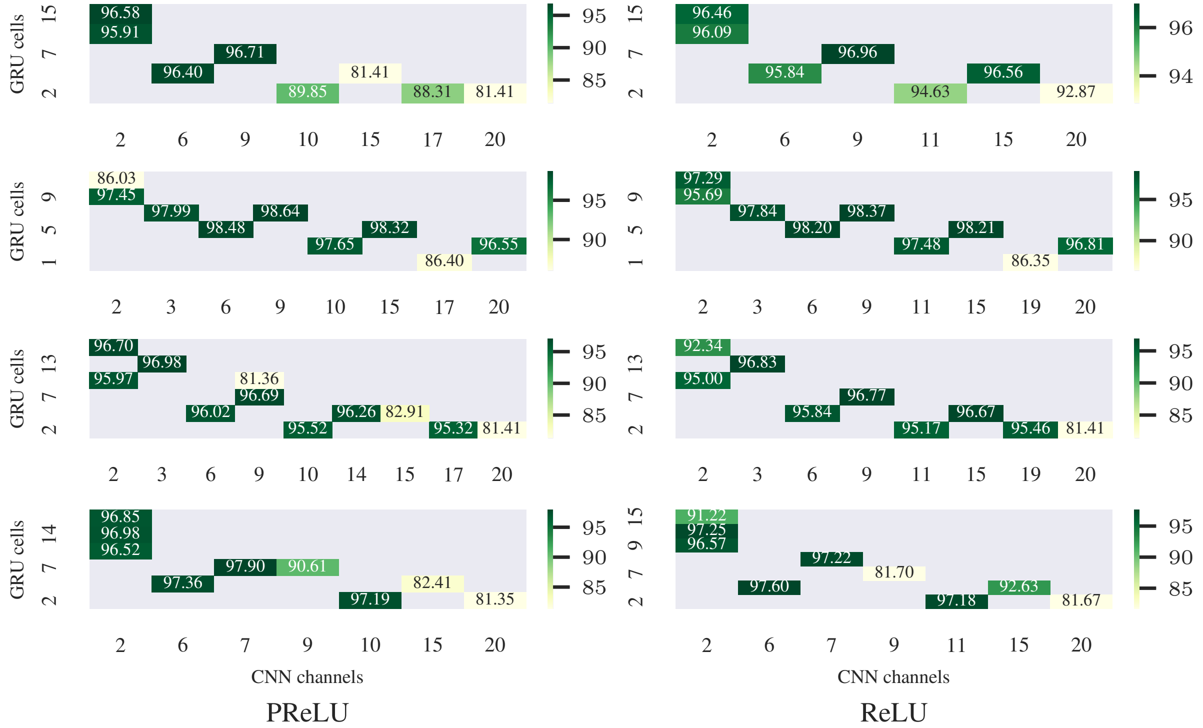


Fig. 4.4.4 Hybrid {1, 8}-bits weight quantization accuracy versus the number of CNN channels and GRU cells for various combinations of quantized layers, with ReLU or PReLU as CNN activation on the BTS dataset. 1st row: All layers, 2nd: CNN1D layer only, 3rd: CNN1D and GRU layers, and 4th: GRU layer. For reference, we also added the baseline model architecture at 6 channels and 5 cells with hybrid quantized layers. Darker colors are better. Each model here is trained using our 1-bit regularization (Section 4.2.1) without a schedule.

BTS, with a 0.3% performance loss, and 47.5% reduction on GSCv2-12, with less than a 0.8% accuracy loss. Moreover, employing 1-bit with ReLU and schedule on BTS yields a 45.2% size gain for less than a 1.2% accuracy loss. Furthermore, 2-bit quantization of CNN and GRU on BTS yields a reduction in 37.3% byte size at the cost of a 1.5% performance loss.

For CNN1D, we determined that 1-bit quantization is feasible with a faster schedule, and a steeper minima curve (Q1, Q2, Q4), while still maintaining accuracy. In this scenario, we achieved a byte size gain of approximately 15% on BTS for less than 0.2% performance loss, corresponding to a CNN1D with a weight size of merely 11.25 bytes (compared to the original 90 bytes), which is smaller in memory than the word “quantization”. On GSCv2-12, we obtained a byte size gain of 6.6% on GSCv2-12 with a 0.5% accuracy decrease.

For GRU layer with 1-bit schedule and ReLU on BTS, we attain a size gain of about 28% for less than 0.8% accuracy loss.

In the case of CNN and GRU with ReLU, 2-bit quantization on BTS yields a 37.28% byte size reduction at the cost of a 1.5% performance loss.

However, our method presents certain limitations. For instance, {1,2}-bit quantization may underperform, particularly without schedulers, in large layers like GRU, and on the GSCv2-12 dataset, as illustrated in Figure 4.4.2. Moreover, despite the expectation, 8-bit regularized quantization does not surpass the 8-bit baseline. Additionally, our method requires the consideration of extra hyperparameters.

In summary, our study emphasizes the importance of carefully selecting model hyper-parameters such as activation, regularization, scheduler choice, and layer sizes, on quantization performance. We provide evidence supporting the effectiveness of functional hybrid quantization techniques, especially with 4-bits, both with or without schedulers, in reducing the model size without compromising performance.

4.5 Conclusion

In this work, we introduced a novel regularized quantization method for deep neural networks, enabling uniform hybrid quantization at the weight matrix level across various bit precisions ({1, 2, 4, 8}-bits) in a model-agnostic manner. Our approach introduces alternative 1-bit regularization functions that extend beyond previous studies and presents novel regularization functions for generalizations to {2, 4, 8}-bits. We also proposed a scheduled regularization scheme for quantization, addressing the aspect of time in regularization. Our approach demonstrated the ability to reduce the model byte size by nearly half while maintaining minimal accuracy loss. We showed that our method retains the flexibility benefits of PTQ and seamlessly integrates into any existing training pipelines with minimal computational overhead for resource-constrained devices, providing a uniform hybrid quantization scheme that reduces the model inference power footprint of neural networks. Overall, this work contributes to the expanding field of tinyML, enabling the efficient deployment of deep learning models on ultra-low power MCUs for tinyML applications.

Future work can encompass further exploration of different regularization functions and hybrid combinations to {1, 2, 4, 8}-bits, and extending the method to other deep learning architectures beyond CNNs and GRUs. Investigating methods for reducing activation size from 32-bit integers represents another essential direction for achieving substantial improvements in power efficiency and performance of tinyML systems.

Appendix

4.A Hybrid {1,8}-bit model byte size reference on the BTS dataset

We reference the byte size with regard to different model configuration size (baseline in Table 4.3.1) when quantizing different set of weights to 1-bit and keeping the rest in 8-bit integers. Activations are in 32-bit integers.

The following reference table uses kernel size k=5 by default.

For each following tables, green cells refers to all the models that have a smaller memory size than the baseline or bigger (red cells) and a larger configuration in CNN or GRU size than the baseline (orange cells). We select the yellow cells in this configuration space for sampling.

4.A.1 CNN1D only

Table 4.A.1 Bytes size of the BTS baseline ReLU model with CNN1D weights quantized to 1-bit.

Filters	GRU cells								
	1	2	3	4	5	6	7	8	9
1	61.9	97.9	139.9	187.9	241.9	301.9	367.9	439.9	517.9
2	86.8	127.8	172.8	223.8	280.8	343.8	412.8	487.8	568.8
3	111.6	157.6	205.6	259.6	319.6	385.6	457.6	535.6	619.6
4	136.5	187.5	238.5	295.5	358.5	427.5	502.5	583.5	670.5
5	161.4	217.4	271.4	331.4	397.4	469.4	547.4	631.4	721.4
6	186.3	247.3	304.3	367.3	436.3	511.3	592.3	679.3	772.3
7	211.1	277.1	337.1	403.1	475.1	553.1	637.1	727.1	823.1
8	236.0	307.0	370.0	439.0	514.0	595.0	682.0	775.0	874.0
9	260.9	336.9	402.9	474.9	552.9	636.9	726.9	822.9	924.9
10	285.8	366.8	435.8	510.8	591.8	678.8	771.8	870.8	975.8
11	310.6	396.6	468.6	546.6	630.6	720.6	816.6	918.6	1026.6
12	335.5	426.5	501.5	582.5	669.5	762.5	861.5	966.5	1077.5
13	360.4	456.4	534.4	618.4	708.4	804.4	906.4	1014.4	1128.4
14	385.3	486.3	567.3	654.3	747.3	846.3	951.3	1062.3	1179.3
15	410.1	516.1	600.1	690.1	786.1	888.1	996.1	1110.1	1230.1
16	435.0	546.0	633.0	726.0	825.0	930.0	1041.0	1158.0	1281.0
17	459.9	575.9	665.9	761.9	863.9	971.9	1085.9	1205.9	1331.9
18	484.8	605.8	698.8	797.8	902.8	1013.8	1130.8	1253.8	1382.8
19	509.6	635.6	731.6	833.6	941.6	1055.6	1175.6	1301.6	1433.6
20	534.5	665.5	764.5	869.5	980.5	1097.5	1220.5	1349.5	1484.5

Table 4.A.2 Bytes size of the BTS baseline PReLU model with CNN1D weights quantized to 1-bit.

Filters	GRU cells								
	1	2	3	4	5	6	7	8	9
1	65.9	103.9	147.9	197.9	253.9	315.9	383.9	457.9	537.9
2	94.8	135.8	182.8	235.8	294.8	359.8	430.8	507.8	590.8
3	123.6	167.6	217.6	273.6	335.6	403.6	477.6	557.6	643.6
4	152.5	199.5	252.5	311.5	376.5	447.5	524.5	607.5	696.5
5	181.4	231.4	287.4	349.4	417.4	491.4	571.4	657.4	749.4
6	210.3	263.3	322.3	387.3	458.3	535.3	618.3	707.3	802.3
7	239.1	295.1	357.1	425.1	499.1	579.1	665.1	757.1	855.1
8	268.0	327.0	392.0	463.0	540.0	623.0	712.0	807.0	908.0
9	296.9	358.9	426.9	500.9	580.9	666.9	758.9	856.9	960.9
10	325.8	390.8	461.8	538.8	621.8	710.8	805.8	906.8	1013.8
11	354.6	422.6	496.6	576.6	662.6	754.6	852.6	956.6	1066.6
12	383.5	454.5	531.5	614.5	703.5	798.5	899.5	1006.5	1119.5
13	412.4	486.4	566.4	652.4	744.4	842.4	946.4	1056.4	1172.4
14	441.3	518.3	601.3	690.3	785.3	886.3	993.3	1106.3	1225.3
15	470.1	550.1	636.1	728.1	826.1	930.1	1040.1	1156.1	1278.1
16	499.0	582.0	671.0	766.0	867.0	974.0	1087.0	1206.0	1331.0
17	527.9	613.9	705.9	803.9	907.9	1017.9	1133.9	1255.9	1383.9
18	556.8	645.8	740.8	841.8	948.8	1061.8	1180.8	1305.8	1436.8

4.A.2 GRU only

Table 4.A.3 Bytes size of the BTS baseline ReLU model with GRU weights quantized to 1-bit.

Filters	GRU cells														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	69.8	97.3	125.5	154.5	184.3	214.8	246.0	278.0	310.8	344.3	378.5	413.5	449.3	485.8	523.0
2	105.1	133.0	161.6	191.0	221.1	252.0	283.6	316.0	349.1	383.0	417.6	453.0	489.1	526.0	563.6
3	140.5	168.8	197.8	227.5	258.0	289.3	321.3	354.0	387.5	421.8	456.8	492.5	529.0	566.3	604.3
4	175.9	204.5	233.9	264.0	294.9	326.5	358.9	392.0	425.9	460.5	495.9	532.0	568.9	606.5	644.9
5	211.3	240.3	270.0	300.5	331.8	363.8	396.5	430.0	464.3	499.3	535.0	571.5	608.8	646.8	685.5
6	246.6	276.0	306.1	337.0	368.6	401.0	434.1	468.0	502.6	538.0	574.1	611.0	648.6	687.0	726.1
7	282.0	311.8	342.3	373.5	405.5	438.3	471.8	506.0	541.0	576.8	613.3	650.5	688.5	727.3	766.8
8	317.4	347.5	378.4	410.0	442.4	475.5	509.4	544.0	579.4	615.5	652.4	690.0	728.4	767.5	807.4
9	352.8	383.3	414.5	446.5	479.3	512.8	547.0	582.0	617.8	654.3	691.5	729.5	768.3	807.8	848.0
10	388.1	419.0	450.6	483.0	516.1	550.0	584.6	620.0	656.1	693.0	730.6	769.0	808.1	848.0	888.6
11	423.5	454.8	486.8	519.5	553.0	587.3	622.3	658.0	694.5	731.8	769.8	808.5	848.0	888.3	929.3
12	458.9	490.5	522.9	556.0	589.9	624.5	659.9	696.0	732.9	770.5	808.9	848.0	887.9	928.5	969.9
13	494.3	526.3	559.0	592.5	626.8	661.8	697.5	734.0	771.3	809.3	848.0	887.5	927.8	968.8	1010.5
14	529.6	562.0	595.1	629.0	663.6	699.0	735.1	772.0	809.6	848.0	887.1	927.0	967.6	1009.0	1051.1

Table 4.A.4 Bytes size of the BTS baseline PReLU model with GRU weights quantized to 1-bit.

Filters	GRU cells															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	73.8	101.3	129.5	158.5	188.3	218.8	250.0	282.0	314.8	348.3	382.5	417.5	453.3	489.8	527.0	565.0
2	113.1	141.0	169.6	199.0	229.1	260.0	291.6	324.0	357.1	391.0	425.6	461.0	497.1	534.0	571.6	610.0
3	152.5	180.8	209.8	239.5	270.0	301.3	333.3	366.0	399.5	433.8	468.8	504.5	541.0	578.3	616.3	655.0
4	191.9	220.5	249.9	280.0	310.9	342.5	374.9	408.0	441.9	476.5	511.9	548.0	584.9	622.5	660.9	700.0
5	231.3	260.3	290.0	320.5	351.8	383.8	416.5	450.0	484.3	519.3	555.0	591.5	628.8	666.8	705.5	745.0
6	270.6	300.0	330.1	361.0	392.6	425.0	458.1	492.0	526.6	562.0	598.1	635.0	672.6	711.0	750.1	790.0
7	310.0	339.8	370.3	401.5	433.5	466.3	499.8	534.0	569.0	604.8	641.3	678.5	716.5	755.3	794.8	835.0
8	349.4	379.5	410.4	442.0	474.4	507.5	541.4	576.0	611.4	647.5	684.4	722.0	760.4	799.5	839.4	880.0
9	388.8	419.3	450.5	482.5	515.3	548.8	583.0	618.0	653.8	690.3	727.5	765.5	804.3	843.8	884.0	925.0
10	428.1	459.0	490.6	523.0	556.1	590.0	624.6	660.0	696.1	733.0	770.6	809.0	848.1	888.0	928.6	970.0
11	467.5	498.8	530.8	563.5	597.0	631.3	666.3	702.0	738.5	775.8	813.8	852.5	892.0	932.3	973.3	1015.0
12	506.9	538.5	570.9	604.0	637.9	672.5	707.9	744.0	780.9	818.5	856.9	896.0	935.9	976.5	1017.9	1060.0
13	546.3	578.3	611.0	644.5	678.8	713.8	749.5	786.0	823.3	861.3	900.0	939.5	979.8	1020.8	1062.5	1105.0

4.A.3 CNN1D+GRU only

Table 4.A.5 Bytes size of the BTS baseline ReLU model with CNN1D+GRU weights quantized to 1-bit.

Filters	GRU cells															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	56.6	84.1	112.4	141.4	171.1	201.6	232.9	264.9	297.6	331.1	365.4	400.4	436.1	472.6	509.9	547.9
2	78.9	106.8	135.4	164.8	194.9	225.8	257.4	289.8	322.9	356.8	391.4	426.8	462.9	499.8	537.4	575.8
3	101.1	129.4	158.4	188.1	218.6	249.9	281.9	314.6	348.1	382.4	417.4	453.1	489.6	526.9	564.9	603.6
4	123.4	152.0	181.4	211.5	242.4	274.0	306.4	339.5	373.4	408.0	443.4	479.5	516.4	554.0	592.4	631.5
5	145.6	174.6	204.4	234.9	266.1	298.1	330.9	364.4	398.6	433.6	469.4	505.9	543.1	581.1	619.9	659.4
6	167.9	197.3	227.4	258.3	289.9	322.3	355.4	389.3	423.9	459.3	495.4	532.3	569.9	608.3	647.4	687.3
7	190.1	219.9	250.4	281.6	313.6	346.4	379.9	414.1	449.1	484.9	521.4	558.6	596.6	635.4	674.9	715.1
8	212.4	242.5	273.4	305.0	337.4	370.5	404.4	439.0	474.4	510.5	547.4	585.0	623.4	662.5	702.4	743.0
9	234.6	265.1	296.4	328.4	361.1	394.6	428.9	463.9	499.6	536.1	573.4	611.4	650.1	689.6	729.9	770.9
10	256.9	287.8	319.4	351.8	384.9	418.8	453.4	488.8	524.9	561.8	599.4	637.8	676.9	716.8	757.4	798.8
11	279.1	310.4	342.4	375.1	408.6	442.9	477.9	513.6	550.1	587.4	625.4	664.1	703.6	743.9	784.9	826.6
12	301.4	333.0	365.4	398.5	432.4	467.0	502.4	538.5	575.4	613.0	651.4	690.5	730.4	771.0	812.4	854.5
13	323.6	355.6	388.4	421.9	456.1	491.1	526.9	563.4	600.6	638.6	677.4	716.9	757.1	798.1	839.9	882.4
14	345.9	378.3	411.4	445.3	479.9	515.3	551.4	588.3	625.9	664.3	703.4	743.3	783.9	825.3	867.4	910.3
15	368.1	400.9	434.4	468.6	503.6	539.4	575.9	613.1	651.1	689.9	729.4	769.6	810.6	852.4	894.9	938.1
16	390.4	423.5	457.4	492.0	527.4	563.5	600.4	638.0	676.4	715.5	755.4	796.0	837.4	879.5	922.4	966.0
17	412.6	446.1	480.4	515.4	551.1	587.6	624.9	662.9	701.6	741.1	781.4	822.4	864.1	906.6	949.9	993.9
18	434.9	468.8	503.4	538.8	574.9	611.8	649.4	687.8	726.9	766.8	807.4	848.8	890.9	933.8	977.4	1021.8
19	457.1	491.4	526.4	562.1	598.6	635.9	673.9	712.6	752.1	792.4	833.4	875.1	917.6	960.9	1004.9	1049.6
20	479.4	514.0	549.4	585.5	622.4	660.0	698.4	737.5	777.4	818.0	859.4	901.5	944.4	988.0	1032.4	1077.5
21	501.6	536.6	572.4	608.9	646.1	684.1	722.9	762.4	802.6	843.6	885.4	927.9	971.1	1015.1	1059.9	1105.4
22	523.9	559.3	595.4	632.3	669.9	708.3	747.4	787.3	827.9	869.3	911.4	954.3	997.9	1042.3	1087.4	1133.3

Table 4.A.6 Bytes size of the BTS baseline PReLU model with CNN1D+GRU weights quantized to 1-bit.

Filters	GRU cells															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	60.6	88.1	116.4	145.4	175.1	205.6	236.9	268.9	301.6	335.1	369.4	404.4	440.1	476.6	513.9	551.9
2	86.9	114.8	143.4	172.8	202.9	233.8	265.4	297.8	330.9	364.8	399.4	434.8	470.9	507.8	545.4	583.8
3	113.1	141.4	170.4	200.1	230.6	261.9	293.9	326.6	360.1	394.4	429.4	465.1	501.6	538.9	576.9	615.6
4	139.4	168.0	197.4	227.5	258.4	290.0	322.4	355.5	389.4	424.0	459.4	495.5	532.4	570.0	608.4	647.5
5	165.6	194.6	224.4	254.9	286.1	318.1	350.9	384.4	418.6	453.6	489.4	525.9	563.1	601.1	639.9	679.4
6	191.9	221.3	251.4	282.3	313.9	346.3	379.4	413.3	447.9	483.3	519.4	556.3	593.9	632.3	671.4	711.3
7	218.1	247.9	278.4	309.6	341.6	374.4	407.9	442.1	477.1	512.9	549.4	586.6	624.6	663.4	702.9	743.1
8	244.4	274.5	305.4	337.0	369.4	402.5	436.4	471.0	506.4	542.5	579.4	617.0	655.4	694.5	734.4	775.0
9	270.6	301.1	332.4	364.4	397.1	430.6	464.9	499.9	535.6	572.1	609.4	647.4	686.1	725.6	765.9	806.9
10	296.9	327.8	359.4	391.8	424.9	458.8	493.4	528.8	564.9	601.8	639.4	677.8	716.9	756.8	797.4	838.8
11	323.1	354.4	386.4	419.1	452.6	486.9	521.9	557.6	594.1	631.4	669.4	708.1	747.6	787.9	828.9	870.6
12	349.4	381.0	413.4	446.5	480.4	515.0	550.4	586.5	623.4	661.0	699.4	738.5	778.4	819.0	860.4	902.5
13	375.6	407.6	440.4	473.9	508.1	543.1	578.9	615.4	652.6	690.6	729.4	768.9	809.1	850.1	891.9	934.4
14	401.9	434.3	467.4	501.3	535.9	571.3	607.4	644.3	681.9	720.3	759.4	799.3	839.9	881.3	923.4	966.3
15	428.1	460.9	494.4	528.6	563.6	599.4	635.9	673.1	711.1	749.9	789.4	829.6	870.6	912.4	954.9	998.1
16	454.4	487.5	521.4	556.0	591.4	627.5	664.4	702.0	740.4	779.5	819.4	860.0	901.4	943.5	986.4	1030.0
17	480.6	514.1	548.4	583.4	619.1	655.6	692.9	730.9	769.6	809.1	849.4	890.4	932.1	974.6	1017.9	1061.9
18	506.9	540.8	575.4	610.8	646.9	683.8	721.4	759.8	798.9	838.8	879.4	920.8	962.9	1005.8	1049.4	1093.8
19	533.1	567.4	602.4	638.1	674.6	711.9	749.9	788.6	828.1	868.4	909.4	951.1	993.6	1036.9	1080.9	1125.6
20	559.4	594.0	629.4	665.5	702.4	740.0	778.4	817.5	857.4	898.0	939.4	981.5	1024.4	1068.0	1112.4	1157.5

4.A.4 All layers

Table 4.A.7 Bytes size of the BTS baseline ReLU model with all weights quantized to 1-bit.

Filters	GRU cells															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	54.9	80.6	107.1	134.4	162.4	191.1	220.6	250.9	281.9	313.6	346.1	379.4	413.4	448.1	483.6	519.9
2	77.1	103.3	130.1	157.8	186.1	215.3	245.1	275.8	307.1	339.3	372.1	405.8	440.1	475.3	511.1	547.8
3	99.4	125.9	153.1	181.1	209.9	239.4	269.6	300.6	332.4	364.9	398.1	432.1	466.9	502.4	538.6	575.6
4	121.6	148.5	176.1	204.5	233.6	263.5	294.1	325.5	357.6	390.5	424.1	458.5	493.6	529.5	566.1	603.5
5	143.9	171.1	199.1	227.9	257.4	287.6	318.6	350.4	382.9	416.1	450.1	484.9	520.4	556.6	593.6	631.4
6	166.1	193.8	222.1	251.3	281.1	311.8	343.1	375.3	408.1	441.8	476.1	511.3	547.1	583.8	621.1	659.3
7	188.4	216.4	245.1	274.6	304.9	335.9	367.6	400.1	433.4	467.4	502.1	537.6	573.9	610.9	648.6	687.1
8	210.6	239.0	268.1	298.0	328.6	360.0	392.1	425.0	458.6	493.0	528.1	564.0	600.6	638.0	676.1	715.0
9	232.9	261.6	291.1	321.4	352.4	384.1	416.6	449.9	483.9	518.6	554.1	590.4	627.4	665.1	703.6	742.9
10	255.1	284.3	314.1	344.8	376.1	408.3	441.1	474.8	509.1	544.3	580.1	616.8	654.1	692.3	731.1	770.8
11	277.4	306.9	337.1	368.1	399.9	432.4	465.6	499.6	534.4	569.9	606.1	643.1	680.9	719.4	758.6	798.6
12	299.6	329.5	360.1	391.5	423.6	456.5	490.1	524.5	559.6	595.5	632.1	669.5	707.6	746.5	786.1	826.5
13	321.9	352.1	383.1	414.9	447.4	480.6	514.6	549.4	584.9	621.1	658.1	695.9	734.4	773.6	813.6	854.4
14	344.1	374.8	406.1	438.3	471.1	504.8	539.1	574.3	610.1	646.8	684.1	722.3	761.1	800.8	841.1	882.3
15	366.4	397.4	429.1	461.6	494.9	528.9	563.6	599.1	635.4	672.4	710.1	748.6	787.9	827.9	868.6	910.1
16	388.6	420.0	452.1	485.0	518.6	553.0	588.1	624.0	660.6	698.0	736.1	775.0	814.6	855.0	896.1	938.0
17	410.9	442.6	475.1	508.4	542.4	577.1	612.6	648.9	685.9	723.6	762.1	801.4	841.4	882.1	923.6	965.9
18	433.1	465.3	498.1	531.8	566.1	601.3	637.1	673.8	711.1	749.3	788.1	827.8	868.1	909.3	951.1	993.8
19	455.4	487.9	521.1	555.1	589.9	625.4	661.6	698.6	736.4	774.9	814.1	854.1	894.9	936.4	978.6	1021.6
20	477.6	510.5	544.1	578.5	613.6	649.5	686.1	723.5	761.6	800.5	840.1	880.5	921.6	963.5	1006.1	1049.5
21	499.9	533.1	567.1	601.9	637.4	673.6	710.6	748.4	786.9	826.1	866.1	906.9	948.4	990.6	1033.6	1077.4
22	522.1	555.8	590.1	625.3	661.1	697.8	735.1	773.3	812.1	851.8	892.1	933.3	975.1	1017.8	1061.1	1105.3

Table 4.A.8 Bytes size of the BTS baseline PReLU model with all weights quantized to 1-bit.

Filters	GRU cells																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	58.9	84.6	111.1	138.4	166.4	195.1	224.6	254.9	285.9	317.6	350.1	383.4	417.4	452.1	487.6	523.9	560.9
2	85.1	111.3	138.1	165.8	194.1	223.3	253.1	283.8	315.1	347.3	380.1	413.8	448.1	483.3	519.1	555.8	593.1
3	111.4	137.9	165.1	193.1	221.9	251.4	281.6	312.6	344.4	376.9	410.1	444.1	478.9	514.4	550.6	587.6	625.4
4	137.6	164.5	192.1	220.5	249.6	279.5	310.1	341.5	373.6	406.5	440.1	474.5	509.6	545.5	582.1	619.5	657.6
5	163.9	191.1	219.1	247.9	277.4	307.6	338.6	370.4	402.9	436.1	470.1	504.9	540.4	576.6	613.6	651.4	689.9
6	190.1	217.8	246.1	275.3	305.1	335.8	367.1	399.3	432.1	465.8	500.1	535.3	571.1	607.8	645.1	683.3	722.1
7	216.4	244.4	273.1	302.6	332.9	363.9	395.6	428.1	461.4	495.4	530.1	565.6	601.9	638.9	676.6	715.1	754.4
8	242.6	271.0	300.1	330.0	360.6	392.0	424.1	457.0	490.6	525.0	560.1	596.0	632.6	670.0	708.1	747.0	786.6
9	268.9	297.6	327.1	357.4	388.4	420.1	452.6	485.9	519.9	554.6	590.1	626.4	663.4	701.1	739.6	778.9	818.9
10	295.1	324.3	354.1	384.8	416.1	448.3	481.1	514.8	549.1	584.3	620.1	656.8	694.1	732.3	771.1	810.8	851.1
11	321.4	350.9	381.1	412.1	443.9	476.4	509.6	543.6	578.4	613.9	650.1	687.1	724.9	763.4	802.6	842.6	883.4
12	347.6	377.5	408.1	439.5	471.6	504.5	538.1	572.5	607.6	643.5	680.1	717.5	755.6	794.5	834.1	874.5	915.6
13	373.9	404.1	435.1	466.9	499.4	532.6	566.6	601.4	636.9	673.1	710.1	747.9	786.4	825.6	865.6	906.4	947.9
14	400.1	430.8	462.1	494.3	527.1	560.8	595.1	630.3	666.1	702.8	740.1	778.3	817.1	856.8	897.1	938.3	980.1
15	426.4	457.4	489.1	521.6	554.9	588.9	623.6	659.1	695.4	732.4	770.1	808.6	847.9	887.9	928.6	970.1	1012.4
16	452.6	484.0	516.1	549.0	582.6	617.0	652.1	688.0	724.6	762.0	800.1	839.0	878.6	919.0	960.1	1002.0	1044.6
17	478.9	510.6	543.1	576.4	610.4	645.1	680.6	716.9	753.9	791.6	830.1	869.4	909.4	950.1	991.6	1033.9	1076.9
18	505.1	537.3	570.1	603.8	638.1	673.3	709.1	745.8	783.1	821.3	860.1	899.8	940.1	981.3	1023.1	1065.8	1109.1
19	531.4	563.9	597.1	631.1	665.9	701.4	737.6	774.6	812.4	850.9	890.1	930.1	970.9	1012.4	1054.6	1097.6	1141.4
20	557.6	590.5	624.1	658.5	693.6	729.5	766.1	803.5	841.6	880.5	920.1	960.5	1001.6	1043.5	1086.1	1129.5	1173.6

Chapter 5

Discussion

5.1 Conclusion

Neural networks are popular and powerful algorithms that have strong theoretical and practical advantages covering a wide range of applications using a uniform design approach, from labeled data to state-of-the-art algorithm performance. Their complex and computationally expensive nature makes them challenging to deploy and leverage, especially on resource-constrained platforms. Fostering new directions in efficient neural networks has significant potential for the research and industry community, as there is a great opportunity to reduce costs related to power consumption, and production for consumer markets, and to broaden the scope of the applicability of deep learning models.

MEMS and microcontrollers are small hardware components integrated on a scale of billions annually in various consumer products such as cars, wearables, or home appliances. Their capacity to serve as a sensing interface from the real-world to the digital world by collecting, processing, and interacting with their immediate surroundings, in a continuous and real-time manner at a very low-cost, makes them highly attractive hardware platforms. This inherently makes them operate in very constrained settings.

This thesis focused on the emerging field of tinyML, and particularly the intersection of deep learning with microcontrollers for MEMS-based applications. The convergence of these technologies has the potential to revolutionize various industries by enabling embedded hardware to process local data and interact with their environment in an automated and intelligent way. However, tinyML on microcontrollers comes with high theoretical and practical challenges and opportunities in a wide range of applications and products. We emphasized that succeeding to deliver ultra-low power inference can result in billion-scale production and consumer savings in costs, annually. Consequently, we argue that this research is particularly significant in the context of deep learning

applications on resource-constrained devices, where the deployment of deep learning methods is challenging due to power constraints and hardware limitations. We aim to provide and open new opportunities for theoretical and deep learning applications, as well as respond to the strong industrial and consumer incentives to design and deliver smart sensing applications in ultra-low power settings. Additionnally, we seek to fill the promises of deep learning to provide non-practitioners with a uniform design process to create algorithm at their will for MEMS-based applications.

In Chapter 1, we first reviewed neural networks and motivated our interest to apply them to MEMS-based applications on ultra-low power devices, reinforcing the unique challenges and opportunities of this work. In particular, our use case targets the most constrained hardware microcontrollers found in the state-of-the-art to deploy neural networks on. Our review of the literature helped us identify key methods and tools for designing efficient neural networks that fit our low-power constraints.

In particular, in Chapter 2, we explored efficient RNNs called MGUs, pruning, and knowledge distillation. We obtained positive results in CNNs or MLP-based models with pruning and knowledge distillation. In other scenarios, we found contrasting results when involving RNN-based architectures in large pruned models, as well as no significant gain with knowledge distillation and MGUs. This highlights the need to explore and research more specialized methods for RNN-based models.

In Chapter 3, we provided a comprehensive overview of tinyMLOps and proposed our own automated tinyMLOps solution to collect and create robust datasets for MEMS-based applications, train models, quantize them to 8-bit integers, export them using lightweight C code. We successfully deployed neural networks requiring as low as 1 kB of memory with 6 MHz clock frequency with motion-based applications, reaching a new extreme-low power level of inference, not previously achieved in the state-of-the-art. The main other achievements of our framework are to offer a platform to facilitate the automated creation of algorithm design, through a uniform approach, accessible by non-practitioners. The incentive is to allow customers to manipulate and create sensor applications at their will.

Building upon this foundation work, we propose a novel generalized method to quantize deep learning models to N-bits and extend to extreme quantization levels as low as 1-bit in Chapter 4. We achieved the best size-performance tradeoff by quantizing all weight parameters to 4-bits, with less than 1% of accuracy.

The results of this work contribute to the advance of tinyML and enable the broader adoption of intelligent and efficient sensing devices in various real-world applications.

5.2 Future directions

Methods for efficient neural networks (Chapter 2). Future work will involve specialized knowledge distillation techniques for RNN-based models, including multi-teacher approaches (Liu et al., 2020b), or combine it with quantization (Qu et al., 2020). Moreover, finding new structured or unstructured pruning strategies that hardware can leverage to accelerate on-device inference is paramount. Furthermore, investigating alternative model compression techniques such as weight sharing or tensor decomposition may also prove valuable for tinyML.

TinyML for 8-bit neural networks on ultra-low power MCUs (Chapter 3). Several key future directions include extending the portfolio of MEMS-based applications to refine the deployment pipeline for efficient neural networks on MCUs. Moreover, exploring novel data augmentation strategies to enhance the quality of datasets, the design of more compact architectures, and the reduction of manual data collection and labeling efforts are key aspects for further study.

Regularized N-bit Quantization for Ultra-Low Power Microcontrollers (Chapter 4). An interesting perspective encompasses finding new regularization functions and hybrid combinations to {1, 2, 4, 8}-bits. Additionally, reducing the activation size from 32-bit integers is a critical area for future research.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems.

Achille, A., Rovere, M., and Soatto, S. (2019). Critical learning periods in deep networks. In *7th International Conference on Learning Representations*.

Al-Rfou, R., Alain, G., Almahairi, A., Angermüller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Snyder, J. B., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P. L., Cho, K., Chorowski, J., Christiano, P. F., Cooijmans, T., Côté, M.-A., Côté, M., Courville, A. C., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Ducoffe, M., Dumoulin, V., Kahou, S. E., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I. J., Graham, M., Gülçehre, Ç., Hamel, P., Harlouchet, I., Heng, J.-P., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Korobov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrançois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P.-A., Mastropietro, O., McGibbon, R., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C. J., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, É., Spieckermann, S., Subramanyam, S. R., Sygnowski, J., Tanguay, J., van Tulder, G., Turian, J. P., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., and Zhang, Y. (2016). Theano: A Python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688.

Alemdar, H., Leroy, V., Prost-Boucle, A., and Pétrot, F. (2017). Ternary neural networks for resource-efficient AI applications. In *International Joint Conference on Neural Networks, IJCNN*, pages 2547–2554.

Alnaim, N. and Abbod, M. F. (2019). Mini gesture detection using neural networks algorithms. In *International Conference on Machine Vision*.

Alqahtani, A., Xie, X., and Jones, M. W. (2021). Literature review of deep network compression. *Informatics*, 8(4).

Alvarez, J. M. and Salzmann, M. (2016). Learning the Number of Neurons in Deep Networks. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc.

Alvarez, J. M. and Salzmann, M. (2017). Compression-aware Training of Deep Networks. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Antonini, M., Pincheira, M., Vecchio, M., and Antonelli, F. (2022). Tiny-MLOps: A framework for orchestrating ML applications at the far edge of IoT systems. In *2022 IEEE International Conference on Evolving and Adaptive Intelligent Systems (EAIS)*, pages 1–8.

Arik, S. Ö., Kliegl, M., Child, R., Hestness, J., Gibiansky, A., Fougner, C., Prenger, R. J., and Coates, A. (2017). Convolutional recurrent neural networks for small-footprint keyword spotting. In *18th Annual Conference of the International Speech Communication Association, Interspeech*.

AskariHemmat, M., Hemmat, R. A., Hoffman, A., Lazarevich, I., Saboori, E., Mastropietro, O., Sah, S., Savaria, Y., and David, J.-P. (2022). QReg: On Regularization Effects of Quantization. *arXiv:2206.12372 [cs]*.

Ataya, A. (2022). Tiny ML for tiny sensors: Waking smarter for less. Presented at the TinyML Summit 2022, San Francisco Bay Area, March 28-30, 2022.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In Bengio, Y. and LeCun, Y., editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Baldi, P. (1995). Gradient descent learning algorithm overview: A general dynamical systems perspective. *IEEE Transactions on Neural Networks*, 6(1):182–195.

Ballas, C. (2022). *Inducing Sparsity in Deep Neural Networks through Unstructured Pruning for Lower Computational Footprint*. PhD thesis, Dublin City University.

Banbury, C., Zhou, C., Fedorov, I., Navarro, R. M., Thakker, U., Gope, D., Reddi, V. J., Mattina, M., and Whatmough, P. N. (2021a). MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers.

Banbury, C. R., Reddi, V. J., Lam, M., Fu, W., Fazel, A., Holleman, J., Huang, X., Hurtado, R., Kanter, D., Lokhmotov, A., Patterson, D., Pau, D., Seo, J.-s., Sieracki, J., Thakker, U., Verhelst, M., and Yadav, P. (2021b). Benchmarking TinyML Systems: Challenges and Direction.

Banner, R., Nahshan, Y., Hoffer, E., and Soudry, D. (2019). Post-training 4-bit quantization of convolution networks for rapid-deployment. *arXiv:1810.05723 [cs]*.

Bengio, Y., Léonard, N., and Courville, A. C. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv:1308.3432 [cs]*.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Bhalgat, Y., Lee, J., Nagel, M., Blankevoort, T., and Kwak, N. (2020). LSQ+: Improving low-bit quantization through learnable offsets and better initialization. *arXiv:2004.09576 [cs, stat]*.

Bhardwaj, K., Aryan, and Yadav, R. (2022). Single input-based CNN-LSTM and CNN-GRU based HAR using wearable sensors. In *Advancement in Electronics & Communication Engineering*.

Bichler, O., Briand, D., Gacoin, V., Bertelone, B., Allenet, T., and Thiele, J. C. (2017). N2D2 - An Open-source Design Environment For DNN. Manual available on GitHub: https://github.com/CEA-LIST/N2D2.

Bishop, C. (1995). Regularization and complexity control in feed-forward networks. In *Proceedings International Conference on Artificial Neural Networks ICANN'95*, volume 1, pages 141–148. EC2 et Cie.

Bjorck, N., Gomes, C. P., Selman, B., and Weinberger, K. Q. (2018). Understanding batch normalization. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Buciluă, C., Caruana, R., and Niculescu-Mizil, A. (2006). Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '06*, page 535, Philadelphia, PA, USA. ACM Press.

Bulat, A., Tzimiropoulos, G., Kossaifi, J., and Pantic, M. (2019). Improved training of binary networks for human pose estimation and image recognition. *arXiv:1904.05868*.

Burkholz, R., Laha, N., Mukherjee, R., and Gotovos, A. (2021). On the existence of universal lottery tickets. *arXiv:2111.11146*.

Cahuantzi, R., Chen, X., and Güttel, S. (2021). A comparison of LSTM and GRU networks for learning symbolic sequences. *arXiv:2107.02248*.

Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. (2020a). Once-for-All: Train One Network and Specialize it for Efficient Deployment. *arXiv:1908.09791 [cs, stat]*.

Cai, Y., Yao, Z., Dong, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. (2020b). ZeroQ: A novel zero shot quantization framework. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13166–13175.

Chen, M. (2018). MinimalRNN: Toward More Interpretable and Trainable Recurrent Neural Networks. *arXiv:1711.06788*.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. (2018). TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 579–594, USA. USENIX Association.

Cheng, Y., Wang, D., Zhou, P., and Zhang, T. (2020). A Survey of Model Compression and Acceleration for Deep Neural Networks. *arXiv:1710.09282 [cs]*.

Cho, J. H. and Hariharan, B. (2019). On the efficacy of knowledge distillation. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4793–4801.

Choi, J., Wang, Z., Venkataramani, S., Chuang, P. I.-J., Srinivasan, V., and Gopalakrishnan, K. (2018). PACT: Parameterized clipping activation for quantized neural networks. *arXiv:1805.06085*.

Choromańska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2014). The loss surfaces of multilayer networks. In *International Conference on Artificial Intelligence and Statistics*.

Choukroun, Y., Kravchik, E., Yang, F., and Kisilev, P. (2019). Low-bit Quantization of Neural Networks for Efficient Inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv:1412.3555*.

Courbariaux, M., Bengio, Y., and David, J.-P. (2015). BinaryConnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, volume 28.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314.

Darabi, S., Belbahri, M., Courbariaux, M., and Nia, V. P. (2020). Regularized Binary Network Training. *arXiv:1812.11800*.

David, R., Duke, J., Jain, A., Reddi, V. J., Jeffries, N., Li, J., Kreeger, N., Nappier, I., Natraj, M., Regev, S., Rhodes, R., Wang, T., and Warden, P. (2021). TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *arXiv:2010.08678 [cs]*.

de Foras, E. and Lê, M. T. (2022). One bit quantization for embedded systems. *Pending patent*.

Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. (2013). Predicting parameters in deep learning. In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc.

Doyu, H., Morabito, R., and Brachmann, M. (2021). A TinyMLaaS Ecosystem for Machine Learning in IoT: Overview and Research Challenges. In *2021 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–5.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). Neural architecture search. In Hutter, F., Kotthoff, L., and Vanschoren, J., editors, *Automated Machine Learning: Methods, Systems, Challenges*, pages 63–77. Springer International Publishing, Cham.

Fang, J., Shafiee, A., Abdel-Aziz, H., Thorsley, D., Georgiadis, G., and Hassoun, J. H. (2020). Post-training Piecewise Linear Quantization for Deep Neural Networks. In Vedaldi, A., Bischof, H., Brox, T., and Frahm, J.-M., editors, *Computer Vision – ECCV 2020*, volume 12347, pages 69–86. Springer International Publishing, Cham.

Fedorov, I., Stamenovic, M., Jensen, C., Yang, L.-C., Mandell, A., Gan, Y., Mattina, M., and Whatmough, P. N. (2020). TinyLSTMs: Efficient Neural Speech Enhancement for Hearing Aids. In *Interspeech 2020*, pages 4054–4058. ISCA.

Fischer, J. and Burkholz, R. (2022). Plant 'n' seek: Can you find the winning ticket? In *10th International Conference on Learning Representations, ICLR 2020*.

Frankle, J. and Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Frankle, J., Schwab, D. J., and Morcos, A. S. (2020). The Early Phase of Neural Network Training. *arXiv:2002.10365*.

Freire, P. J., Napoli, A., Ron, D. A., Spinnler, B., Anderson, M., Schairer, W., Bex, T., Costa, N., Turitsyn, S. K., and Prilepsky, J. E. (2023). Reducing computational complexity of neural networks in optical channel equalization: From concepts to implementation. *Journal of Lightwave Technology*, pages 1–26.

Fu, F., Hu, Y., He, Y., Jiang, J., Shao, Y., Zhang, C., and Cui, B. (2020). Don't waste your bits! Squeeze activations and gradients for deep neural networks via TinyScript. In III, H. D. and Singh, A., editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3304–3314. PMLR.

Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20(3):121–136.

Gal, Y. and Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1050–1059, New York, NY, USA. JMLR.org.

Gale, T., Elsen, E., and Hooker, S. (2019). The State of Sparsity in Deep Neural Networks. *arXiv:1902.09574 [cs, stat]*.

Garbay, T., Hachicha, K., Dobias, P., Dron, W., Lusich, P., Khalis, I., Pinna, A., and Granado, B. (2022). Accurate estimation of the CNN inference cost for TinyML devices. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*, pages 1–6.

Gers, F., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with LSTM. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855 vol.2.

Gers, F. A., Schraudolph, N. N., and Schmidhuber, J. (2003). Learning precise timing with lstm recurrent networks. *Journal of Machine Learning Research*, 3:115–143.

Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. (2021). A Survey of Quantization Methods for Efficient Neural Network Inference. *arXiv:2103.13630 [cs]*.

Golatkar, A. S., Achille, A., and Soatto, S. (2019). Time Matters in Regularizing Deep Networks: Weight Decay and Data Augmentation Affect Early Learning Dynamics, Matter Little Near Convergence. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Golubeva, A., Gur-Ari, G., and Neyshabur, B. (2021). Are wider nets better given the same number of parameters? In *9th International Conference on Learning Representations, ICLR 2021*.

Gong, R., Liu, X., Jiang, S., Li, T., Hu, P., Lin, J., Yu, F., and Yan, J. (2019). Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 4851–4860.

Gong, Y. and Poellabauer, C. (2018). Impact of aliasing on deep CNN-Based end-to-end acoustic models. In *Proc. Interspeech 2018*, pages 2698–2702.

Goodfellow, I. J., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA, USA.

Graves, A. and Jaitly, N. (2014). Towards end-to-end speech recognition with recurrent neural networks. In Xing, E. P. and Jebara, T., editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1764–1772, Bejing, China. PMLR.

Gray, R. and Neuhoff, D. (1998). Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383.

Guo, Y. (2018). A Survey on Methods and Theories of Quantized Neural Networks. *arXiv:1808.04752 [cs, stat]*.

Hagiwara, M. (1993). Removal of hidden units and weights for back propagation networks. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, volume 1, pages 351–354 vol.1.

Han, H. and Siebert, J. (2022). TinyML: A Systematic Review and Synthesis of Existing Research. In *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pages 269–274.

Han, S., Mao, H., and Dally, W. J. (2016). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]*.

Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pages 1135–1143, Cambridge, MA, USA. MIT Press.

Hawks, B., Duarte, J., Fraser, N. J., Pappalardo, A., Tran, N., and Umuroglu, Y. (2021). Ps and Qs: Quantization-Aware Pruning for Efficient Low Latency Neural Network Inference. *Frontiers in Artificial Intelligence*, 4.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

He, Y., Zhang, X., and Sun, J. (2017). Channel pruning for accelerating very deep neural networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1398–1406.

Heck, J. C. and Salem, F. M. (2017). Simplified minimal gated unit variations for recurrent neural networks. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1593–1596, Boston, MA, USA. IEEE.

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the Knowledge in a Neural Network. *arXiv:1503.02531 [cs, stat]*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. (2021). Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *arXiv:2102.00554 [cs]*.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., and Adam, H. (2019). Searching for MobileNetV3.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv:1704.04861*.

Huang, Y. (2009). Advances in artificial neural networks – methodological development and application. *Algorithms*, 2(3):973–1007.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 4114–4122, Red Hook, NY, USA. Curran Associates Inc.

Huffman, D. A. (2006). A method for the construction of minimum-redundancy codes. *Resonance*, 11(2):91–99.

Hymel, S., Banbury, C., Situnayake, D., Elium, A., Ward, C., Kelcey, M., Baaijens, M., Majchrzycki, M., Plunkett, J., Tischler, D., Grande, A., Moreau, L., Maslov, D., Beavis, A., Jongboom, J., and Reddi, V. J. (2023). Edge impulse: An mlops platform for tiny machine learning. *arXiv:2212.03332*.

Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:1602.07360*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 448–456, Lille, France. JMLR.org.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. (2017). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *arXiv:1712.05877 [cs, stat]*.

Kanjo, E. (2022). Sensing on the edge: Smartening up sensors. In *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 1–1.

Kawaguchi, K. and Huang, J. (2019). Gradient descent finds global minima for generalizable deep neural networks of practical sizes. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 92–99.

Khan, A., Sohail, A., Zahoora, U., and Qureshi, A. S. (2020). A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516.

Kim, H., Park, J., Lee, C., and Kim, J.-J. (2021). Improving Accuracy of Binary Neural Networks using Unbalanced Activation Distribution.

Kreuzberger, D., Kühl, N., and Hirschl, S. (2022). Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access*, page 13.

Krishnamoorthi, R. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv:1806.08342 [cs, stat]*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

Lai, L., Suda, N., and Chandra, V. (2018). CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *arXiv:1801.06601 [cs]*.

Lammel, G. (2015). The future of MEMS sensors in our connected world. In *2015 28th IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*, pages 61–64.

Lê, M. T. and Arbel, J. (2023). TinyMLOps for real-time ultra-low power MCUs applied to frame-based event classification. In *EuroMLSys '23: Proceedings of the 3rd European Workshop on Machine Learning and Systems, Rome, Italy, May 08-12, 2023*. ACM, New York, NY.

Lê, M. T. and de Foras, E. (2022). Towards Universal 1-Bit Weight Quantization of Neural Networks with end-to-end deployment on ultra-low power sensors. *tinyML EMEA Innovation Forum 2022*.

Lê, M. T., de Foras, E., and Arbel, J. (2023). Regularization for hybrid N-bit weight quantization of neural networks on ultra-low power microcontrollers. In Iliadis, L., Papaleonidas, A., Angelov, P., and Jayne, C., editors, *Artificial Neural Networks and Machine Learning – ICANN 2023*, Cham. Springer Nature Switzerland.

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I. V., and Lempitsky, V. S. (2015). Speeding-up convolutional neural networks using fine-tuned CP-Decomposition. In Bengio, Y. and LeCun, Y., editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Lee, J. H., Ha, S., Choi, S., Lee, W.-J., and Lee, S. (2018). Quantization for rapid deployment of deep neural networks. *CoRR*, abs/1810.05488.

Leroux, S., Simoens, P., Lootus, M., Thakore, K., and Sharma, A. (2022). TinyMLOps: Operational Challenges for Widespread Edge AI Adoption. *arXiv:2203.10923*.

Li, Y. and Liang, Y. (2018). Learning overparameterized neural networks via stochastic gradient descent on structured data. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 8168–8177, Red Hook, NY, USA. Curran Associates Inc.

Li, Z., Wallace, E., Shen, S., Lin, K., Keutzer, K., Klein, D., and Gonzalez, J. (2020). Train big, then compress: Rethinking model size for efficient training and inference of transformers. In III, H. D. and Singh, A., editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5958–5968. PMLR.

Liang, T., Glossner, J., Wang, L., Shi, S., and Zhang, X. (2021). Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403.

Liberis, E., Dudziak, Ł., and Lane, N. D. (2021). μNAS: Constrained Neural Architecture Search for Microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, EuroMLSys '21, pages 70–79, New York, NY, USA. Association for Computing Machinery.

Liberis, E. and Lane, N. D. (2020). Neural networks on microcontrollers: Saving memory at inference via operator reordering. *arXiv:1910.05110*.

Lin, D. D., Talathi, S. S., and Annapureddy, V. S. (2016). Fixed point quantization of deep convolutional networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2849–2858, New York, NY, USA. JMLR.org.

Lin, H. and Jegelka, S. (2018). ResNet with one-neuron hidden layers is a universal approximator. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 6172–6181, Red Hook, NY, USA. Curran Associates Inc.

Lin, J., Chen, W.-M., Lin, Y., cohn, j., Gan, C., and Han, S. (2020). MCUNet: Tiny Deep Learning on IoT Devices. In *Advances in Neural Information Processing Systems*, volume 33, pages 11711–11722. Curran Associates, Inc.

Lin, T., Wang, Y., Liu, X., and Qiu, X. (2022). A survey of transformers. *AI Open*, 3:111–132.

Liu, C., Jobst, M., Guo, L., Shi, X., Partzsch, J., and Mayr, C. (2023). Deploying machine learning models to ahead-of-time runtime on edge using microtvm. *arXiv:2304.04842*.

Liu, Y., Che, W., Qin, B., and Liu, T. (2020a). Exploring segment representations for neural semi-markov conditional random fields. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 28:813–824.

Liu, Y., Zhang, W., and Wang, J. (2020b). Adaptive multi-teacher multi-level knowledge distillation. *Neurocomputing*, 415:106–113.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. (2017). Learning efficient convolutional networks through network slimming. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2755–2763, Los Alamitos, CA, USA. IEEE Computer Society.

Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. (2019). Rethinking the Value of Network Pruning. *arXiv:1810.05270 [cs, stat]*.

Liu, Z., Wu, B., Luo, W., Yang, X., Liu, W., and Cheng, K.-T. (2018). Bi-real net: Enhancing the performance of 1-Bit CNNs with improved representational capability and advanced training algorithm. In Ferrari, V., Hebert, M., Sminchisescu, C., and Weiss, Y., editors, *Computer Vision – ECCV 2018*, pages 747–763, Cham. Springer International Publishing.

Louizos, C., Reisser, M., Blankevoort, T., Gavves, E., and Welling, M. (2019). Relaxed quantization for discretized neural networks. In *7th International Conference on Learning Representations, ICLR 2019*.

Louizos, C., Welling, M., and Kingma, D. P. (2018). Learning sparse neural networks through L_0 regularization. In *6th International Conference on Learning Representations, ICLR 2018*.

Lu, L., Zhang, C., Cao, K., Deng, T., and Yang, Q. (2022). A multichannel CNN-GRU model for human activity recognition. *IEEE access : practical innovations, open solutions*, 10:66797–66810.

Ma, J. (2020). A higher-level neural network library on microcontrollers (NNoM). Zenodo.

Maas, A. L., Hannun, A. Y., Ng, A. Y., et al. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, Georgia, USA.

Mazlan, A. B., Ng, Y. H., and Tan, C. K. (2022). Teacher-assistant knowledge distillation based indoor positioning system. *Sustainability*, 14(21).

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133.

Menard, D., Chillet, D., and Sentieys, O. (2006). Floating-to-Fixed-Point Conversion for Digital Signal Processors. *EURASIP Journal on Advances in Signal Processing*, 2006(1):096421.

Menghani, G. (2023). Efficient deep learning: A survey on making deep learning models smaller, faster, and better. *ACM Computing Surveys*, 55(12).

Mirzadeh, S. I., Farajtabar, M., Li, A., Levine, N., Matsukawa, A., and Ghasemzadeh, H. (2020). Improved knowledge distillation via teacher assistant. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):5191–5198.

Miyashita, D., Lee, E. H., and Murmann, B. (2016). Convolutional neural networks using logarithmic data representation. *ArXiv:1603.01025*.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, Madison, WI, USA. Omnipress.

Neill, J. O. (2020). An Overview of Neural Network Compression. *arXiv:2006.03669 [cs, stat]*.

Novac, P.-E., Boukli Hacene, G., Pegatoquet, A., Miramond, B., and Gripon, V. (2021). Quantization and Deployment of Deep Neural Networks on Microcontrollers. *Sensors*, 21(9):2984.

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. P. (2015). Tensorizing neural networks. *CoRR*, abs/1509.06569.

Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4(4):473–493.

OpenAI (2023). GPT-4 technical report. *arXiv:2303.08774*.

Ozerov, A. and Duong, N. Q. K. (2021). Inplace knowledge distillation with teacher assistant for improved training of flexible deep neural networks. In *2021 29th European Signal Processing Conference (EUSIPCO)*, pages 1356–1360.

Park, D. S., Chan, W., Zhang, Y., Chiu, C.-C., Zoph, B., Cubuk, E. D., and Le, Q. V. (2019). SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition. In *Interspeech 2019*, pages 2613–2617.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Poggio, T., Banburski, A., and Qianli Liao (2020a). Theoretical issues in deep networks. *Proceedings of the National Academy of Sciences*, 117(48):30039–30045.

Poggio, T., Liao, Q., and Banburski, A. (2020b). Complexity control by gradient descent in deep networks. *Nature Communications*, 11(1):1027.

Ponçot, R., Lê, M. T., de Foras, E., Ataya, A., and Hartwell, P. G. (2022). Method for improved keyword spotting. *Pending patent.*

Qin, H., Gong, R., Liu, X., Bai, X., Song, J., and Sebe, N. (2020). Binary neural networks: A survey. *Pattern Recognition*, 105:107281.

Qu, X., Wang, J., and Xiao, J. (2020). Quantization and knowledge distillation for efficient federated learning on edge devices. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 967–972.

Rajaraman, V. (2016). Ieee standard for floating point numbers. *Resonance*, 21:11–30.

Ramanujan, V., Wortsman, M., Kembhavi, A., Farhadi, A., and Rastegari, M. (2020). What's hidden in a randomly weighted neural network? In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11890–11899.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). XNOR-Net: ImageNet classification using binary convolutional neural networks. In Leibe, B., Matas, J., Sebe, N., and Welling, M., editors, *Computer Vision – ECCV 2016*, pages 525–542, Cham. Springer International Publishing.

Ray, P. P. (2022). A review on TinyML: State-of-the-art and prospects. *Journal of King Saud University - Computer and Information Sciences*, 34(4):1595–1623.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

Sah, A., Chatterjee, S., and Vaidheeswaran, A. (2022). TinyMLOps: Overview, Challenges and Implementation. *tinyML Summit 2022*.

Saha, S. S., Sandha, S. S., and Srivastava, M. (2022). Machine learning for microcontroller-class hardware: A review. *IEEE Sensors Journal*, 22(22):21362–21390.

Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. (2013). Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6655–6659.

Sakib, S., Ahmed, N., Kabir, A. J., and Ahmed, H. (2018). An overview of convolutional neural network: Its architecture and applications. *Preprints.org*, 2018(2018110546).

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520.

Schizas, N., Karras, A., Karras, C., and Sioutas, S. (2022). TinyML for ultra-low power AI and large scale IoT deployments: A systematic review. *Future Internet*, 14(12).

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. (2015). Hidden technical debt in machine learning systems. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc.

Shafique, M., Theocharides, T., Reddy, V. J., and Murmann, B. (2021). TinyML: Current Progress, Research Challenges, and Future Roadmap. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1303–1306.

Shen, M., Yin, H., Molchanov, P., Mao, L., Liu, J., and Alvarez, J. (2022). Structural pruning via latency-saliency knapsack. In *Advances in Neural Information Processing Systems*.

Shorten, C. and Khoshgoftaar, T. M. (2019). A survey on Image Data Augmentation for Deep Learning. *Journal of Big Data*, 6(1):60.

Simonyan, K. and Zisserman, A. (2014). Two-stream convolutional networks for action recognition in videos. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pages 568–576, Cambridge, MA, USA. MIT Press.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In Bengio, Y. and LeCun, Y., editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Sipola, T., Alatalo, J., Kokkonen, T., and Rantonen, M. (2022). Artificial intelligence in the IoT era: A review of edge AI hardware and software. In *2022 31st Conference of Open Innovations Association (FRUCT)*, pages 320–331.

Sjöberg, J. and Ljung, L. (1992). Overtraining, regularization, and searching for minimum in neural networks. *IFAC Proceedings Volumes*, 25(14):73–78.

Sponner, M., Waschneck, B., and Kumar, A. (2021). Compiler toolchains for deep learning workloads on embedded platforms. In *Research Symposium on Tiny Machine Learning*.

Srinivas, S. and Babu, R. V. (2015). Data-free parameter pruning for deep neural networks.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

Sudharsan, B., Salerno, S., Nguyen, D.-D., Yahya, M., Wahid, A., Yadav, P., Breslin, J. G., and Ali, M. I. (2021). TinyML Benchmark: Executing Fully Connected Neural Networks on Commodity Microcontrollers. In *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, pages 883–884.

Svoboda, F., Fernandez-Marques, J., Liberis, E., and Lane, N. D. (2022). Deep learning on microcontrollers: A study on deployment costs and challenges. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, pages 54–63, Rennes France. ACM.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Takeuchi, D., Yatabe, K., Koizumi, Y., Oikawa, Y., and Harada, N. (2020). Real-time speech enhancement using equilibriated rnn. *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

Tan, M. and Le, Q. V. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR.

Tang, W., Hua, G., and Wang, L. (2017). How to train a compact binary neural network with high accuracy? In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 2625–2631, San Francisco, California, USA. AAAI Press.

Thakker, U., Beu, J. G., Gope, D., Dasika, G., and Mattina, M. (2020). Rank and run-time aware compression of NLP Applications. *CoRR*, abs/2010.03193.

Timpl, L., Entezari, R., Sedghi, H., Neyshabur, B., and Saukh, O. (2021). Understanding the effect of sparsity on neural networks' robustness. In *Sparsity in Neural Networks - Advancing Understanding and Practice : SNN Workshop 2021*.

Tompson, J., Goroshin, R., Jain, A., LeCun, Y., and Bregler, C. (2015). Efficient object localization using convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 648–656.

Tu, Z., Chen, X., Ren, P., and Wang, Y. (2022). AdaBin: Improving binary neural networks with adaptive binary sets. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XI*, pages 379–395. Springer.

Ullrich, K., Meeds, E., and Welling, M. (2017). Soft weight-sharing for neural network compression. In *5th International Conference on Learning Representations, ICLR 2017*.

Unlu, H. (2020). Efficient Neural Network Deployment for Microcontroller. *arXiv:2007.01348 [cs]*.

Van Baalen, M., Louizos, C., Nagel, M., Amjad, R. A., Wang, Y., Blankevoort, T., and Welling, M. (2020). Bayesian Bits: Unifying Quantization and Pruning. *Advances in neural information processing systems*, 33:5741–5752.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Vladimirova, M., Arbel, J., and Girard, S. (2021). Bayesian neural network unit priors and generalized Weibull-tail property. In Balasubramanian, V. N. and Tsang, I. W., editors, *Asian Conference on Machine Learning, ACML 2021, 17-19 November 2021, Virtual Event*, volume 157 of *Proceedings of Machine Learning Research*, pages 1397–1412. PMLR.

Vladimirova, M., Verbeek, J., Mesejo, P., and Arbel, J. (2019). Understanding priors in bayesian neural networks at the unit level. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6458–6467. PMLR.

Walsh, C. A. (2013). Peter Huttenlocher (1931–2013). *Nature*, 502(7470):172–172.

Warden, P. (2018). Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv:1804.03209*.

Warden, P. and Situnayake, D. (2020). *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. (2016). Learning structured sparsity in deep neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 2082–2090, Red Hook, NY, USA. Curran Associates Inc.

Wu, H., Judd, P., Zhang, X., Isaev, M., and Micikevicius, P. (2020). Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv:2004.09602*.

Wu, J., Wang, Y., Wu, Z., Wang, Z., Veeraraghavan, A., and Lin, Y. (2018). Deep k-Means: Re-Training and Parameter Sharing with Harder Cluster Assignments for Compressing Deep Convolutions. *arXiv:1806.09228 [cs, stat]*.

Xue, J., Li, J., and Gong, Y. (2013). Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*.

Yin, P., Zhang, S., Lyu, J., Osher, S. J., Qi, Y., and Xin, J. (2018). BinaryRelax: A relaxation approach for training deep neural networks with quantized weights. *SIAM J. Imaging Sci.*, 11(4):2205–2223.

Yiu, J. (2019). Cortex-M resources - processor documentation.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2017). Understanding deep learning requires rethinking generalization.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2021a). Understanding deep learning (still) requires rethinking generalization. *Communications of The ACM*, 64(3):107–115.

Zhang, X., Colbert, I., Kreutz-Delgado, K., and Das, S. (2021b). Training deep neural networks with joint quantization and pruning of weights and activations. *arXiv:2110.08271*.

Zhang, Y., Suda, N., Lai, L., and Chandra, V. (2018). Hello Edge: Keyword Spotting on Microcontrollers. *arXiv:1711.07128 [cs, eess]*.

Zhou, A., Yao, A., Guo, Y., Xu, L., and Chen, Y. (2017). Incremental network quantization: Towards lossless CNNs with low-precision weights. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Zhou, G.-B., Wu, J., Zhang, C.-L., and Zhou, Z.-H. (2016). Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing*, 13(3):226–234.

Zhu, C., Han, S., Mao, H., and Dally, W. J. (2017). Trained ternary quantization. In *5th International Conference on Learning Representations, ICLR 2017*.

Zhu, J., Liu, X., Shi, Q., He, T., Sun, Z., Guo, X., Liu, W., Sulaiman, O. B., Dong, B., and Lee, C. (2020). Development trends and perspectives of future sensors and MEMS/NEMS. *Micromachines*, 11(1).

Zhu, M. and Gupta, S. (2017). To prune, or not to prune: Exploring the efficacy of pruning for model compression. *arXiv:1710.01878 [cs, stat]*.

Zingg, R. and Rosenthal, M. (2020). Artificial intelligence on microcontrollers. In *Embedded World Conference 2020, Nürnberg, 25.-27. February 2020*.

Zmora, N., Jacob, G., Zlotnik, L., Elharar, B., and Novik, G. (2019). Neural Network Distiller: A Python Package For DNN Compression Research. *arXiv:1910.12232 [cs, stat]*.