Ph.D. Thesis

Young Dae Kwon

Efficient Continual Learning and On-Device Training for Mobile and IoT Devices

Churchill College
University of Cambridge

September 2024

This dissertation is submitted for the degree of *Doctor of Philosophy* at the Department of Computer Science and Technology

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the preface and specified in the text. It is not substantially the same as any work that has already been submitted, or is being concurrently submitted, for any degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

Abstract

The surge in mobile phones, wearables, and Internet of Things (IoT) devices has resulted in an abundance of sensor data. This played a pivotal role in the widespread adoption of deep neural networks (DNN) to support various real-world scenarios in mobile computing, including personalising user experiences and enabling adaptive household robots. Such use cases require DNNs to continuously learn and adapt to changing real-world conditions, despite constraints such as limited labelled data, memory, and computational power. However, achieving continual learning (CL) and on-device training on resource-constrained edge devices poses significant challenges, both in terms of resource limitations and the complexity of learning algorithms to continually learn new tasks without forgetting old ones. This dissertation tackles these challenges by developing hardware-aware algorithms and systems that substantially optimise the utilisation of system resources for deployed DNNs on embedded and IoT platforms, while upholding high accuracy.

Initially, this dissertation explores the feasibility and applicability of various CL methods in diverse mobile sensing applications, taking into account constraints such as low computational capability, limited memory and storage. Drawing from this analysis, we identify the bottlenecks of existing CL systems. We then overcome the stringent resource limitations of mobile and embedded systems by crafting a novel CL approach called FastICARL that optimises the computational and storage demands of the representative CL method.

Subsequently, to seamlessly support on-device training and CL on extremely resource-constrained devices like microcontrollers (MCUs), we propose YONO, a multi-task inference system enabling in-memory model execution and seamless switching of varying tasks involving multiple user applications, which could facilitate on-device training and CL with multi-user scenarios. Furthermore, we propose TinyTrain, an efficient on-device training approach that minimises resource requirements while coping with limited data availability. TinyTrain significantly reduces memory usage, training latency, and energy consumption by effectively identifying and updating the essential model parts on the fly. This makes TinyTrain crucial for enabling CL on edge devices with limited resources.

Finally, the dissertation pushes the boundaries of CL in mobile computing by extending CL to embedded systems and highly resource-constrained MCUs. Building on our thorough analysis of CL and the technology developed for resource-constrained devices, we propose LifeLearner, an efficient CL system that comprehensively addresses on-device resource requirements namely data, memory, and computation.

LifeLearner is optimised for various hardware platforms such as edge devices (Jetson Nano and Raspberry Pi 3B+) and the STM32H747 MCU. Specifically, we co-design meta-learning with an efficient rehearsal strategy, enabling LifeLearner to rapidly learn new classes using only a few samples while alleviating forgetting. We then design a CL-tailored Compression Module that minimises the resource overheads of CL and hardware-aware optimisations to enhance overall runtime efficiency.

The methodologies developed, systems optimised, and insights gleaned from this dissertation lay the foundation for the widespread deployment of continual and on-device training systems that dynamically adapt to users and environments while operating efficiently within resource-constrained settings.

Acknowledgements

As I conclude this pivotal chapter of my academic journey, I am filled with profound gratitude for the many individuals who have supported, guided, and inspired me throughout my doctoral studies at the University of Cambridge.

First and foremost, I extend my deepest appreciation to my PhD supervisor, Prof. Cecilia Mascolo. Your insightful guidance, expertise, and unwavering support have been instrumental in shaping both my research and academic growth. Your mentorship has been invaluable, and I am truly grateful for the opportunity to learn from you.

I owe sincere thanks to my labmates: Hong Jia, Tong Xia, Lorena Qendro, Kayla-Jade Butkow, Dong Ma, Abhirup Ghosh, Ting Dang, Jing Han, Georgios Rizos, Yang Liu, Qiang Yang, Ian Tang, Yu (Yvonne) Wu, Yuwei (Evelyn) Zhang, Erika Bondareva, Jake Stuchbury-Wass, Sotirios Vavaroutas, Andrea Ferlini, Apinan Hasthanasombat, Dimitris Spathis, Andreas Grammenos, and Mathias Ciliberto. Our intellectual discussions, collaborative projects, and shared moments of both frustration and triumph have enriched my research experience immeasurably, making this journey not just educational but truly memorable.

I am particularly grateful to Dr Jagmohan Chauhan, formerly a postdoc and now a lecturer at the University of Southampton, whose guidance and insights helped shape my research approach. Similarly, I wish to thank my mentors during my internship at Samsung AI Centre-Cambridge, Dr Stylianos Venieris and Dr Rui Li, whose expertise and mentorship significantly accelerated my growth as a researcher.

My experience was further enriched by my colleagues at Samsung AI Centre-Cambridge: Sourav Bhattacharya, Hongxiang Fan, Alexandros Kouris, Royson Lee, Łukasz Dudziak, Konstantin Mishchenko, Abhinav Mehrotra, Alberto Gil Ramos, and Malcolm Chadwick. Our collaborations opened new perspectives that have profoundly influenced my work and broadened my understanding of the field.

Finally, to my family – my father, mother, sister, and brother – your unconditional love and encouragement have been my anchor throughout this challenging yet rewarding journey. Above all, my heart is full of gratitude for my wife. Your endless love, patience, and belief in me have been and will be my greatest source of strength and motivation. This achievement is as much yours as it is mine.

To everyone who has been part of this significant milestone in my life – thank you. Your contributions, whether large or small, have helped shape not only this research but also the person I have become today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent advances in deep learning have made unprecedented progress in various applications ranging from computer vision [3], natural language processing [4], and mobile computing [5]. In particular, mobile sensing applications that rely on fine-grained and continuous data collection from sensors have benefited significantly from deep learning approaches [6, 7]. Researchers have successfully applied Deep Neural Networks (DNNs) to mobile sensing tasks such as human activity recognition [8, 9], gesture recognition [10], tracking and localization [11], mental health and wellbeing assessment [12], keyword spotting [13, 14], emotion recognition [15], and environmental sound classification [16].

A critical challenge common to these applications is the need for deployed DNNs to continually accommodate new tasks and user contexts [17] over time, adapting to dynamically changing input distributions [18] while operating within strict resource constraints imposed by edge devices. For instance, smart appliances may need to learn different voice commands based on varying usage patterns and users. Household robotic devices would require continuous learning to recognise new objects and human poses for seamless human-robot interaction. Similarly, Human Activity Recognition (HAR) applications using smartphone sensors may need to expand their capabilities from recognising simple activities like sitting and standing to capturing more complex activities such as walking or playing sports, accommodating diverse patterns, use cases and users, as well as adapting to sensor orientation changes [19].

In these scenarios, two key capabilities become crucial:

- **Continual Learning (CL)**: The ability to learn consecutive tasks without forgetting previously learned ones [20, 21].
- **On-device Training**: The capacity to adapt DNNs to specific user data or environmental conditions without relying on external servers or cloud computing

resources [22]

These capabilities are essential in real-world applications where DNNs are deployed on resource-constrained devices, enabling adaptive systems that continually provide practical value for users.

In recent years, researchers and engineers have attempted to enable CL and on-device training on edge devices with limited resources such as Raspberry Pi Zero with 512 MB RAM. However, achieving efficient on-device CL and training on such devices presents significant challenges. First, on-device training requires excessive memory footprint (*e.g.* 1 GB even for small models with few hundred KBs [23]) and computational costs (*e.g.* 3× compared to *inference* [24]), let alone on *microcontroller units (MCUs)* which has extremely limited on-chip memory/storage space (*e.g.* 512 KB SRAM and 1 MB Flash). Also, the scarcity of user-labelled data on mobile and IoT devices further complicates the task. In addition, CL methods typically demand additional resources to tackle the forgetting problem of DNNs (refer to Section 1.2 for more details). These factors make it extremely challenging, if not impossible, to operate CL and on-device training on mobile and IoT devices. However, enabling accurate and efficient CL and on-device training is essential for facilitating adaptive capabilities in billions of mobile and IoT devices, allowing them to respond to frequently changing environments and real-world user applications.

This dissertation addresses these challenges and advances the boundaries of CL and on-device training in mobile computing by enabling their operation on both mobile/embedded devices and extremely resource-constrained MCUs. Overall, this dissertation encompasses the following key contributions:

1. We started with a comprehensive evaluation of various CL methods and identified various system bottlenecks causing inefficiency in resource-constrained systems. This analysis led to the development of FastICARL, a novel algorithm that drastically improves the system efficiency without compromising accuracy (Chapter 3).

2. To empower the potential of CL and on-device training on extremely resource-constrained devices such as MCUs, we developed YONO. This multi-task inference system enables efficient in-memory model execution and swapping of multiple heterogeneous DNNs on MCUs, which could facilitate CL and on-device training of multi-user and multi-task applications (Chapter 4).

3. We proposed TinyTrain, an on-device training approach that jointly optimises data, memory, and computation. TinyTrain drastically reduces training time and energy consumption on devices by selectively updating parts of the model while achieving state-of-the-art (SOTA) accuracy by explicitly addressing data scarcity (Chapter 5).

4. To extend CL to both mobile/embedded systems and MCUs, we introduced Life-Learner, a hardware-aware CL and on-device training system that can operate

on both mobile/embedded and extreme edge devices (with only 512 KB SRAM and 1 MB Flash) by optimising system resources while retaining high CL accuracy (Chapter 6).

5. Finally, we summarise the dissertation's contributions, reflect on insights drawn from the technical chapters, discuss the implications, and outline future research directions (Chapter 7).

To our knowledge, this dissertation presents the first co-designed hardware-aware algorithms and systems enabling efficient CL and on-device training on (extremely) resource-constrained devices. This innovation paves the way for the ubiquitous deployment of continual and on-device learning systems that can continually adapt to users and environments over time.

1.1 Motivating Example: Adaptive Personal Assistants on Resource-Constrained Devices

To illustrate the importance of continual learning and on-device training in mobile computing, consider a wearable health monitoring device that tracks a user's activities throughout the day. Initially, the device comes with a pre-trained model to recognise basic activities like walking, running, and sleeping. However, users could engage in diverse and personalised activities—yoga poses, specific sports movements, or rehabilitation exercises that were not included in the original training dataset.

A traditional approach would deploy a large, foundation model that attempts to cover all possible activities that a user might perform [25, 26]. However, such a model would:

1. Require prohibitively large memory footprint (potentially over several GBs) on an edge device with limited system resources (with a few hundred MBs RAM).
2. Consume excessive energy during inference, drastically reducing battery life.
3. Still fail to recognise many user-specific activities or movement patterns.
4. Struggle to adapt to how a user's movement patterns change over time (e.g., during recovery from injury).

Instead, efficient CL and on-device training offer a more elegant solution by enabling:

1. To start with a compact base model that handles common activities.
2. To incrementally learn new user-specific activities on-device as they occur.
3. To adapt to changes in user movement patterns over time.

4. To maintain privacy by keeping sensitive health data on-device.
5. To operate within strict resource constraints through techniques developed in this dissertation (see Chapters 3, 4, 5, and 6).

This example illustrates why CL and on-device training is not merely a theoretical nicety but a practical necessity for truly personalised and adaptive mobile sensing applications. The alternative–periodically retraining a large foundation model from scratch with all existing source and new data–would be prohibitively expensive in terms of computation, memory usage, and energy consumption, making it infeasible for resource-constrained devices.

1.2 Challenges in Enabling Continual Learning and On-device Training in Mobile Computing

In the context of mobile computing, user applications require DNNs to continuously learn new contexts and adapt to diverse real-world scenarios within strict resource constraints. This makes efficient CL and on-device systems essential components. Additionally, as CL often involves training (*i.e.* backpropagation) of deployed DNNs, the challenges of enabling on-device training must be considered jointly.

Challenge 1: Lack of Comprehensive Evaluation of CL in Mobile Computing

Machine Learning (ML) models typically suffer from Catastrophic Forgetting (CF) [27, 28], that is, a learned model experiences performance degradation on previously learned task(s) (*e.g.* task A) when incorporating information relevant to a new task (*e.g.* task B). Initially, researchers have developed various Continual Learning (CL) methods to address this issue, enabling models to learn new tasks over time while retaining previously learned knowledge [29]. Such CL methods have been developed primarily in the computer vision domain [29, 30, 31, 32].

However, it is unclear whether they can be effectively applied to other data modalities such as those used in mobile sensing tasks. For example, Human Activity Recognition (HAR) relies on Inertial Measurement Unit (IMU) data [33], Gesture Recognition (GR) uses Surface Electromyography (sEMG) [34], and Emotion Recognition (ER) depends on audio and speech signals [15]. As argued by Purwins et al. [16], these signal streams, consisting of a one-dimensional time-series signal, are fundamentally different from the two-dimensional images typically used in computer vision, and they require capturing sequence information for effective representation. Therefore, it is crucial to examine how

existing CL methods perform in mobile sensing tasks and investigate the trade-offs among them.

Challenge 2: Resource Constraints

The limited system resources on mobile and IoT devices pose a significant challenge for designing and implementing CL and on-device training.

Memory/Storage Requirement: Unlike conventional model training on powerful GPU clouds with large memory and storage sizes (*e.g.,* 80 GB RAM and 1 TB storage), mobile and IoT platforms have severely limited memory and storage sizes, often in the range of 512 MB or less. Furthermore, the rise of the IoT has led to the deployment of billions of microcontrollers (MCUs) with even more constrained resources, such as 512 KB Static Random-Access Memory (SRAM) and 2 MB embedded flash memory. Existing CL methods [31, 35] and on-device training approaches [36], however, often demand prohibitively large memory footprint (*e.g.* 1 GB even using a small model with few hundred KBs [23]), making their deployment on such resource-limited devices extremely challenging. Yet, it is essential to facilitate billions of mobile and IoT devices being able to adapt to frequently changing environments without the help of powerful servers. Besides, updating models manually from scratch would be very tedious and time-consuming, considering a massive number of deployed devices. Hence, more intelligent and efficient on-device systems must be deployed in the real world and updated to the environment continuously.

Furthermore, among CL methods, the exemplar-based method (*i.e.* A CL method where a subset of past data samples, known as exemplars, is retained and used to prevent CF when training DNNs on new tasks) shows superior performance with an additional storage/memory usage for saving exemplars of the learned classes [31, 32], as will be discussed in Chapter 3. However, on resource-constrained devices with small memory and storage sizes, it has not been fully understood how CL methods ensure high performance under such extremely resource-constrained environments. Hence, it is critical to investigate the feasibility of CL with limited resources regarding memory and storage requirements.

Computational Cost: Both CL methods and on-device training heavily rely on computationally expensive backpropagation which includes forward and backward passes and weight updates, requiring at least three times more computation (measured in multiply-accumulate (MAC) operations) than inference, which involves only the forward pass [24]. This computational burden places a heavy load on the limited capabilities of edge devices, hindering the feasibility of on-device training. To address this, several lightweight training approaches have been proposed, such as fine-tuning only the last layer, however, these often lead to a considerable loss in accuracy [37, 38].

Moreover, CL methods introduce additional computational overheads beyond those of on-

device training, further intensifying the computational challenges due to the need to address the forgetting problem in DNNs. For example, Elastic Weight Consolidation (EWC) [30], a regularization-based CL method, requires the computation of Fisher information [39] to identify important parameters for existing classes. Similarly, Incremental Classifier and Representation Learning (iCaRL) [31], a rehearsal-based CL method, involves herding-based sampling to select representative samples of existing classes for replay while learning new classes. Thus, it is essential to develop compute-efficient CL and on-device training techniques and systems.

Latency and Energy Consumption: It is also crucial to consider the end-to-end latency and energy consumption of running CL and on-device training, as mobile and IoT devices are typically battery-powered. Many existing CL and on-device training methods incur excessive latency and energy consumption, which is unacceptable for real-world deployments [30, 31, 35]. Furthermore, Prabhu et al. [40] found that if the computation budget is constrained, many of the existing CL methods show substantial accuracy loss, unable to achieve the high CL accuracy as reported in their papers. Prior works of memory-efficient training by means of recomputation [22, 41, 42, 43] attempt to resolve the problem of the memory wall of on-device training. However, this approach trades off more computation for lower memory usage and incurs significant computation overhead, further increasing the already excessive training time and energy consumption on-device.

Challenge 3: Limited Labelled Data

CL and on-device training are further limited by the requirement for a relatively large amount of labelled data to learn new tasks or classes [29, 31, 32, 44]. In mobile sensing applications, it is much more difficult to collect labelled data from a continuous stream of user inputs compared to labelling static images [45]. Additionally, the more training data required for CL and on-device training, the more computational resources are needed, exacerbating the resource constraints mentioned earlier. Thus, it is essential to enable CL and on-device training with a small amount of labelled data to minimise expensive manual labelling, computational costs, and memory requirements.

1.3 Thesis and Substantiation

Motivated by these challenges, the main objective of this thesis is to ***design algorithms and systems that enable efficient CL and on-device training for mobile and IoT devices by drastically minimising resource requirements while maintaining high precision.*** The proposed algorithms and systems are evaluated in terms of accuracy, memory footprint, storage usage, latency, and energy consumption, targeting applications in computer vision, audio, human activity, and biosignal-based tasks.

To achieve this objective, this dissertation aims to answer the following research questions:

- ***Research Question 1****: To what extent are existing CL methods viable for sensor-based applications, where data modalities significantly differ from images? What strategies can be employed to reduce the resource overheads of representative CL methods without compromising their performance?*
- ***Research Question 2****: How can on-device AI be effectively implemented on extremely resource-limited devices like MCUs, ensuring efficient processing, minimal energy consumption, and reliable performance while supporting multi-tasking and adaptive system functionalities?*
- ***Research Question 3****: How can we jointly optimise data, memory, and computation aspects of CL and on-device training on both mobile and IoT devices?*

To answer these research questions and achieve the objective of this dissertation, we first investigate the advantages and disadvantages of existing CL methods in mobile sensing tasks. Based on the identified trade-offs and bottlenecks, we design a compression technique that optimises the computational costs and storage usage to overcome the strict resource constraints of mobile and embedded systems performing CL. Furthermore, we investigate the feasibility of operationalising various tasks of multi-modality data on severely resource-limited devices like MCUs. Further, we devise the on-device training framework that enables efficient training on MCUs that drastically minimises the requirements regarding the volume of labelled data, memory and computational costs, and latency and energy consumption. Finally, we leverage our developed techniques and frameworks, enabling a continual and on-device learning system that can be deployed on mobile and IoT devices with minimal human intervention (a few labelled samples) and system resources (low memory, compute, energy requirements).

1.4 Contributions and Thesis Outline

This dissertation begins with thoroughly reviewing the necessary preliminaries and theoretical backgrounds required to understand the research presented in the technical chapters (Chapters 3, 4, 5, and 6). Chapter 2 summarises essential concepts, research areas, and terminologies related to mobile and embedded sensing applications, continual learning, few-shot learning, model compression, and on-device training. The subsequent chapters describe three major contributions addressing the research questions posed earlier. The main contributions of this dissertation are outlined as follows.

Contribution 1: Initial Exploration of Continual Learning in Mobile Computing

We first conduct a systematic study to examine the CF problem and evaluate the advantages and disadvantages of current CL methods in mobile and embedded sensing applications. We then develop a new CL method that optimises the computational and storage requirements of one of the representative CL methods to overcome the strict resource constraints of mobile and embedded systems.

Chapter 3 analyses the feasibility and applicability of CL methods in various mobile sensing applications while considering the limitations posed by mobile and edge platforms, namely, low computational power, smaller memory and storage. Specifically, we adopt six CL methods from three different CL categories to evaluate their effectiveness and efficiency. We employ six datasets from three modalities of mobile sensing tasks such as (1) Human Activity Recognition (HAR) [46] based on accelerometer, gyroscope, and magnetometer data, (2) Gesture Recognition (GR) [47] based on surface electromyography (sEMG), and (3) Emotion Recognition (ER) [15] based on speech. Furthermore, we implement an end-to-end CL framework on two devices with different specifications: an Nvidia Jetson Nano GPU (used in mobile robotics and tablets) and a smartphone CPU. Through extensive evaluation, we find that the rehearsal-based CL approach (saving a small set of samples for rehearsal to preserve existing knowledge while learning new classes) often outperforms other CL approaches at the expense of larger system resources. This work allows us to understand the challenges and limitations of current CL methods when applied to mobile sensing applications on resource-constrained devices.

Based on the findings of the first work, we identify that one of the major bottlenecks in enabling end-to-end CL on-device is the expensive computational requirement for learning new user inputs/classes (*e.g.,* activities in HAR, gestures in GR). Furthermore, iCaRL (a rehearsal-based CL method that often outperforms other methods as studied in [7]) requires a large storage budget to store representative samples of learned classes. Motivated by these limitations, we propose a novel CL method, *FastICARL*, that improves upon iCaRL by reducing the CL time and alleviates the storage requirements for rehearsal samples. We optimise the construction process of an exemplar set (which accounts for most of the CL time) to shorten the CL time and address the limitation of computational overhead. In addition, to address the storage burden in resource-constrained devices, we further optimise FastICARL by applying quantisation on rehearsal samples to reduce the storage requirement. Moreover, we implement the end-to-end CL framework on embedded and mobile devices of two different specifications: Jetson Nano and a smartphone, respectively. To demonstrate its effectiveness and efficiency, we experiment with it in two audio sensing tasks: Emotion Recognition and Environmental Sound Classification (ESC) as case studies. Note that, as FastICARL builds directly on iCaRL, we focus on demonstrating its improvements within a specific domain while incorporating one more audio task (ESC).

Contribution 2: Bringing On-device Machine Learning from Edge to Microcontrollers

In recent years, the increasing need to make mobile and IoT devices intelligent has attracted much attention from academia and industry. This requires efficient CL and on-device training to facilitate various use cases (*e.g.* smart homes, smart buildings, and user personalisation). Therefore, in Chapters 4 and 5, we expand the scope of our research to investigate the extent to which we can bring deep learning capabilities from edge devices to MCUs, *i.e.*, extreme edge.

Chapter 4 focuses on supporting multi-task inference on MCUs, as multi-application systems capable of directly supporting a wide range of applications on-device would be more versatile and useful in practice. However, as described in Section 1.2, the resources of such systems are extremely limited (*e.g.* "high-end" MCU such as STM32F769 has only 512 KB SRAM and 2 MB Flash). Hence, maintaining a pre-trained model for each application is not scalable nor practical (applying scalar quantisation [48] on each model can mitigate memory/storage issues; however, the compression rate is low). Sharing network structures [49] can be a solution for *correlated and similarly structured models*; however, it is not feasible when systems need to operate *multiple heterogeneous models*.

We propose *YONO*, a product quantisation (PQ) [50] based approach that compresses multiple heterogeneous models and enables in-memory model execution and model switching for dissimilar multi-task inference on MCUs. YONO adopts PQ to learn codebooks that store the weights of different models and introduces a novel network optimisation and heuristics to maximise the compression rate and minimise the accuracy loss. We develop an online component of YONO for efficient model execution and switching between multiple tasks on an MCU at runtime without relying on external storage devices. Through extensive experiments, YONO demonstrates remarkable performance, compressing multiple heterogeneous models with negligible or no loss of accuracy up to 12.37×. Furthermore, YONO's online component enables efficient execution (latency of 16-159 ms and energy consumption of 3.8-37.9 mJ per operation) and reduces model loading/switching latency and energy consumption by 93.3-94.5% and 93.9-95.0%, respectively, compared to external storage access. Interestingly, YONO can compress various architectures trained with datasets that were not shown during YONO's offline codebook learning phase, showing the generalisability of YONO. To summarise, YONO shows great potential and opens further doors to enable multi-tasking systems on extremely resource-constrained devices.

Chapter 5 focuses on on-device training at the (extreme) edge, as it can benefit diverse real-world applications by dynamically adapting to new tasks and different (*i.e.* cross-domain/out-of-domain) data distributions from users while protecting the privacy of sensitive user data (*e.g.* healthcare). While on-device training becomes an essential building block to facilitate efficient CL, there are many challenges to enabling on-device

training at the (extreme) edge, as described in Section 1.2. In addition, prior works have several limitations: (1) substantial accuracy loss (≥10%), (2) excessively long training time, and (3) negligence of the data scarcity issues of real deployment scenarios.

To address these challenges and limitations, we propose *TinyTrain*, an on-device training approach that drastically reduces training time by selectively updating parts of the model and explicitly coping with data scarcity. TinyTrain introduces a task-adaptive sparse-update method that dynamically selects layers/channels based on a multi-objective criterion that jointly captures user data, memory, and compute capabilities of the target device. This leads to high accuracy on unseen tasks with reduced computation and memory footprint. TinyTrain outperforms vanilla fine-tuning of the entire network by 3.6-5.0% in accuracy while significantly reducing backward-pass memory and computation costs by up to 1,098× and 7.68×, respectively. Targeting broadly used real-world edge devices, TinyTrain achieves 9.5× faster and 3.5× more energy-efficient training over status-quo approaches, and 2.23× smaller memory footprint than SOTA, while remaining within the 1 MB memory envelope of MCU-grade platforms.

Contribution 3: Efficient Continual and On-device Training on Edge and Microcontrollers

In Chapter 6, after establishing essential components for efficient CL and on-device training on resource-constrained devices, we focus on integrating our proposed frameworks and techniques to enable efficient, adaptive, and evolving systems on mobile and IoT devices. In addition, we design a data- and system-efficient method tailored for CL. It overcomes the key limitation of many prior works of CL requiring a large amount of labelled data to learn new tasks/classes [29, 31, 32].

We design *LifeLearner*, a hardware-aware meta-continual learning system that drastically optimises system resources (lower memory, latency, energy consumption) while ensuring high accuracy. Specifically, LifeLearner exploits meta-learning and rehearsal strategies to explicitly cope with data scarcity issues and ensure high accuracy. It effectively combines lossless and lossy compression to significantly reduce the resource requirements of CL and rehearsal samples. We developed a hardware-aware system on embedded and IoT platforms, considering the hardware characteristics. As a result, LifeLearner achieves near-optimal CL performance, falling short by only 2.8% on accuracy compared to an Oracle baseline. With respect to the SOTA Meta CL method, LifeLearner drastically reduces the memory footprint (by 178.7×), end-to-end latency by 80.8-94.2%, and energy consumption by 80.9-94.2%. Additionally, we successfully deployed LifeLearner on two edge devices and an MCU, enabling efficient CL on resource-constrained platforms, where it would be impractical to run SOTA methods, and achieving the far-reaching deployment of adaptable CL in a ubiquitous manner.

Finally, Chapter 7 summarises the contributions presented in this dissertation, reflects on findings and insights drawn from the technical chapters, discusses their implications, and points out the potential avenue for further studies.

1.5 Publications and Author Contributions

The research efforts described in this dissertation have led to several publications at peer-reviewed international conferences.

1. Chapter 3 is based on two papers published at SEC '21 [7] and Interspeech '21 [51], respectively.
2. Chapters 4 and 5 build upon two papers published at IPSN '22 [52] and ICML '24 [23], respectively.
3. Chapter 6 is drawn from work published at SenSys '23 [53].

Additionally, I co-authored other works in broader areas of mobile systems, machine learning, and human-centred computing, which influenced the ideas and contributions of this dissertation while not directly related to it.

Works Related to This Dissertation

[7] *Exploring System Performance of Continual Learning for Mobile and Embedded Sensing Applications*
Young D. Kwon, Jagmohan Chauhan, Abhishek Kumar, Pan Hui, and Cecilia Mascolo.
Proceedings of the Sixth ACM/IEEE Symposium on Edge Computing, 2021. (SEC '21).
Best Paper Award

[51] *FastICARL: Fast Incremental Classifier and Representation Learning with Efficient Budget Allocation in Audio Sensing Applications*
Young D. Kwon, Jagmohan Chauhan, Cecilia Mascolo.
Proceedings of the Annual Conference of the International Speech Communication Association, 2021. (INTERSPEECH '21)

[52] *YONO: Modeling Multiple Heterogeneous Neural Networks on Microcontrollers*
Young D. Kwon, Jagmohan Chauhan, Cecilia Mascolo.
Proceedings of the 21st International Conference on Information Processing in Sensor Networks, 2022. (IPSN '22)

[23] *TinyTrain: Resource-Aware Task-Adaptive Sparse Training of DNNs at the Data-Scarce Edge*
Young D. Kwon, Rui Li, Stylianos I. Venieris, Jagmohan Chauhan, Nicholas D. Lane, and Cecilia Mascolo.

Proceedings of the Forty-first International Conference on Machine Learning (ICML '24). *Silver Medal at Samsung Best Paper Award 2024*

[53] *LifeLearner: Hardware-Aware Meta Continual Learning System for Embedded Computing Platforms*
Young D. Kwon, Jagmohan Chauhan, Hong Jia, Stylianos I. Venieris, and Cecilia Mascolo.
Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems, 2023. (SenSys '23).

Author Contributions Statement

For each of the five works related to this dissertation, I outline my contributions and the roles of my co-authors:

Exploring System Performance of Continual Learning for Mobile and Embedded Sensing Applications (SEC '21)

As the lead author, I conceptualised the research idea, designed and implemented the experimental framework, conducted most experiments, analysed the results, and wrote the majority of the manuscript. Dr. Jagmohan Chauhan provided guidance on the experimental design, implemented iCaRL on smartphone CPUs, produced latency results (Table 3.6 in Section 3.2.3), and contributed to the writing and revision of the paper. Abhishek Kumar assisted with the evaluation metrics and preliminary data analysis. Professors Pan Hui and Cecilia Mascolo supervised the research, provided critical feedback, and helped refine the manuscript.

FastICARL: Fast Incremental Classifier and Representation Learning with Efficient Budget Allocation in Audio Sensing Applications (INTERSPEECH '21)

I led this work by formulating the research problem, developing the FastICARL algorithm, implementing the system, conducting extensive experiments, and drafting the manuscript. Dr. Jagmohan Chauhan contributed to the experimental design and helped with the analysis of results. Professor Cecilia Mascolo provided supervision, critical insights, and guidance throughout the research process and manuscript preparation.

YONO: Modeling Multiple Heterogeneous Neural Networks on Microcontrollers (IPSN '22)

I led the entire research process, including conceptualisation of the YONO framework, algorithm design, system implementation, experimental evaluation, and manuscript writing. Dr. Jagmohan Chauhan contributed to the discussion of the methodology and assisted

with manuscript editing. Professor Cecilia Mascolo provided research supervision, critical feedback on the approach, and helped with manuscript revisions.

TinyTrain: Resource-Aware Task-Adaptive Sparse Training of DNNs at the Data-Scarce Edge (ICML '24)

I conceptualised the research idea, developed the TinyTrain approach, implemented the framework, conducted experiments, and wrote the majority of the paper. Dr. Rui Li and Dr. Stylianos Venieris contributed to the algorithm design and system optimisation. Dr. Jagmohan Chauhan assisted with the experimental setup and results analysis. Professors Nicholas Lane and Cecilia Mascolo supervised the work, provided guidance throughout the research process, and helped refine the manuscript.

LifeLearner: Hardware-Aware Meta Continual Learning System for Embedded Computing Platforms (SenSys '23)

I led this research by formulating the LifeLearner system, designing and implementing the algorithms, conducting experiments on various hardware platforms, and writing the manuscript. Dr. Jagmohan Chauhan and Hong Jia contributed to the experimental design and assisted with the manuscript preparation. Dr. Stylianos Venieris provided insights on hardware-aware optimisation and contributed to manuscript editing. Professor Cecilia Mascolo supervised the research, provided guidance on the research direction, and helped finalise the manuscript.

For all the publications, I performed the primary role in conceptualising the research ideas, developing methodologies, implementing systems, conducting experiments, analysing results, and preparing manuscripts. My co-authors provided invaluable guidance, feedback, and assistance throughout the research process, helping to refine the ideas and improve the quality of the work.

Other Works

[54] *TinyTTA: Efficient Test-time Adaptation via Early-exit Ensembles on Edge Devices*
Hong Jia, **Young D. Kwon**, Alessio Orsino, Ting Dang, Domenico Talia, and Cecilia Mascolo.
Proceedings of the Thirty-Eighth Annual Conference on Neural Information Processing Systems (NeurIPS '24).

[55] *UR2M: Uncertainty and Resource-Aware Event Detection on Microcontrollers*
Hong Jia, **Young D. Kwon**, Dong Ma, Nhat Pham, Lorena Qendro, Tam Vu, and Cecilia Mascolo.
Proceedings of the 2024 IEEE International Conference on Pervasive Computing and Communications (PerCom '24).

[56] *ChatGPT in education: A blessing or a curse? A qualitative study exploring early adopters' utilization and perceptions*
Reza Hadi Mogavi, Chao Deng, Justin Juho Kim, Pengyuan Zhou, **Young D. Kwon**, Ahmed Hosny Saleh Metwally, Ahmed Tlili, Simone Bassanelli, Antonio Bucchiarone, Sujit Gujar, Lennart E Nacke, Pan Hui.
Computers in Human Behavior: Artificial Humans, 2024.

[57] *Enabling On-Device Smartphone GPU based Training: Lessons Learned*
Anish Das, **Young D. Kwon**, Jagmohan Chauhan, and Cecilia Mascolo.
Proceedings of the 2022 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom '22 Workshops)

[58] *Exploring On-Device Learning Using Few Shots for Audio Classification*
Jagmohan Chauhan, **Young D. Kwon**, and Cecilia Mascolo.
Proceedings of the 30th European Signal Processing Conference, 2022. (EUSIPCO '22)

[59] *PROS: an Efficient Pattern-Driven Operating System for Low-Power Healthcare Wearables*
Nhat Pham, Hong Jia, Minh Tran, Tuan Dinh, Nam Bui, **Young D. Kwon**, Dong Ma, VP Nguyen, Cecilia Mascolo, and Tam Vu.
Proceedings of the 28th Annual International Conference on Mobile Computing and Networking, 2022. (MobiCom '22)

[60] *MyoKey: Inertial Motion Sensing and Gesture-based QWERTY Keyboard for Extended Realities*
Kirill Shatilov, **Young D. Kwon**, Lik-Hang Lee, Dimitris Chatzopoulos, and Pan Hui.
IEEE Transactions on Mobile Computing (TMC), 2022

[61] *Causal Analysis on the Anchor Store Effect in a Location-based Social Network*
Anish Krishna Vallapuram, **Young D. Kwon**, Lik-Hang Lee, Fengli Xu, and Pan Hui.
Proceedings of the 2022 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM '22).

[62] *Hidenseek: Federated lottery ticket via server-side pruning and sign supermask*
Anish K. Vallapuram, Pengyuan Zhou, **Young D. Kwon**, Lik-Hang Lee, Hengwei Xu, Pan Hui.
arXiv preprint, 2022.

[17] *ContAuth: Continual Learning Framework for Behavioral-based User Authentication*
Jagmohan Chauhan, **Young D. Kwon**, Cecilia Mascolo, and Pan Hui.
Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT/UbiComp), 2021

[63] *Aquilis: Using Contextual Integrity for Privacy Protection on Mobile Devices*

Abhishek Kumar, Tristan BRAUD, **Young D. Kwon**, and Pan Hui.
Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT/UbiComp), 2021

[64] *Interpretable Business Survival Prediction*
Anish Krishna Vallapuram, Nikhil Nanda, **Young D. Kwon**, and Pan Hui.
Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM '21).

[65] *IAN: Interpretable Attention Network for Churn Prediction in LBSNs*
Liang-Yu Chen, Yutong Chen, **Young D. Kwon**, Youwen Kang, and Pan Hui.
Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM '21).

[66] *Knowing when we do not know: Bayesian continual learning for sensing-based analysis tasks*
Sandra Servia-Rodriguez, Cecilia Mascolo, **Young D. Kwon**.
arXiv preprint, 2021.

Chapter 2

Background

In the previous chapter, we have highlighted the potential and challenges of applying CL and on-device training on mobile and IoT devices. This chapter provides a comprehensive overview of previous studies and background information relevant to the works discussed in this dissertation. We begin by examining mobile and embedded sensing applications (Section 2.1) and CL (Section 2.2). Subsequently, we describe various techniques that enable efficient ML on-device (Section 2.3). Finally, we discuss the literature on on-device training (Section 2.4).

2.1 Mobile and Embedded Sensing Applications

Deep learning is increasingly being applied in mobile and embedded systems as it achieves SOTA performances on many sensing applications such as computer vision applications [67], activity recognition [6, 33], gesture recognition [68], and audio sensing [69]. First of all, there exist many vision applications, ranging from image classification [1, 70], video analytics [71], to augmented reality [72], autonomous driving [73] and traffic sign analysis [74]. Ren et al. [75] developed efficient and distributed object detection systems for real-time surveillance applications based on Faster R-CNN [76]. Additionally, prior work [77] established a client-edge collaboration framework and solved an optimisation problem of offloading to minimise the response time. Furthermore, many prior works [73, 78, 79] focused on autonomous driving. SqueezeDet [78] was proposed to achieve high object detection accuracy with minimal model size for energy efficiency. Chen et al. [79] proposed multi-view 3D networks that process LIDAR point cloud data and camera images to predict 3D boundaries.

Human Activity Recognition (HAR) is one of the most extensively studied mobile sensing applications [33, 80, 81, 82, 83]. HAR aims to automatically determine various human activities (*e.g.,* from simple actions like walking to complex activities like cooking a meal)

based on smartphone and wearable (body-worn) Inertial Movement Unit (IMU) sensors. Hammerla et al. [33] experimented with three variants of deep learning approaches such as feed-forward, convolutional, and recurrent neural networks on HAR datasets, and presented guidelines for training neural networks. Moreover, Ordóñez and Roggen [84] proposed the DeepConvLSTM model, in which convolutional layers extract features from raw IMU data, and Long Short-Term Memory (LSTM) recurrent layers capture temporal dynamics of feature activations to improve HAR performance. Murahari and Ploetz enhanced DeepConvLSTM by employing attention mechanisms to identify crucial time steps [81]. Zhang et al. [83] enriched the domain diversity by synthesising intra- and inter-domain style data while maintaining robustness to class labels to improve domain generalisation of HAR.

Another frequently used application in mobile sensing is the recognition of hand gestures (*e.g.,* fist and open palm) using surface Electromyography (sEMG) signals generated during muscle contractions [10, 34, 85, 86, 87]. sEMG signals are quantitatively measured in a non-invasive fashion by estimating the electrical potential differences between muscle and ground electrodes. These signals are used in various fields, including medical applications [88], rehabilitation [89], human-computer interactions [90], and upper-limb prostheses control [91]. Zhai et al. [47] proposed a self-recalibrating framework that can be updated to maintain model performance, eliminating the need for additional user labels for re-training. Shatilov et al. [92] developed a low-cost and adaptable system for a prosthetic hand by incorporating an sEMG sensor, a mobile phone, a cloud component, and a 3D-printed arm. Also, Becker et al. [87] used sEMG of the forearm to classify finger touches with their proposed neural architecture, combining convolutional, feed-forward, and LSTM layers. Shin et al. [93] developed a wireless multimodal wearable system based on EMGs and sounds for automatic, accurate clinical assessment of silent aspirations from dysphagia patients.

Audio sensing applications are also foundational in mobile sensing research, providing behavioural insights to users. These applications include Emotion Recognition (ER) [15], Speaker Identification [94], Speaker Counting [95], Environmental Sound Classification (ESC) [96], and Conversation Analysis [97], and Keyword Spotting (KWS) [98, 99]. Lu et al. [100] proposed SoundSense, a scalable framework designed to recognise meaningful sound events that occur in users' everyday lives. The authors developed the general-purpose sound sensing system to operate on resource-limited phones. Lee et al. [97] developed SocioPhone, a new mobile platform for face-to-face interaction monitoring to facilitate group conversations by utilising meta-linguistic contexts. Moreover, Georgiev et al. [101] proposed a deep learning modelling and optimisation framework that explicitly targets various audio sensing tasks in resource-constrained embedded systems. Lee et al. [102] developed a soft smart biopatch that can continuously collect biometric data from an individual's heart sound for authentication.

Looking beyond individual applications, a clear evolutionary trajectory emerges in mobile sensing research. Over the past decade, we have witnessed a gradual shift from cloud-dependent systems, where mobile devices merely collect data to be processed remotely, toward on-device edge intelligence. This progression has been driven not only by technological advances in hardware efficiency but also by fundamental requirements of real-world deployments: the need for privacy preservation, real-time responsiveness, and operation in connectivity-constrained environments. This shift shows the importance of on-device adaptation capabilities that form the focus of this dissertation.

2.2 Continual Learning

Continual Learning (CL) studies the ability to learn over time from an incoming stream of data by incorporating new knowledge while retaining previously learned information [7, 29, 40, 103, 104]. It is also referred to as incremental learning (IL) [31], lifelong learning [29], and sequential learning [27]. In a CL setup, learning methods typically suffer from CF [27, 28], where a learned model experiences performance degradation on previously learned task(s) (*e.g.,* task A) as information relevant to a new task (*e.g.,* task B) is incorporated. It occurs because the learned parameters of the network, optimised to perform well in task A (important weights to task A), are changed to maximise/minimise the objective/loss of task B. At its core, CL addresses the fundamental *stability-plasticity dilemma* that has long challenged research [27, 28, 105, 106]: how can a model remain sufficiently plastic to incorporate new information while ensuring enough stability to retain existing knowledge?

In recent years, to resolve CF and the stability-pasticity dilemma, researchers proposed various CL approaches that represent different philosophical stances. These approaches can be categorised into four main groups:

1. **Regularisation-based Methods** [30, 66, 107, 108, 109, 110, 111]: These methods explicitly prioritise stability by constraining weight updates. Specifically, they add regularisation terms to the loss function to minimise changes to important weights of a model for previous tasks, thereby preventing forgetting. For instance, Kirkpatrick et al. [30] proposed Elastic Weight Consolidation (EWC), which uses the Fisher matrix to estimate the importance of each weight in a model and adjusts the update of the weights based on their importance. Additionally, Zenke et al. [107] proposed Synaptic Intelligence (SI), which maintains an online estimate of the weights' importance with respect to their contribution to the change in the loss function.

2. **Replay with Exemplars-based Methods** [31, 32, 51, 53, 112, 113, 114, 115, 116, 117]: These techniques attempt to simulate joint training on past and present data. They require training data from the new class and also few training samples from

earlier classes to update the model. Rebuffi et al. [31] proposed an Incremental Classifier and Representation Learning (iCaRL) that stores a small set of raw data from previous tasks as exemplars to prevent forgetting and to learn new tasks. In addition, Lopez-Paz and Ranzato [112] proposed Gradient Episodic Memory (GEM) that utilises episodic memory to enable CL by minimizing negative backward transfer (*i.e.,* forgetting).

3. **Replay-based Methods** [118, 119]: Different from Replay with Exemplars-based methods, Replay-based methods update model parameters by complementing the training data for each new task with *"pseudo-data"* representative of previous tasks. Li and Hoiem [118] developed Learning without Forgetting (LwF), which labels the input data of the current task using the model trained on the previous tasks and uses them as pseudo-data. The replayed data helps the model's output remain close to that of the model trained on previous tasks. Also, Shin et al. [119] proposed a Generative Adversarial Networks (GAN) [120] based CL method, utilising a GAN to generate data from past experiences when learning new data.

4. **Dynamic Architecture-based Methods** [121, 122, 123, 124, 125]: These methods try to circumvent the dilemma by allocating distinct resources for new knowledge. In detail, they dynamically expand and freeze DNN architectures to incorporate new classes and prevent forgetting, respectively. For instance, Yoon et al. [122] proposed Dynamically Expandable Network (DEN), which dynamically expands network capacity for each task with minimal addition of neurons using an overlapping knowledge-sharing structure among tasks. Hung et al. [123] applied model compression, critical weights selection, and progressive network expansion whenever it is needed to avoid forgetting and maintain the compactness of the model. Also, Yan et al. [124] introduced a two-stage learning approach that leverages a dynamically expandable representation for more effective incremental concept modeling.

Despite promising performance, dynamic architectures pose significant challenges, particularly for on-device implementation. They require costly modifications to the model architecture, leading to higher computational costs as the model expands. Furthermore, commonly used on-device ML frameworks such as DeepLearning4j, Mobile Neural Network (MNN) [126], and TensorFlow Lite for Microcontrollers (TFLM) [127] (adopted in our implementation of on-device systems for mobile and IoT devices) are limited in supporting dynamic modifications of the network architecture on-device as they prohibit the utilisation of compile-time optimisations on a fixed computation graph of the model. Consequently, this dissertation excludes dynamic architecture-based methods from its scope of investigation, focusing instead on the first three categories of CL approaches in subsequent chapters.

While current CL methods addressing CF are empirically evaluated using small and large

datasets [21, 128], these studies have limitations. They either neglect resource constraints of mobile and embedded devices with respect to storage and latency [21] or adopt only a few methods [21, 128]. To fill this gap, we investigated to what extent current CL methods can enable practical CL systems for mobile and embedded sensing applications on-device and examined the implications of such systems regarding performance, efficiency, and generalisability. Furthermore, to fully understand the issue of CF in mobile sensing, where the data modality differs significantly from the image datasets [16] typically used to evaluate CL methods, we assessed six representative CL methods across three embedded sensing applications (*e.g.,* HAR, GR, and ER) with different data modalities (*e.g.,* accelerometer, sEMG, and speech). Moreover, we implemented an end-to-end CL framework that addresses resource constraints of mobile devices by minimising required CL time and storage for exemplars crucial for real-world user applications (see Chapter 3).

2.2.1 Few-Shot Learning

In real-world deployment scenarios, obtaining a large volume of labelled training data to update deployed models or add new classes is challenging. This is primarily because users are typically willing to annotate only a few samples for new classes. As we discussed in Chapter 1, the annotation problem is even more pronounced in mobile sensing applications, as labelling a stream of signals is not straightforward. Therefore, developing Few-Shot Learning (FSL) techniques is a natural fit for efficient CL and on-device training [129]. FSL methods aim to learn a target task given a few examples (*e.g.* 5-30 samples per class) by transferring knowledge from large source data (*i.e.* meta-training) to scarcely annotated target data (*i.e.* meta-testing).

To date, several FSL schemes have been proposed, ranging from gradient-based [130, 131, 132], metric-based [133, 134, 135] to Bayesian-based approaches [136]. Recently, there has been growing interest in cross-domain (out-of-domain) FSL (CDFSL) [137]. The CDFSL setting dictates that the source (meta-train) dataset differs significantly from the target (meta-test) dataset. Although CDFSL is more challenging than the standard in-domain (*i.e.* within-domain) FSL [36], it tackles more realistic scenarios, which align closely with the real-world deployment scenarios targeted by this dissertation.

In our work, we focus on realistic use-cases where the available source data (*e.g.* Mini-ImageNet [2]) are significantly different from target data (*e.g.* meta-dataset [138]) with limited samples such as 5-30 samples per class. While FSL-based methods address data efficiency, they often neglect the memory and computation constraints of on-device training. Therefore, we explore the joint optimisation of three major pillars of on-device training, such as data, memory, and computation (Chapters 5 and 6).

2.2.2 Meta-Continual Learning

Despite the scarcity of labelled user data on devices, conventional CL methods [29] still require much training data to prevent CF while learning new classes. To overcome this limitation, Meta CL or Few-shot Class-Incremental Learning (FSCIL) [139, 140, 141, 142, 143, 144] has been proposed. This method aims to address two challenges: (i) to avoid CF of old classes and (ii) to prevent overfitting to a few samples of new classes.

Tao et al. [143] proposed a novel FSCIL framework that employs a neural gas (NG) network [145] to learn the topology of feature space to represent learned knowledge and solves FSCIL problem by adjusting the update of the NG network. In addition, Javed and White [140] demonstrated that their proposed method, Online-aware Meta-Learning (OML), can successfully avoid interference and ensure learning of new knowledge for up to 200 classes by continually learning through a meta-objective. Beaulieu et al. [139] extended the work of Javed and White [140] and proposed A Neuromodulated Meta-Learning (ANML) that learns selective activation of another network using a neuromodulatory (NM) network. Beaulieu et al. showed that ANML could successfully solve CF problems for up to 600 classes. Furthermore, an Attentive Independent Mechanisms (AIM) module [35] was proposed to capture independent concepts for learning new knowledge. AIM and its combinations, ANML+AIM and OML+AIM, have achieved SOTA results.

Although prior works in Meta CL enable CL with limited data samples, they have certain limitations. For example, OML [140] and ANML [139] can retain high CL performance on the Omniglot dataset [146] over many classes. However, these methods (including AMML+AIM and OML+AIM) fail to generalise to other datasets, showing low accuracy on CIFAR-100 [1] and have extremely high memory requirements (as detailed in Chapter 6), which limits their applicability to low-end devices. Hence, we aim to design an efficient Meta CL system that achieves high accuracy and minimal forgetting while making practical deployment on embedded devices a reality (Chapter 6).

2.3 Efficient Machine Learning

While deep learning models continue to demonstrate impressive performance, this often comes at the cost of increasingly complex network architectures and vast amounts of training data [147]. Although higher performance is critical for many applications, increased model complexity demands significantly more computation resources and often exceeds the hardware memory limit of resource-constrained devices such as mobile and embedded devices or MCUs [148]. Consequently, many researchers have focused on developing methods to improve efficiency without compromising model accuracy.

2.3.1 Model Compression

Researchers have explored various approaches for model compression:

1. **Efficient Network Architectures:** Many studies have focused on designing and hand-drafting more efficient network architectures, namely, SqueezeNets [149, 150], ShuffleNets [151, 152], and MobileNets [153, 154]. Howard et al. [153] proposed MobileNet, a new network architecture, that utilises depth-wise separable convolutions. MobileNet is based on depth-wise separable convolutions which factorize a standard convolution into a depthwise convolution and a 1×1 convolution (pointwise convolution), making MobileNet consumes $8 - 9$ times less computation than standard convolutions. Sandler et al. [154] designed a novel layer called the inverted residual with a linear bottleneck, on which MobileNetV2 is based, further improving the performance and efficiency over the original MobileNet.

2. **Neural Network Search (NAS):** Recently, NAS methods [148, 155, 156, 157, 158, 159, 160, 161, 162] have gained popularity in designing efficient neural architectures by automating the process of finding network structures. NAS often achieves better efficiency than hand-drafted neural architectures. For instance, Liu et al. [155] introduced a novel algorithm for differentiable NAS based on bilevel optimisation, which can be applied to both convolutional and recurrent layers. Also, Cai et al. [158] developed a progressive shrinking algorithm that first trains a large neural network and then progressively fine-tunes the network to support smaller sub-networks that share weights with the larger ones.

3. **Weight Pruning:** Another thread of research is weight pruning methods that leverage the inherent redundancy in the weights of neural networks [163, 164, 165, 166, 167, 168, 169, 170, 171, 172]. Han et al. [163, 164] proposed a iterative pruning framework that removes weights with small magnitudes and retrains the model until it reaches a certain threshold in terms of performance or model size. In addition, Yang et al. [165] introduced energy-aware pruning, while Liu et al. [167] utilised structural regularity of neural networks and an Alternating Direction Method of Multipliers (ADMM)-based pruning framework [169], showing impressive results.

4. **Quantisation:** Quantisation of model weights and activiations has been an active area of research. Many prior works quantise the weights and activations from 32-bit float to 8-bit integer [48], ternary values (2-bit) [173, 174], binary values (1-bit) [175, 176, 177, 178], and mixed precision [179, 180]. Also, weight clustering methods have been proposed to group weights into several clusters to compress a model. Chen et al. [181] proposed a hashing trick to group each weight, and Han et al. [164] incorporated weight clustering into their iterative pruning framework. Moreover, researchers studied techniques that quantise an array of scalars of the weights to compress a model or a particular layer. Some works extended a sparse

coding [182] to learn a compact representation that covers the feature space of weights of a model [183, 184]. Bagherinezhad et al. [184] proposed LCNN, lookup-based convolutional neural network, encoding convolutions by a few lookups to a learned dictionary that covers weights' feature space. However, while sparse coding-based methods could well represent a feature space of weights, it requires non-trivial memory to store the learned dictionary and its indices of weights to the dictionary.

5. **Vector Quantisation:** Researchers have examined vector quantisation-based methods for model compression. Gong et al. [185] conducted an empirical study comparing binarized networks, scalar quantisation using k-means (i.e., weight clustering), Product Quantisation (PQ) [50]. The authors found that PQ shows the best result among the studied methods. Wu et al. [147] showed $4-6\times$ speed-up and $15-20\times$ compression with one percent loss of accuracy by applying PQ together with an error-correction mechanism. In addition, Hayes et al. [32] utilised PQ to compress an intermediate layer of a network as exemplars to maintain high CL performance. Chen et al. [186] proposed a differentiable PQ to compress an embedding layer in Natural Language Processing, which is typically very large in terms of size.

As shown in the works described above, the vector quantisation technique (PQ) demonstrates effectiveness and potential in weight compression. Nevertheless, prior works are limited in the utilisation of PQ to a single model or a layer (typically on a scale of tens of millions of weights). Given that PQ is commonly used to index billions of vectors for approximate nearest neighbour search in the database community [50, 187, 188], we argued that PQ has not been fully utilised in model compression. Also, PQ can be employed to compress multiple heterogeneous networks [189]. Also, the learned PQ codebook might generalise well to unseen model weights. Therefore, we explore a novel model compression methodology based on PQ in Chapter 4. It is worth noting that the scope of this chapter is limited to lossy compression to maximise the compression rate (although lossy compression can incur accuracy loss), as Ko et al. [190] showed that the lossless compression technique alone can only achieve a relatively low compression rate ($\sim 2\times$) without accuracy loss.

2.3.2 Multi-Task Learning

Multi-task learning allows for the simultaneous learning of correlated tasks such that the accuracy of both or one of the tasks is improved by exploiting the similarities and differences across tasks [191]. Prior works include low-rank parameter search [192, 193], common feature learning [194, 195], task clustering [196, 197], and task relation learning [198, 199]. These works achieve limited compression by sharing the first few network layers. However, their main goal is to increase the robustness and generalisation of multiple task learners. Thus, keeping multiple heterogeneous DNN models in the extremely limited memory of embedded devices, along with managing and executing these models (achieving different tasks) efficiently at run-time, are challenging to the aforementioned works.

Neural Weight Virtualisation (NWV) [189] was introduced to compress multiple heterogeneous models of different network architectures and tasks on mobile and IoT devices. NWV minimises the context switching overhead by retaining all shared weights in memory. However, NWV's compression ratio is constrained to 8.08×, limiting the multi-tasking IoT system with a small memory footprint to operate many tasks in real-time. Also, it only employs a simplified LeNet architecture in the experiments of IoT use cases, thus limiting the accuracy of the system.

To overcome the limitations, we developed YONO, which enables the efficient execution of multiple DNN models while remaining within the limited resource constraints on embedded devices. YONO not only increases compression rates with highly optimised models (*e.g.,* MicroNet, DS-CNN) but also achieves high accuracy that is useful in practice (Chapter 4).

2.4 On-Device Training

Scarce memory and compute resources are major bottlenecks in deploying DNNs on tiny edge devices. In this context, researchers have largely focused on optimising *the inference stage* (*i.e.* forward pass) by proposing lightweight DNN architectures [150, 152, 154], pruning [164, 167], and quantisation methods [48, 177, 200] (refer to Section 2.3.1 for more details). Researchers have also investigated how to efficiently leverage heterogeneous processors [201, 202, 203], and offload computation [204]. Driven by increasing privacy concerns and the need for post-deployment adaptability to new tasks or users, the research community has recently turned its attention to enabling DNN *training* (*i.e.,* backpropagation having both forward and backward passes, and weights update) at the edge.

Researchers proposed memory-saving techniques to resolve the memory constraints of training [205, 206, 207, 208, 209]. For example, gradient checkpointing [41, 210, 211] discards activations of some layers in the forward pass and recomputes those activations in the backward pass. Microbatching [212] splits a minibatch into smaller subsets that are processed iteratively, to reduce the peak memory. Swapping [213, 214] offloads activations or weights to an external memory/storage (*e.g.* from GPU to CPU or from an MCU to an SD card). Some works [22, 42, 43] proposed a hybrid approach by combining two or three memory-saving techniques. Although these methods reduce the memory footprint, they incur additional computation overhead on top of the already prohibitively expensive on-device training time at the edge.

A few existing works [44, 215, 216, 217, 218] have also attempted to optimise both memory and computations. By selectively updating only a subset of layers and channels during on-device training, these methods effectively reduce both the memory and computation load. However, *TinyTL* still demands excessive memory and computation, as shown in Chapter 5. *p-Meta* enables pre-selected layer-wise updates learned during offline meta-

training and dynamic channel-wise updates during online on-device training. However, as *p-Meta* requires additional learned parameters such as a meta-attention module identifying important channels for every layer, its computation and memory saving are relatively low. For example, *p-Meta* still incurs up to 4.7× higher memory usage than updating the last layer only. Furthermore, as shown in Chapter 5, the performance of *SparseUpdate* drops dramatically up to 7.7% when applied at the edge where data availability is low. This occurs because the approach requires access to the entire target dataset (*e.g. SparseUpdate* [44] uses the entire CIFAR-100 dataset [1]), which is unrealistic for such devices in the wild. More importantly, it requires a large number of epochs (*e.g. SparseUpdate* requires 50 epochs) to reach high accuracy, which results in an excessive training time of up to 10 days when deployed on tiny edge devices, such as STM32F746 MCUs. In addition, many methods [44, 216, 218] are unable to adapt dynamically to target data because they require running *a few thousands of* computationally heavy searches [44], pruning processes [216], or pre-selecting layers to be updated [218] on powerful GPUs to identify important layers/channels for each target dataset during the offline stage before deployment. As such, the current *static* layer/channel selection scheme cannot be adapted on-device to match the properties of the user data and hence remains fixed after deployment, which may lead to a suboptimal accuracy.

To overcome these limitations, we proposed an efficient on-device training method, TinyTrain, that drastically minimises memory and computation while achieving SOTA accuracy given scarce target data, enabling data-, compute-, and memory-efficient training on tiny edge devices (Chapter 5).

2.5 Summary

The literature reviewed in this chapter reveals several important trends that frame the contributions of this dissertation. First, there is a growing convergence between previously distinct research areas–CL, few-shot learning, model compression, and on-device training–driven by the practical requirements of real-world mobile applications. Second, while significant progress has been made in each individual area, holistic approaches that jointly optimise across multiple constraints remain underexplored. Finally, there exists a persistent gap between theoretical capabilities demonstrated in controlled research environments and real-world systems that can operate effectively within the strict constraints of mobile and IoT devices. This dissertation addresses these gaps through the following contributions:

1. Conducting a comprehensive empirical study of various representative CL methods to better understand the feasibility of CL on multiple data modalities on resource-constrained devices, followed by developing an efficient CL method (Chapter 3).
2. Designing a novel compression method that minimises model size and enables efficient

model execution and switching to benefit multi-user and multi-application setups (Chapter 4).

3. Proposing an efficient on-device training approach that drastically minimises the data, memory, and computation requirements to support efficient CL on extremely resource-constrained devices (Chapter 5).

4. Leveraging our previous efforts in enabling efficient and adaptive systems to develop a data-, memory-, and compute-efficient CL system that can operate on both embedded and IoT devices, where it is impractical and/or unable to run SOTA, respectively (Chapter 6).

Chapter 3

Initial Exploration of Continual Learning in Mobile Computing

3.1 Introduction

In Chapter 1, we discussed the untapped potential of continual learning and on-device training to support various real-world applications in mobile computing. Further, we described the associated challenges in Chapter 1 and pointed out the limitations of prior research in Chapter 2. Then, this chapter presents our initial investigation into the Catastrophic Forgetting (CF) problem, as described in Section 1.2, and examines the advantages and disadvantages of current CL methods in mobile and embedded sensing applications (Section 3.2). After that, based on our investigation, we developed a novel CL method that remarkably reduces the resource requirements of the representative CL method, making CL practically useful on mobile and embedded systems (Section 3.3).

As discussed in Chapter 1, in practice, enabling deep learning models to continually learn is challenging due to the CF problem. Since the identification of CF in Multi-Layer Perceptrons (MLPs), researchers have proposed various methods to mitigate it [30, 31, 112, 118, 122] and evaluate it using both small and large datasets [21, 128, 219] (refer to Section 2.2 for further details). However, these methods have primarily been evaluated in computer vision applications using MLPs or Convolutional Neural Networks (CNN) based deep learning models. *It remains unclear whether these methods are viable in sensor-based applications, where data modality is significantly different from images, and sequence information needs to be captured [16].* Moreover, *most of the existing Incremental Learning (IL)*[1] *techniques [220] do not take into account the resource requirements of mobile and embedded devices*, potentially limiting their applicability. There is a clear need

[1] In this work, we use continual learning (CL) and incremental learning (IL) interchangeably.

to understand the resource consumption limitations of existing CL methods to assess their suitability for resource-constrained edge platforms.

To address the aforementioned limitations, we conducted the first systematic study to investigate the CF problem on mobile and embedded sensing applications using various IL methods. **First,** we employ three datasets from the widely researched application of Human Activity Recognition (HAR) [46] based on accelerometer, gyroscope, and magnetometer data. Next, we include two datasets from Gesture Recognition (GR) [47] based on surface electromyography (sEMG). We further incorporate an Emotion Recognition (ER) dataset [15] based on speech among audio sensing tasks to make our results generalisable to different data modalities across diverse applications. **Second,** we examine trade-offs of studied IL methods in terms of their performance, storage footprint, computational costs, and peak memory limit to assess their feasibility on mobile and embedded devices. To investigate the system limitations imposed by different configurations of IL, we implemented the IL framework on two device types with different specifications: an Nvidia Jetson Nano GPU (used in mobile robotics and tablets) and a smartphone CPU.

Overall, the major contributions and findings of our systematic investigation are:

- We conduct a comprehensive study of the CF problem in mobile and embedded applications using six representative IL methods falling under three paradigms: **regularisation** ((1) Elastic Weight Consolidation: EWC [30], (2) Synaptic Intelligence: SI [107], and (3) Online EWC [108]), **replay** ((4) Learning without Forgetting: LwF [118]), and **replay with exemplars** ((5) Incremental Classifier and Representation Learning: iCaRL [31] and (6) Gradient Episodic Memory: GEM [112]).
- To evaluate CF in real-life scenarios, we employ Sequential Learning Tasks (SLTs), successively learning two or more sub-tasks $D_1, ..., D_k$, instead of learning a single task D [21]. Learning new tasks continuously becomes vital since the number of classes (activities or users) and the environments of edge applications often change over time. We adopt a class-incremental learning setup with three scenarios of increasing complexity (see Section 3.2.1 for detail). Through extensive experiments, we find that all IL methods perform well in simple scenarios; only iCaRL maintains strong performance in complex scenarios, primarily due to its use of exemplar samples. To our knowledge, we are the first to train and implement IL methods to run on mobile and embedded systems, with the aim to build an end-to-end on-device CL system and to evaluate trade-offs of studied IL methods in terms of their performance, storage, and computational costs, as well as the peak memory usage.
- Our analysis shows that iCaRL and GEM require a modest amount of storage, (*i.e.,* 2 MB–115 MB for 20% - 40% of training samples). However, GEM and EWC-based algorithms are computationally expensive, with average IL time ranging from 46.3–2,660 seconds on Jetson Nano. For all other algorithms, it ranged from 8.46–150

seconds on both Jetson Nano and a smartphone. Moreover, our study shows that simple deep learning architectures such as one and two-layer long short-term memory (LSTM) [221] can be trained entirely on the smartphone, thereby ensuring complete user privacy.

- Based on our findings, we present a series of lessons and guidelines to assist practitioners and researchers in applying continual deep learning to mobile sensing applications.

Our initial exploration of CL in mobile computing revealed opportunities for further system optimisations. We identified two main bottlenecks of prior works in applying CL on mobile and embedded systems. First, it is challenging to enable CL on-device since CL methods are computationally heavy. Second, exemplar-based methods require storing exemplars, which can impose a considerable burden on resource-constrained systems.

Moreover, many techniques have been proposed to facilitate efficient ML systems on resource-constrained devices, as described in Chapter 2. Quantisation and low-bit precision of model parameters are utilised to reduce the size of the model [48, 176]. Low-rank factorisation [222, 223] and pruning [164] have been proven effective in reducing model size, while retaining accuracy (refer to Section 2.3). IL with optimisations that allow its use on-device, however, has never been explored in the context of audio-based applications.

To address these issues, we developed FastICARL, an end-to-end framework to enable efficient and accurate on-device IL in two audio sensing applications. FastICARL improves upon the representative exemplar-based IL method, iCaRL, as we observed that iCaRL consistently outperforms other IL methods [107, 108]. First, we optimise the construction process of an exemplar set (which takes most of IL time) to reduce IL time. Second, we optimise FastICARL by applying quantisation to exemplars to decrease the storage requirement. Furthermore, we implement FastICARL on both Jetson Nano and a smartphone, employing MNN [126] for complete on-device training of new tasks/classes.

Overall, the major contributions and findings of FastICARL are as follows.

- We design, implement, and evaluate FastICARL, which overcomes the limitations of the prior work.
- FastICARL effectively mitigates the CF issue in audio-based datasets, achieving 69% and 71% weighted F1-scores for ER and ESC, respectively.
- FastICARL reduces the latency of exemplar set selection by up to 78% on Jetson Nano and 92% on a smartphone. Moreover, FastICARL decreases the storage requirement by 2-4 times without sacrificing its performance.
- We demonstrate that FastICARL can enable on-device IL without the support of the cloud, ensuring complete data privacy as user data does not need to leave the device.

- To the best of our knowledge, FastICARL is the first end-to-end and on-device framework that incorporates exemplar-based IL and quantisation techniques in the context of audio sensing applications. Note that building on our comprehensive evaluation which identified rehearsal-based methods such as iCaRL often performed best, we focus on demonstrating FastICARL's effectiveness within a specific domain while incorporating one more audio task (ESC) as FastICARL builds directly upon iCaRL.

3.2 Systematic Study of CL in Mobile Computing

In this section, we conduct our initial systematic investigation of CL in the context of mobile and embedded system domains. We first introduce the necessary components for systematic study, our CL framework (Section 3.2.1) and an experimental setup (Section 3.2.2). Then, we present our results of the comprehensive evaluation of various CL methods, datasets of multiple modalities, and diverse hardware platforms (Section 3.2.3). Based on these results, we derive practical guidelines for researchers and ML practitioners interested in applying CL in real-world applications in mobile computing (Section 3.2.4).

3.2.1 CL Framework for Mobile and Embedded Systems

We now present our framework to comprehensively evaluate the performance of various IL methods for three mobile and embedded applications (HAR, GR, and ER). We first explain the continual learning setup and three scenarios adopted in our experiments (§3.2.1). Then, we present six IL methods evaluated in this work (§3.2.1). We then describe the hyper-parameters of the LSTM based deep learning model and the different IL methods (§3.2.1). After that, we propose our novel IL model training process in §14. Next, we describe the datasets used in this study (§3.2.2). Finally, we provide brief details about our implementation (§14)

Continual Learning Setup and Three Scenarios

In this work, we focus on Sequential Learning Tasks (SLTs) from the mobile and embedded systems domain where new classes can emerge over time. Thus, the learning model has to continuously learn to accommodate new classes without CF, as would happen in real-life scenarios. Learning tasks of this type, called SLTs, indicates that a model continuously learns two or more tasks $D_1, ..., D_k$, one after another instead of learning a single task D once [21]. Figure 3.1 shows an overview of our continual learning system for sensing applications using HAR as an illustrative example. A user starts with a model containing a fixed set of classes on their devices which is then incrementally updated over time as new classes arrives.

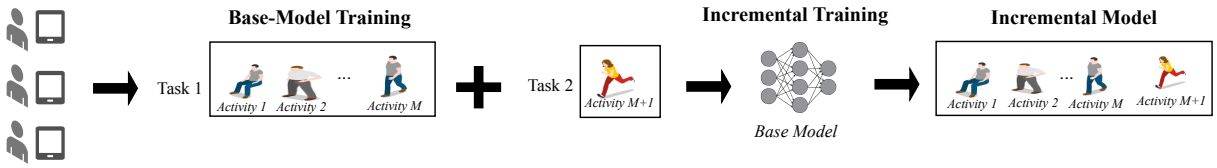


Figure 3.1: Overview of our continual learning system.

We introduce three scenarios of different levels of difficulties for models to learn continuously (from easy to difficult scenarios). First of all, inspired by Pfulb et al. [21], we adopt the SLTs consisting of two tasks: D_1 and D_2. Hence, Scenario 1 consists of two tasks, where the first task contains the $N-1$ classes, and the second task contains the other one class (N is the total number of classes). Scenario 2 includes two tasks where the first task contains half of the classes, $N/2$, and the second task contains the remainder of the classes. Finally, Scenario 3 deals with a more realistic situation where many tasks are to be learned sequentially [128]. In the third scenario, we first train a model in the first task with $N/2$ classes and then incrementally train the model by adding subsequent tasks with one class (essentially $N/2+1$ tasks). Unlike the first scenario (which has only N different cases of task permutations), it is not practical to consider every random permutation of classes to be included in different tasks for the second and third scenarios. Hence, we consider ten variations by randomly choosing classes in each task for the last two scenarios. Note that each task consists of disjoint groups of classes as we adopt class-incremental learning [224].

Incremental Learning Methods

As described in the related work section, various methods exist that can mitigate CF in IL. We describe them in depth as they form the basis of our exploration. To mitigate CF, there exist three main categories of IL approaches: (1) Regularization, (2) Replay, and (3) Replay with Exemplars. We select at least one representative method for each of the above categories. These methods are the most representative (most cited) methods for IL and are most often used in machine learning papers for comparison. We now describe the employed methods.

LSTMs [221]: LSTMs are a type of recurrent neural networks widely used for a sequence classifier in many applications, specifically for time-series data. We use LSTMs as a base neural network.

EWC [30]: Elastic Weight Consolidation (EWC) is a regularization based method which adds a penalty to regular loss function when learning a new task (i-th task), i.e.,

$$L(\theta) = L_i(\theta) + \lambda/2 \sum_{j=0}^{i-1} F_j(\Theta_i - \Theta_j^*)^2 \tag{3.1}$$

where $L(\theta)$ is the total loss, θ is the network's parameters, $L_i(\theta)$ is the loss for the new task, and Θ_j^* are the important parameters of all previous tasks. λ is a hyperparameter that controls how much importance should be given to previous tasks compared to the new task. F is the Fisher matrix used to constrain the parameters important to previously learned tasks to stay close to their old values to retain the knowledge of previous tasks and to be able to learn new tasks simultaneously.

Online EWC [108]: It is a variation of EWC method where the loss function is represented as,

$$L(\theta) = L_i(\theta) + \lambda/2(\Theta_i - \Theta_{i-1}^*)^2 \sum_{j=0}^{i-1} F_j \tag{3.2}$$

Online EWC eliminates the need to store mean and Fisher matrices for each previous task and only requires the latest mean and running sum of Fisher matrices to calculate the current task's total loss.

SI [107]: It is another regularization method which is similar to EWC where the loss function is calculated in the following way,

$$L(\theta) = L_i(\theta) + \lambda \sum_k \Omega_k^i (\theta_k^* - \theta_k)^2 \tag{3.3}$$

where k is the subscript for the parameters of the models, λ is the strength parameter, θ_k^* is the parameter value at the end of the previous task, and Ω_k^i represents the per-parameter regularization strength taking into account all previous tasks, calculated as:

$$\sum_{j=0}^{i-1} \frac{w_k^j}{(\triangle \theta_k^j)^2 + \varepsilon} \tag{3.4}$$

parameter distance $\triangle \theta_k^j$ determines how much a parameter moved between tasks during the entire trajectory of training. ε is the dampening parameter to prevent division by zero errors. The main difference between SI and EWC is that SI weights importance, w_k, is continuously updated online during training. In contrast, in EWC, the Fisher matrices (weights importance) are calculated at the end of each task.

LwF [118]: This method relies on adding loss for the replayed data to the loss of the current task. The replayed data is the input data of the current task which is labeled using the model trained on the previous tasks to generate target probabilities. The ultimate aim of the replayed data is to match the probabilities predicted by the model being trained to the target probabilities (a form of data distillation) and is termed as the loss for replayed data.

iCaRL [31]: Incremental Classifier and Representation Learning (iCaRL) store data

from previous tasks (i.e., exemplars) to alleviate the CF problem. The exemplars are a representative set of the small number of samples from a distribution, and those that can approximate the average feature vector over all training examples are selected as exemplars (based on herding [225]). The classification is done based on a nearest-class-mean (NCM) rule using features extracted from the deep learning model, where the class means are calculated from the stored examples. When new tasks (classes) arrive, iCaRL creates a new training set combining the exemplars from all the previous tasks with the data samples of the new task. Then, the model parameters are updated by minimizing a loss function which encourages the model to output the correct class for the new task (classification loss) and to reproduce the scores stored in the previous step for the old tasks (distillation loss) using data samples from the new training set.

GEM [112]: Gradient Episodic Memory (GEM) stores exemplars from the previous tasks like iCaRL and solves CF as a constrained optimization problem. A parameter update while doing IL is made depending on whether it will lead to an increase in loss for the previous tasks. This is calculated by computing the angle between loss gradient vectors of stored examples and the proposed parameter update. If the calculation suggests no loss, then the update is done straight away. Otherwise, the parameter is updated by projecting gradient in such a way that it will incur a minimal loss for the previous tasks.

Other Baselines: Joint refers to the case when training data is available for all the classes from the beginning. It is a classic case to train a model with all data at once and serves as the upper bound in many cases. **None** refers to the case when no IL method is applied to solve CF.

Our Contribution: It is worth noting that the above six IL methods are known in the machine learning literature from a theoretical point of view. Yet, they are not off-the-shelf methods that can be simply used to any dataset to enable continual learning. As will be shown in Section 3.2.3, there exist many factors affecting the performance and applicability of the IL methods in real-world deployment such as the complexity of the continual learning scenario, resource availability of mobile and embedded devices, and choice of hyper-parameters. Thus, a distinctive contribution of our work is a comprehensive evaluation and comparison study of the IL methods in diverse sensing applications and is to develop an end-to-end and on-device IL framework that can investigate trade-offs between performance, storage requirements, and latency.

Characterization of Hyper-parameters

We categorize hyper-parameters into three types and IL-method-specific parameters. First of all, we use architectural hyper-parameters which cannot be changed when learning new tasks, e.g., the number of hidden layers L and its size S. We then use learning and regularization hyper-parameters which can be adaptable when learning new tasks. For

example, a learning rate ϵ and λ term in L2-regularization can be modified during training over time. We denote the set of hyper-parameters as $\mathcal{P}$.

IL-method-specific parameters: Each IL method has method-specific parameters to control the behaviors of the model. For example, in regularization-based methods [30, 107, 118, 226], importance parameter λ is often utilized to modulate how much importance a model puts on previous tasks or a current task. The importance parameter can be adaptable while learning new tasks in our IL model training process (Algorithm 1). In addition, in replay with exemplars-based methods [31, 112], the size of the storage budget is used to balance between storage requirements and the performance of a model. Since the budget size is difficult to be adaptable after completion of the first task, it is given as an input in our experimental protocol (Algorithm 1).

Hyper-parameter setting for experiments: We first fix several hyper-parameters as default values. We set dropout rates for all tasks as 0.2 and 0.5 in input and hidden layers of a model, respectively [227] and a batch size of 32 with Adam optimizer set to a default learning rate of 0.001 for task 1 (D_1). After that, we vary hyper-parameters for all models in each dataset. Specifically, in the task 1 (D_1), we vary architectural hyper-parameters as follows: $L \subset \{1, 2\}$, $S \subset \{32, 64\}$. In subsequent tasks from task 2 to k ($D_2, ..., D_k$), we fix architectural hyper-parameters but vary adaptable hyper-parameters and IL-method-specific parameters as follows: (1) $\epsilon \subset \{0.001, 0.0001\}$ for all models, (2) $\lambda \subset \{1, 10, 10^2, 10^3, 10^4, 10^5, 10^6\}$ for both EWC and Online EWC, (3) $\gamma \subset \{0.5, 1.0\}$ for Online EWC, (4) $c \subset \{0.2, 0.4, 0.6, 0.8, 1.0\}$ for SI. We denote varying IL-method-specific parameters as $\mathcal{P}_{IL}$. For replay-based methods, the losses of the current and replayed data are weighted according to the number of tasks a model has learned so far by following [224]. We define the budget size, $\mathcal{B} \subset \{1\%, 5\%, 10\%, 20\%\}$, as a percentage of the total training samples rather than as a fixed number because the number of samples for each dataset is different.

Model Training Process

We extend protocol [21] to incorporate multiple tasks up to task k ($D_1, ..., D_k$) in an incremental manner based on our characterization of hyper-parameters and IL-method-specific parameters. Algorithm 1 describes our protocol in which we only utilize training data of a current task j ($\leq k$) for model learning and test data of previously learned tasks up to task j for evaluation.

Given an SLT consisting of $D_1, D_2, ..., D_k$ and a model m, the goal is to find a vector of hyper-parameters p^* which produces the best performance q after incrementally training all tasks up to task k. For the first step, we find the best performing hyper-parameters in task 1 (D_1) by searching among the set of architectural hyper-parameters (lines 1-5 in Algorithm 1) and update the model m_{p^*} with the found hyper-parameters (line 6).

Algorithm 1: IL model training process to determine the best model by incrementally learning tasks up to task k

Input: Tasks $D_1, ..., D_k$, model m, budget $\mathcal{B}$, epochs $\mathcal{E}$
Input: The number of hidden layers L, Hidden layer size S
Input: IL-method-specific parameters $\mathcal{P}_{IL}$, learning rate ϵ
Output: The best model with hyper-parameter vector p^*

1 **for** $p \in (L \cup S)$ **do**
2 **for** $t = 1, \mathcal{E}$ **do**
3 Train model m_1 using training set of D_1 with p
4 Test model m_1 using test set of D_1
5 Store performance $q_{1,t}$
6 Update the model m_{1,p^*} with max q_1
7 **for** $p \in (\mathcal{P}_{IL} \cup \epsilon)$ **do**
8 Initialize model m_2 with m_{1,p^*}
9 **for** $j = 2, k$ **do**
10 **for** $t = 1, \mathcal{E}$ **do**
11 Train model m_2 using training set of D_j with p
12 Test model m_2 using test set of $\cup_{l=1}^{j} D_l$
13 Store performance $q_{j,t}$
14 Update the model m_{k,p^*} with max q_k

The next step is to find the best model by searching among the set of learning hyper-parameters and IL-method-specific parameters in subsequent tasks from task 2 to k (lines 7-13). Finally, we select the best model which shows the highest performance based on test sets after incrementally trained up to task k (line 14). Note that to facilitate the extensive experiments performed in our study and to make a fair comparison among the IL methods (Section 3.2.3), we first identify the best architectural hyper-parameter (from $L \subset \{1, 2\}$ and $S \subset \{32, 64\}$) and then use the found hyper-parameter across the different IL methods. The final LSTM architecture we used for each dataset is reported in Table 3.1.

Implementation

We implemented our continual learning framework on Nvidia Jetson Nano and OnePlus 7 Pro smartphone (with Qualcomm SM8150 Snapdragon 855) platforms. All the IL algorithms were explored on Nano GPU, and we used PyTorch 1.1 to implement the framework. Keeping in mind that Scenario 3 is the most practical continual learning scenario and iCaRL is the best performing IL approach, we only implemented iCaRL for Scenario 3 on the smartphone's CPU (as an Android app) using the DeepLearning4j

Table 3.1: Overview of the employed datasets.

Application	Dataset	Dimension	# Train Data	# Test Data	# Classes	*Layer/Size*
HAR	HHAR	20 × 120	59,403	7,721	6	2/64
	PAMAP2	33 × 52	35,263	5,209	12	1/64
	Skoda	33 × 60	10,047	1,193	10	1/64
GR	Ninapro (Per Subject)	40 × 12	3,118	639	10	1/64
	Ninapro (LOUO)	40 × 12	30,488	3,759	10	1/64
ER	EmotionSense	20 × 24	2,011	224	14	2/64

library. The smartphone app size is 134 MB. We choose CPU on the smartphone as it provides an upper bound on the performance of any system and is more challenging to implement. We envisage that if a system can work (or at least feasible) on a CPU, then it would be much easier and faster to run similar systems on accelerators such as GPU. When working on a dataset, we first loaded the training data pertaining to all the tasks in the memory to make the continual learning process work faster. As a limited amount of memory is allocated to each Android app, we set large heap property in the app to True to use larger heaps for our app. We still encountered memory issues, especially when working with large datasets such as Skoda, which we solved by using memory-mapped files.

In addition, we employ a weighted F1-score which is more resilient to class imbalances as the employed datasets (see §3.2.2 for details) are not balanced [33, 228]. As in [229], we applied a weighted loss to all evaluated methods by estimating the inverse class distribution which gives more importance to the loss of a class with fewer samples. Also, as deep learning models can overfit to small datasets such as EmotionSense, with our framework, we experimented with shallow and deep neural network architectures and found that deep architectures show marginal improvement over shallow architectures, indicating that the overfitting is not an issue.

3.2.2 Experimental Setup

Before we present the findings of this work in Section 3.2.3, we describe an experimental setup for conducting a comprehensive evaluation of three continual learning schemes in mobile and embedded sensing applications. We first describe six datasets in three different sensing applications (§3.2.2) and evaluation metrics adopted for systematic comparison of the IL techniques and their trade-offs between system aspects (e.g., storage and computational costs) (§3.2.2).

Datasets

We focus on three sensing applications (e.g., HAR, GR, and ER) as they are some of the most popular applications in mobile sensing. Table 3.1 shows the overview of the employed

datasets.

Human Activity Recognition (HAR): For the HAR application, we used three datasets: (1) HHAR [228], (2) PAMAP2 [230], and (3) Skoda [231]. These datasets contain many real-life activities (e.g., walking, sitting, and cycling) obtained using Inertial Motion Units (IMUs), which contain accelerometer, gyroscope, and magnetometer data of mobile and wearable devices. We next present the detailed summaries of the three datasets.

1. **HHAR:** This dataset considers six different daily activities of users The data was recorded from nine participants, where they followed a scripted set of activities with eight smartphones and four smartwatches of different brands and models. Having various devices for recording makes HHAR an excellent benchmark to study the heterogeneity of HAR (i.e., sensor biases, sampling rate heterogeneity, and sampling rate instability). We follow the preprocessing steps as proposed by Yao et al. [232]. Raw measurements of both accelerometer and gyroscope are segmented into 5-second samples. Each sample is divided into time intervals of 0.25s. After that, we apply a Fourier transform to each time interval. It produces $d \times 2f$ dimensional vectors per time interval, where d is the dimension for each measurement and f is the frequency with magnitude and phase pairs, resulting in 120 dimensions. We adopt leave-one-user-out (LOUO) for evaluation [232]. One user (i.e., the first participant) is used for testing, and the remaining users are left for training.

2. **PAMAP2:** In this dataset, nine subjects carried out various daily living activities and sportive exercises. IMU data (accelerometer, gyroscope, magnetometer), heart rate, and temperature data were recorded from body-worn sensors attached to the hand, chest, and ankle. The resulting dataset has 52 dimensions, and more than 10 hours of data were collected. We follow a preprocessing protocol used by Hammerla et al. [33]. The sensor data are downsampled to 33Hz. After that, all samples are normalized to zero mean and unit variance. Also, to be consistent with the previous works [8, 33, 233], we use runs 1 and 2 from the sixth participant for testing and the remaining data for training.

3. **Skoda:** This dataset contains the activities of assembly-line workers in a manufacturing scenario. One subject wore 20 3D accelerometers on both arms. Following the preprocessing steps [8, 84], we employ raw and calibrated data from ten accelerometers placed on the right arm, resulting in input data of 60 dimensions. The data are downsampled to 33Hz and normalized to zero mean and unit variance. For experiments, the last 10% of each class is used as the test data and the remaining as the training data. Note that Skoda consists of one subject, i.e., subject dependent evaluation.

Gesture Recognition (GR): We employ the Non Invasive Adaptive Prosthetics (Ninapro) database [234] for the GR application in our experiments as it consists of surface

electromyography (sEMG) signals and thus can provide different sensor modalities than IMU sensors present in HAR datasets.

1. **Ninapro (Per Subject):** This dataset is widely used in research on hand movement recognition applications. We employ Ninapro Database2 (DB2) in this study. It includes sEMG data recordings from 40 subjects while performing several repetitive gestures such as wrist movements, grasping and functional movements, and force patterns. Following, Li et al. [235], we select ten types of hand gestures commonly used in daily life. After that, we downsample the sEMG data to 200 Hz and normalize them to zero mean and unit variance. We used a sliding window size of 200 ms with a 50% overlap [47, 114]. We select a subject who has the most amount of data samples for subject dependent (i.e., per subject) evaluation. After that, we use the fifth repetition for a test set and the remainder for training.
2. **Ninapro (LOUO):** To have a consistent evaluation with the HAR application, we adopt the LOUO evaluation for the GR application using the Ninapro dataset. We select the top ten subjects having more data samples than others. After that, we use a subject with the least data samples for testing and the remainder for training. The preprocessing steps are the same as in Ninapro (Per Subject).

Audio Sensing Task: We select Emotion Recognition (ER) since it is one of the most widely adopted audio sensing tasks. We employ the EmotionSense dataset [15] which was collected by recording human participants' emotions as well as proximity and patterns of conversation using an off-the-shelf smartphone. This dataset has been used in multiple studies to understand the correlation and impact of interactions and activities on the emotions and behaviour of individuals in various settings [236][69][12].

1. **EmotionSense:** This dataset contains audio signals which represent 14 different emotions. In the EmotionSense dataset, each measurement corresponding to a particular emotion (or class) is based on a 5-second context window. Following Georgiev et al. [101], we extract 24 log filter banks [237] from each audio frame over a time window of 30 ms with 10 ms stride. Each sample contains 500*24 = 12,000 features where 1–24 features are filter banks from the first 10 ms, and 25–48 features are filter banks for the next 10 ms and so on. After that, as our preprocessing steps, we downsample each sample measurement by averaging corresponding 24 filter banks of every 250 ms (or 25 consecutive windows) without any overlap to reduce the length of the input sequence for a learned neural network. We normalize each window to zero mean and unit variance.

Evaluation Metrics

We consider how much an IL method forgets previous tasks and learns new tasks after it was trained from task 1 to k to assess the actual performance of IL methods [110] by

considering the following metrics.

Average Performance Measure (A): We denote the performance measure of a model on the j-th task ($j \leq k$) as $a_{k,j} \in [0, 1]$ after the model is trained from task 1 to k. The average performance measure at task k is defined as follows:

$$A_k = \frac{1}{k} \sum_{j=1}^{k} a_{k,j} \tag{3.5}$$

The output space consists of $\cup_{j=1}^{k} \boldsymbol{y}^j$, and $a_{k,j}$ is based on a weighted F1-score in this work. Note that $a_{k,j}$ can be used to indicate an accuracy, proportion of correctly classified activities or gestures.

Forgetting Measure (F): The forgetting measure provides an estimate of how much a model forgets about the task given its present state. The forgetting for the j-th task after the model has been trained up to task $k > j$ can be quantified as:

$$f_j^k = \max_{l \in 1,\ldots,k-1} a_{l,j} - a_{k,j}, \quad \forall j < k \tag{3.6}$$

The average forgetting at k-th task is denoted as $F_k = \frac{1}{k-1} \sum_{j=1}^{k-1} f_j^k$ by normalizing the number of tasks seen previously. The lower the F_k, the less forgetting on previous tasks.

Intransigence Measure (I): Intransigence is defined as the inability of a model to learn new tasks. To quantify the inability to learn, the joint model, often considered upper bound, which has access to all the datasets seen so far ($\cup_{l=1}^{k} D_l$) is compared and its performance is denoted as a_k^*. We then denote the intransigence for the k-th task as:

$$I_k = a_k^* - a_{k,k} \tag{3.7}$$

where $a_{k,k}$ represents the performance of a model on the k-th task trained up to task k. Lower I_k implies that a model performs as close as a joint model or performs even better than the joint model when intransigence is negative ($I_k < 0$). Note that we use $a_{k,k}$ and I_k as the main performance indicators of a model since we are interested in the current performance of the model on all learned tasks from 1 to k.

Note that in addition to the metrics mentioned above, we also report **storage** and **latency** required to execute each IL method.

3.2.3 Findings

We now present the results of our evaluation. Firstly, we compare the performances of different IL methods on HAR, GR, and ER tasks using two basic scenarios (Scenario 1

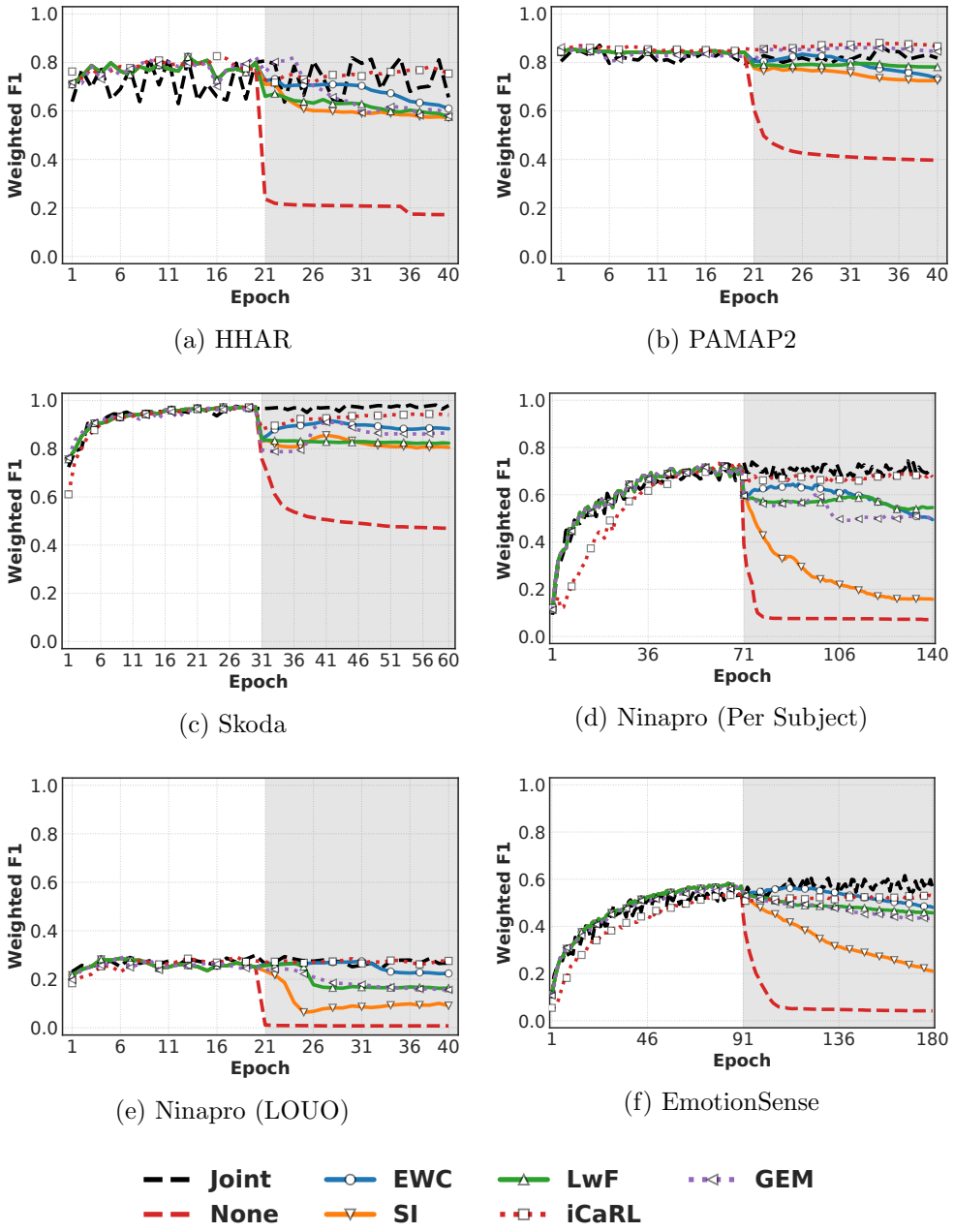
and 2) in §3.2.3. Then, we study the performance of IL methods for Scenario 3 in §3.2.3. We examine the generalizability of IL methods across different datasets (§3.2.3). Then, we discuss the trade-offs of IL methods with respect to the storage, computational costs, and memory footprint. (§3.2.3). Finally, in §3.2.3, we investigate the effect of iCaRL specific parameters on the performance.

Performance on Simple and Mildly Difficult Tasks

We show the best average weighted F1-scores across all runs for different IL methods for Scenario 1 and 2 for different datasets in Figure 3.2 and Figure 3.3, respectively. The white part in the figure shows performance on Task 1, and the grey part shows performance for Task 2. For HAR and GR applications, the results of iCaRL/GEM with the budget size of 20% are shown in Figure 3.2 and Figure 3.3 since the models with the budget size of 20% show the best performance. Then, for the ER application, the results of iCaRL/GEM with the budget size of 40% are shown since the EmotionSense dataset has the least number of training samples, requiring more budget size in ER than the other two applications.

The results show that without any IL method (None), the performance drops sharply as soon as a new task is encountered. The decline in performance is as drastic as 60% in both scenarios. iCaRL provides the best performance in Scenario 1, which stays very close to the performance obtained with the joint model. It is because iCaRL stores representative exemplars and relies on a nearest-class-mean (NCM) rule that is robust against changes in the data representation [31]. In fact, all the IL methods effectively solve the CF problem and achieve comparable performance to the joint model (between 5% and 15%) after only running for a few epochs (5 or less in many cases). *One can conclude that, in general, the existing IL methods we analyzed can solve the CF issue on mobile and embedded sensing applications for simple scenarios.*

However, the same cannot be said for the performance in Scenario 2. *Except iCaRL, none of the other methods seems to solve the CF issue for the mildly complex scenario (i.e., Scenario 2).* The performance drop is up to 60% when the performances between IL methods and the joint model are compared. iCaRL remains the best performing method with its weighted F1-score close to that of the joint model (within 10%). GEM performs the second-best (within few epochs) on HAR datasets while EWC performs well for GR and ER datasets. Although GEM is a replay with exemplars-based approach like iCaRL, it never matches the performance of iCaRL due to its reliance on using gradients and not the actual examples themselves. Another reason might be that iCaRL selects best examples to be stored based on herding (a sort of prioritization), while GEM employs selecting examples randomly which can be less informative. A regularization-based method such as SI and a replay only approach such as LwF perform poorly across all datasets. The weighted F1-score degrades roughly 40–50% of what can be achieved by the joint model. As indicated by [238], the performance of LwF significantly decreases when the



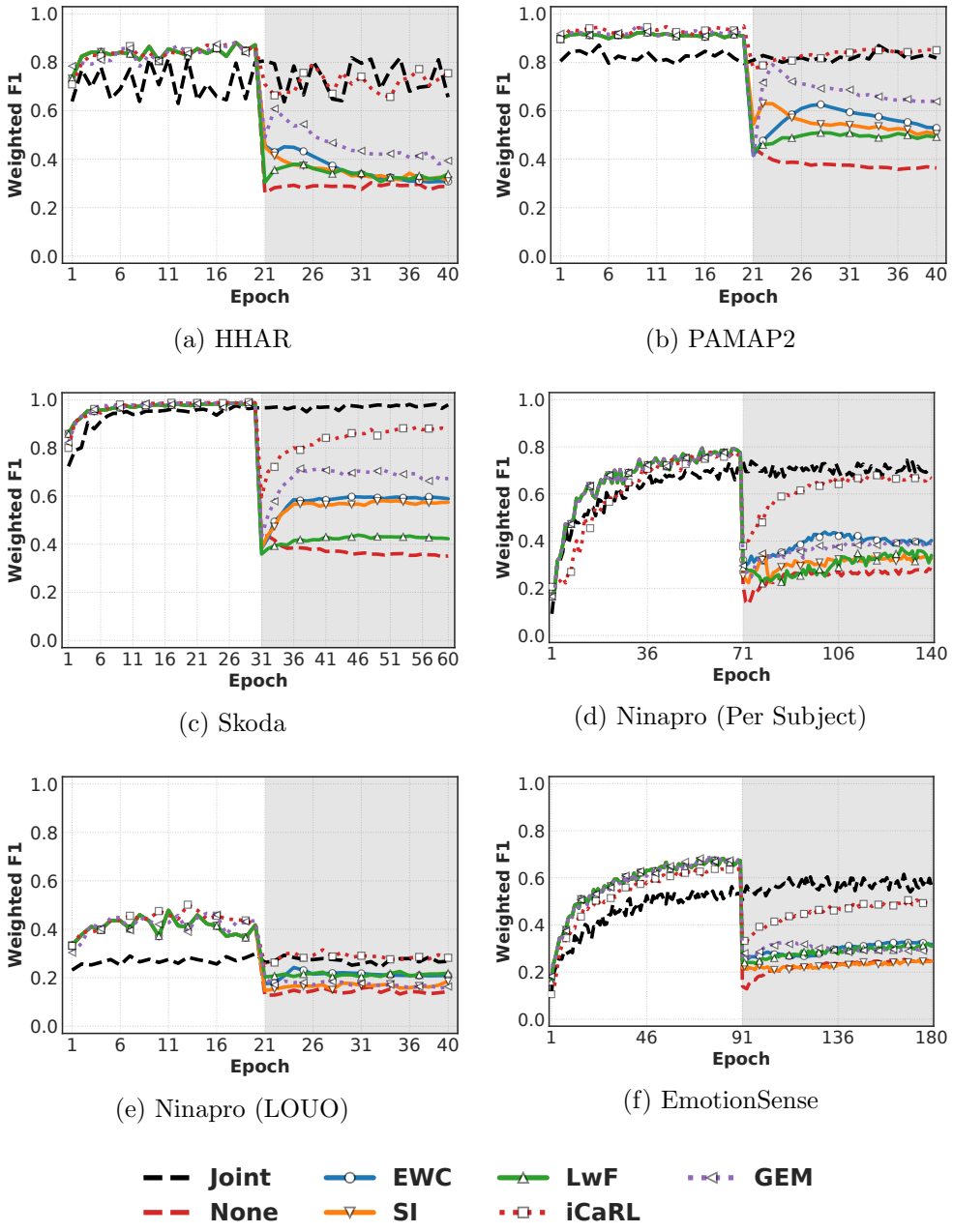
(a) HHAR (b) PAMAP2

(c) Skoda (d) Ninapro (Per Subject)

(e) Ninapro (LOUO) (f) EmotionSense

Figure 3.2: The performance comparison of the five IL methods including two baselines in Scenario 1 on each dataset.

model learns a sequence of tasks drawn from different distributions. In other words, when tasks learned by LwF are not sufficiently related, enforcing the new model to give similar outputs for the old task may hurt the model's performance. SI relies on the weight changes



(a) HHAR (b) PAMAP2

(c) Skoda (d) Ninapro (Per Subject)

(e) Ninapro (LOUO) (f) EmotionSense

Figure 3.3: Performance comparison in Scenario 2.

in a batch gradient descent which can overestimate the importance of the weights and thereby leads to lower performance.

Note that iCaRL employs a different way (i.e., NCM rule) to classify data samples (perform

inference) than other methods (including None and Joint) which use cross-entropy based classification. Also, for GEM, it minimizes the loss on the current task by using inequality constraints, avoiding its increase but allowing its decrease. Therefore, iCaRL and GEM can obtain different weighted F1-scores than the other methods in task 1. Otherwise, ideally one would assume all methods (e.g., None, EWC, SI, LwF in our study) to get the same performance in the first task as it only involves learning a baseline LSTM model without any IL. Also worth mentioning is that initially (especially task 1) IL methods can achieve higher weighted F1-scores than the joint model. It is because their performance is based on classifying the smaller number of classes than the joint model, where all classes need to be classified from the first epoch.

Performance on Many Sequential Tasks

Figure 3.4 shows results for Scenario 3. Recall that Scenario 3 presents the case when classes are added one by one to an already existing deep learning model, which will happen in real-life scenarios and is the most challenging task for any IL method. Note that this graph is shown differently than the graphs for Scenario 1 and 2 (epoch based) as in epoch based graph, we would have only two data points to show as there were only two tasks. In Scenario 3 the number of tasks will be $N/2 + 1$ for N classes. Without the IL method (None), CF happens, and the weighted F1-score almost always lies between 0%–10%. *iCaRL is the best method and appears to solve the CF issue for the challenging third scenario.* Its performance is nearly equal to the joint model in most of the cases. *All other methods do not solve the CF issue, and the performance suffers severely as more tasks are added to the system especially with LwF and SI.*

Generalization

Table 3.2 shows the results in a summarized way for all the datasets and IL methods evaluated in our study. A_k refers to average performance on all tasks while $a_{k,k}$ shows the weighted F1-score at the end of learning all tasks. F_k tells us how good an IL method is in retaining old knowledge about previous tasks. Whereas I_k means how much an IL method is good at learning new tasks. Note that the higher the values of A_k and $a_{k,k}$, the better the model is. However, for F_k and I_k, a low value indicates a better model since low F_k and I_k means that the model forgets knowledge of previous tasks less and performs as close as a joint model, respectively. iCaRL is one of the best-performing methods on all metrics across all datasets. iCaRL can learn new classes (tasks) while retaining old knowledge and maintain high performance even in the most challenging scenario. Given that small errors are allowed when performing HAR, GR and ER, iCaRL alleviates the issue of CF to a large extent. The same is not true for all other IL methods. Although LwF allows previous knowledge to be largely retained (low F value), it does not learn new tasks easily and thus has low performance in general. SI is neither good at learning

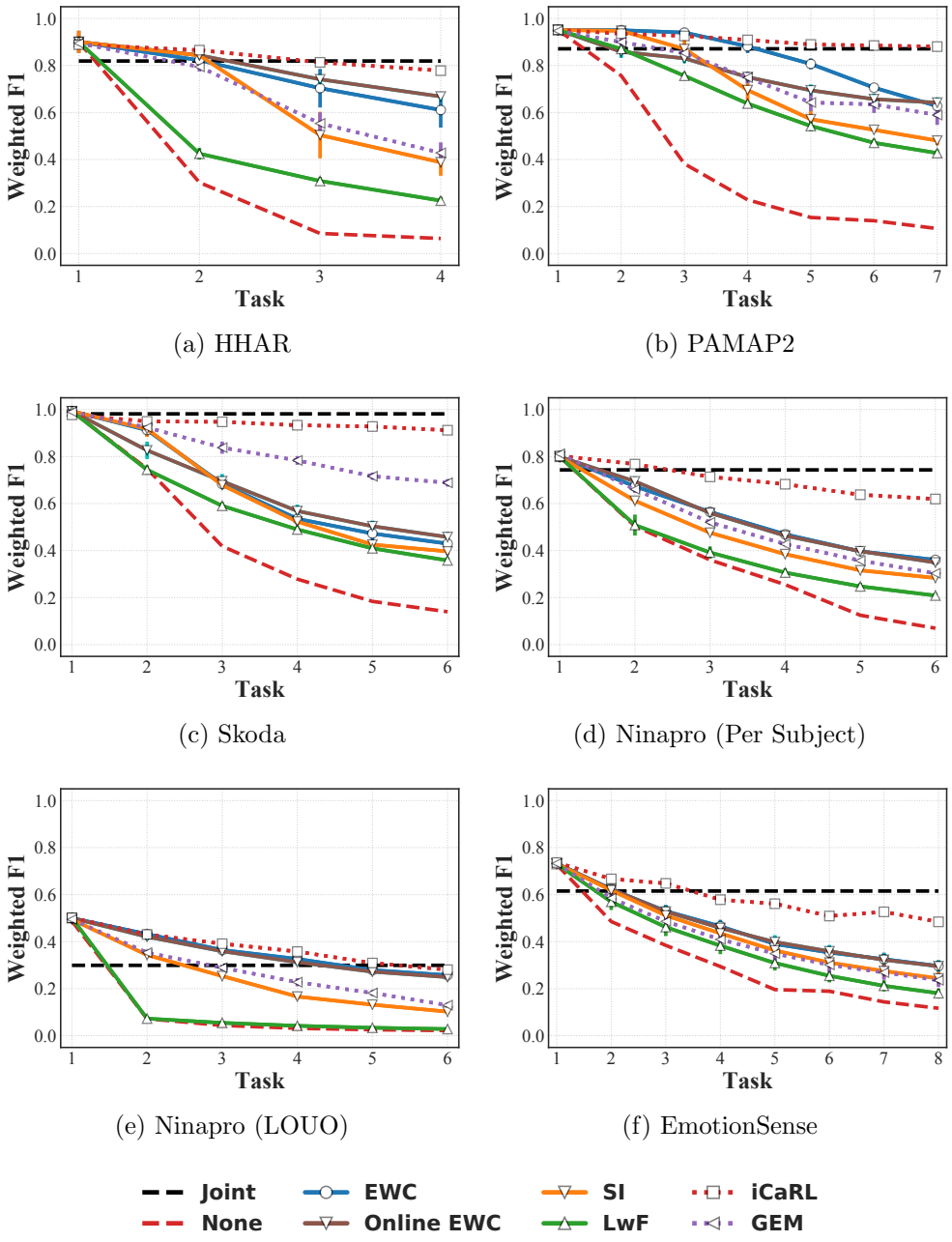


Figure 3.4: The performance comparison in Scenario 3. All reported results are averaged over 10 trials, and standard-error intervals are depicted.

new tasks (high I) nor at remembering old knowledge (high F). EWC and online EWC offer a decent alternative to iCaRL without needing extra storage on-device but at the

Table 3.2: Average performance of different methods in all scenarios on HAR, GR, and ER.

Scenario	Methods	HAR				GR				ER			
		A_k	F_k	$a_{k,k}$	I_k	A_k	F_k	$a_{k,k}$	I_k	A_k	F_k	$a_{k,k}$	I_k
1	None	0.55	0.35	0.54	0.36	0.22	0.29	0.20	0.32	0.47	0.11	0.44	0.16
	EWC	0.88	0.01	0.86	0.03	0.49	0.01	0.47	0.05	0.60	0.01	0.58	0.03
	SI	0.85	0.03	0.81	0.07	0.45	0.04	0.42	0.10	0.57	0.01	0.54	0.07
	LwF	0.84	0.02	0.79	0.10	0.47	0.02	0.44	0.09	0.57	0.01	0.54	0.07
	iCaRL	0.89	0.01	0.88	0.01	0.51	0.01	0.49	0.03	0.57	0.02	0.56	0.05
	GEM	0.88	0.01	0.87	0.02	0.46	0.03	0.43	0.09	0.57	0.01	0.54	0.06
2	None	0.30	0.76	0.41	0.48	0.20	0.48	0.23	0.29	0.27	0.45	0.27	0.34
	EWC	0.77	0.06	0.65	0.24	0.47	0.06	0.35	0.17	0.55	0.01	0.39	0.22
	SI	0.64	0.26	0.60	0.29	0.31	0.31	0.29	0.23	0.38	0.27	0.32	0.29
	LwF	0.70	0.03	0.48	0.41	0.45	0.06	0.31	0.21	0.52	0.04	0.35	0.26
	iCaRL	0.89	0.05	0.86	0.03	0.53	0.09	0.51	0.02	0.57	0.07	0.53	0.08
	GEM	0.77	0.13	0.71	0.18	0.39	0.19	0.31	0.21	0.51	0.08	0.37	0.24
3	None	0.22	0.21	0.10	0.79	0.09	0.17	0.05	0.48	0.18	0.16	0.12	0.49
	EWC	0.75	0.01	0.56	0.34	0.44	0.01	0.31	0.21	0.46	0.01	0.30	0.49
	Online EWC	0.72	0.03	0.59	0.30	0.44	0.01	0.30	0.22	0.45	0.01	0.30	0.31
	SI	0.59	0.10	0.42	0.47	0.32	0.07	0.22	0.31	0.42	0.02	0.24	0.36
	LwF	0.53	0.06	0.34	0.55	0.20	0.08	0.12	0.40	0.29	0.11	0.18	0.43
	iCaRL	0.86	0.01	0.79	0.10	0.53	0.01	0.45	0.07	0.62	0.12	0.48	0.13
	GEM	0.70	0.07	0.57	0.32	0.33	0.08	0.22	0.31	0.33	0.02	0.16	0.44
-	Joint	-	-	0.89	-	-	-	0.52	-	-	-	0.61	-

expense of lower performance than iCaRL. The overall takeaway is that *iCaRL can enable a system to learn incrementally (continuously) in the mobile and embedded sensing domain (if storage is not such a constraint on a device).*

Storage, Latency, and Memory Footprint

Storage: We report the storage overhead of each IL method, as shown in Table 3.3. We first specify the mathematical formulas used to calculate the overall storage requirements of each IL method to show how much storage the IL method needs with respect to the number of tasks ($\mathcal{T}$) added, the model parameters (M), and the budget size ($\mathcal{B}$). This point would help practitioners and researchers easily understand how much storage overhead occurs when they want to deploy their models with a particular IL method. First of all, LwF requires no extra storage other than the storage needed to store the model parameters (M). Then, SI requires a running estimate (w_k), the cumulative importance measures (Ω_k^i), and reference weights (θ_k^*) of importance weights of the current task. EWC stores Fisher matrices and means for each task. Unlike EWC, Online EWC is only required to

Table 3.3: Storage requirements of IL methods. $\mathcal{M}$ refers to the number of model parameters, $\mathcal{T}$ represents number of tasks and $\mathcal{B}$ is the storage budget.

Category	Method	Required Storage
Reg-based	EWC	$2 \times \mathcal{M} \times \mathcal{T}$
	Online EWC	$2 \times \mathcal{M}$
	SI	$3 \times \mathcal{M}$
Replay-based	LwF	$\mathcal{M}$
Replay+Exemplars	iCaRL	$\mathcal{M} + \mathcal{B}$
	GEM	$\mathcal{T} \times \mathcal{M} + \mathcal{B}$

Table 3.4: Storage requirements of IL methods for all datasets - Scenario 3. Units are measured in MB.

IL Method	HHAR	PAMAP2	Skoda	Ninapro (Per Subject)	Ninapro (LOUO)	Emotion Sense
EWC	2.601	3.599	3.177	2.587	2.587	3.663
Online EWC	0.650	0.514	0.529	0.431	0.431	0.458
SI	0.975	0.771	0.794	0.647	0.647	0.687
LwF	0.325	0.257	0.265	0.216	0.216	0.229
iCaRL (1%)	5.990	2.676	1.051	0.270	0.805	0.257
iCaRL (5%)	28.838	12.341	4.187	0.512	3.190	0.407
iCaRL (10%)	57.350	24.421	8.179	0.805	6.179	0.607
iCaRL (20%)	114.374	48.658	16.162	1.410	12.141	0.981
iCaRL (40%)	-	-	-	-	-	1.755
GEM (1%)	6.989	4.205	2.350	1.351	1.884	1.862
GEM (5%)	29.817	13.874	5.537	1.583	4.278	2.016
GEM (10%)	58.372	25.996	9.532	1.884	7.274	2.217
GEM (20%)	115.444	50.240	17.476	2.485	13.266	2.603
GEM (40%)	-	-	-	-	-	3.374

store one Fisher matrix and running means across tasks. Thus, the required storage for Online EWC does not increase as the number of learned tasks increases. Similar to LwF, iCaRL also requires the previous task model for knowledge distillation. For GEM, it stores the gradient of the exemplar set for each learned task. As both iCaRL and GEM rely on stored examples, their storage demands are mainly driven by the number of examples to be stored (i.e., budget size, $\mathcal{B}$).

Numerical model sizes (i.e., $\mathcal{M} + \mathcal{B}$) are shown in Table 3.4 for all the employed datasets

in Scenario 3. Note that we do not add tables containing the results of Scenario1 and 2 due to the page limit. However, by reporting the results of Scenario 3 where the storage requirements of various IL methods are greater than or equal to those of Scenario 1 and 2, we aim to present the upper bound of the required storage. Besides, the reported numerical sizes of storage requirements in Table 3.4 are based on IL methods with the largest model in our experiments (i.e., number of LSTM layers ($L = 2$) and the number of hidden units ($S = 64$)) to capture the upper bound to practically operate IL methods on embedded and mobile devices. Here we take the Skoda dataset to further explain our findings as it represents an ideal use case scenario where IL methods need to be applied to personal mobile devices (single-user scenario with modest dataset size). In the Skoda dataset, replay with exemplars methods such as iCaRL and GEM requires at most around 17 MB, and other IL methods have even smaller storage requirements. For EmotionSense dataset where we use up to 40% budget, iCaRL needs less than 2 MB, and GEM needs less than 3.4 MB at most. Even with the largest dataset of HHAR in our experiments, the storage requirements are constrained within less than about 115 MB, which falls well within the storage capacity of modern embedded devices and smartphones. Many modern mobile and embedded devices already support a large amount of storage (in order of GBs).

In summary, the amount of storage required to practically enable continual learning on many modern edge platforms such as Nvidia Jetson or Raspberry PIs and smartphones is not excessive, as evident from Table 3.4. Note that tuning appropriate parameters in the IL method would still allow IL to perform effectively, i.e., ensuring good performance with a reasonable budget size (discussed in §3.2.3).

Latency: The average training and incremental learning time to execute different IL methods are illustrated in Table 3.5 for all the employed datasets in Scenario 3 on Jetson Nano[2] which is an edge platform having four cores, 4 GB RAM and a GPU and often used in mobile robotics and can be used in tablets. Training time represents the usual training time involved in learning a neural network including updating weights, back-propagation, etc. GEM is computationally the most expensive. On small datasets of Ninapro (Per Subject) and EmotionSense, IL time is around 57.3-85.2 seconds. Then, on the largest dataset of HHAR, IL takes up to 2,660 seconds. It is because gradient computation over previous tasks is computationally expensive. Also, EWC and Online EWC show high IL time, taking over 1,213 seconds in HHAR. This is surprising as EWC is a simple method. However, the time complexity comes from calculating and updating the Fisher matrices, which is a computationally expensive process, after every task. SI (mostly relying on running estimates) and LwF (replay only, calculating distillation loss) are two of the top three fastest IL methods but come at the peril of very low accuracy, making them unsuitable for IL in mobile and embedded applications. iCaRL, the best performing IL

[2]By reporting the results of Scenario 3 where the latency of IL methods is greater than or equal to that of Scenarios 1 and 2, we aim to capture the upper bound of the latency.

Table 3.5: Average Latency (Training Time/IL Time) in seconds for IL methods on different datasets - Scenario 3 on Jetson Nano.

IL Method	HHAR	PAMAP2	Skoda	Ninapro (Per Subject)	Ninapro (LOUO)	Emotion Sense
EWC	672/1213	329/599	120/170	173/73.1	251/558	159/67.8
Online-EWC	651/1188	291/570	105/162	148/60.2	225/539	131/46.3
SI	717/144	336/55.3	118/18.0	146/22.1	269/47.1	123/22.4
LwF	660/88.6	362/70.0	113/15.7	150/19.4	284/58.5	128/14.2
iCaRL (1%)	906/76.2	268/36.3	113/13.6	141/12.9	265/32.4	117/8.46
iCaRL (5%)	928/93.0	269/44.2	131/16.6	147/15.1	244/38.3	118/8.80
iCaRL (10%)	896/109	302/54.7	149/19.3	130/13.2	235/43.6	119/10.5
iCaRL (20%)	924/150	299/71.7	123/19.1	149/16.5	228/57.1	130/11.0
iCaRL (40%)	-	-	-	-	-	111/11.9
GEM (1%)	607/385	262/275	86.6/53.4	117/57.3	196/170	102/70.2
GEM (5%)	1085/1012	289/377	92.3/65.7	119/61.9	219/224	105/81.6
GEM (10%)	1529/1521	379/624	94.2/70.1	122/71.6	295/380	104/76.6
GEM (20%)	2641/2660	576/1247	132/142	124/85.2	454/656	102/72.2
GEM (40%)	-	-	-	-	-	106/83.5

method, is also very fast and takes only a few seconds (e.g., 8.46–16.5 seconds) in the Ninapro (Per Subject) and EmotionSense datasets to complete. In the HHAR dataset, the average latency of IL time of iCaRL with the largest budget size (i.e., 20%) is relatively small of 150 seconds compared to its training time (i.e., 924 seconds) and the IL time of EWC (i.e., 1,213 seconds) and GEM (i.e., 2,660 seconds). In reality, most of the time is taken by actual training (except EWC and Online-EWC), which depends on the number of epochs to be performed and is independent of the IL method. Across scenarios, we observe that the average training time can range from one to 15 minutes in general (except GEM).

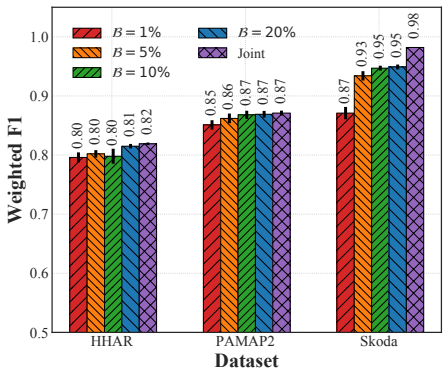
Having realized that iCaRL is the most promising method in terms of accuracy and latency, we wanted to check if iCaRL can also effectively work on modern smartphone CPUs. For this, we have implemented iCaRL on OnePlus 7 Pro for three datasets: Skoda, Ninapro (Per Subject), and EmotionSense as they represent datasets where IL needs to be applied to personal mobile devices (single-user case) and Scenario 3 (most practical scenario). The smartphone has eight cores and 12 GB of RAM. To reiterate, we used DeepLearning4j library to implement iCaRL. The smartphone app size is 134 MB. The results are shown in Table 3.6. Similar to Jetson Nano, iCaRL takes minimal time (0.5–212 seconds) for all the tasks for every dataset. This does not only mean that IL is feasible on modern smartphones but even if a very high number of tasks are to be learned even in the

Table 3.6: Average Latency (Training Time/IL Time) in seconds for iCaRL on three datasets - Scenario 3 on Smartphone.

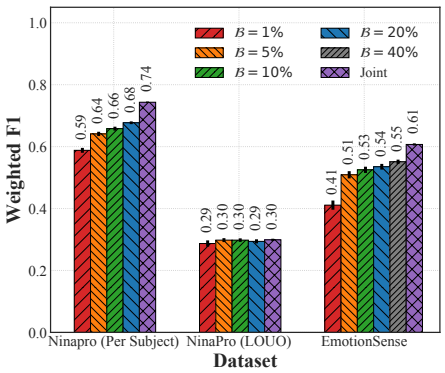
IL Method	**Skoda**	**Ninapro (Per Subject)**	**EmotionSense**
iCaRL (1%)	4400/9	1956/1.28	1568/0.5
iCaRL (5%)	3894/29	1974/3	1388/1.91
iCaRL (10%)	3869/72	2312/4.5	1535/2.6
iCaRL (20%)	3902/212	2008/5.1	1517/4.7
iCaRL (40%)	-	-	1506/8.1

most challenging scenario, iCaRL can do end-to-end IL in a few minutes. The training time slows down the whole process and ranges from 20–75 minutes on the CPU of the smartphone for different datasets. Also note that the training time taken by the tasks after the first task (actual incremental tasks after the initial model is trained) is very small: one to four minutes. This is a relevant result as one can train a baseline model on a powerful machine first and can then move it to a mobile and embedded device to learn incrementally over time. Regardless, we show that the complete incremental learning process can still be done entirely on the smartphone CPU, especially given that the phone can be charged overnight. *This is an interesting result as this suggests that our continual learning framework can be deployed on a smartphone CPU. It is also encouraging because the performance can be further improved by exploiting GPU and NPU once support for training them programmatically starts to emerge.*

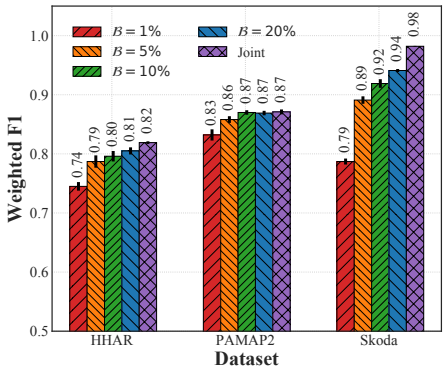
Memory footprint: We further examine the peak memory usage of iCaRL with its largest budget size of 20-40% on all the datasets to evaluate whether or not it can fit the tight memory budget of Jetson Nano. The peak memory overheads of running the end-to-end IL range from 196 MB for our smallest dataset of EmotionSense to 1,194 MB for our largest dataset of HHAR, when the CPU is used for IL. Then, when we use GPU for running iCaRL, it incurs 1,782-2,127 MB peak memory and requires an additional swap space of 750-3,523 MB. Note that we report the upper bound of the peak memory usage to understand the memory resource requirements of IL methods. Also, the memory overheads can be mitigated by using a smaller batch size and budget size that can fit into resource availability of a target resource-constrained device. Furthermore, we observed that the latency reduction using GPU over CPU is largely consistent between 80-86%, indicating that the swap space has minimal impacts on the speed-up of the IL using GPU compared to using CPU on Jetson Nano. *This result confirms that IL in the mobile and embedded sensing domain is applicable on resource-constrained devices within a reasonable memory overhead.*



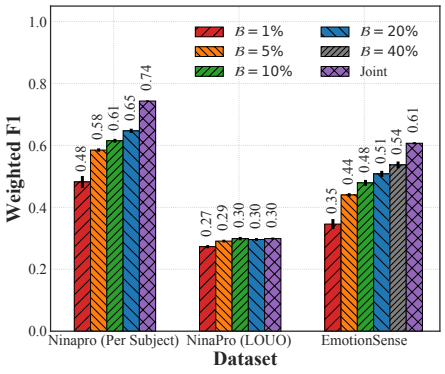
(a) Scenario 1



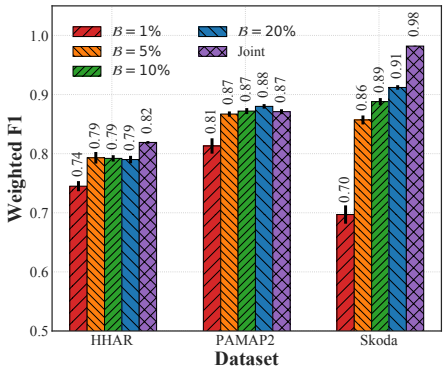
(b) Scenario 1 (GR & ER)



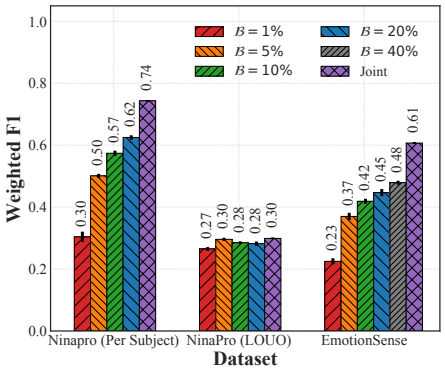
(c) Scenario 2



(d) Scenario 2 (GR & ER)



(e) Scenario 3



(f) Scenario 3 (GR & ER)

Figure 3.5: The parameter analysis of the best performing model, iCaRL, in all tasks (HAR, GR, and ER) for all scenarios according to its storage budgets. Reported results are averaged over 10 trials. Standard-error intervals are depicted.

Performance with IL parameters

We study the importance of the storage budget parameter for iCaRL as it is the best performing IL method. Figure 3.5 shows the weighted F1-score with changing storage budgets of 1%, 5%, 10%, and 20% of total training samples (up to 40% storage budget for the case of ER). In general, more samples are needed to avoid CF as the complexity of the scenario increases. In Scenario 1, only 1% of total samples are needed to achieve similar performance as the joint model. Moreover, in Scenario 2 and 3, the results show that the budget size of 5% is enough to achieve the high performance which is quite close to that of the joint model, although the difficulty of the task increases compared to Scenario 1. In contrast, 10% of samples are required to achieve near joint model's performance (i.e., upper bound performance) in the most challenging setup (Scenario 3).

Note that the performances of iCaRL with the budget size of 5% are often very close to those of iCaRL with budget sizes of 10%, 20%, and 40%. This result indicates that iCaRL enables us to achieve close to the performance of the joint model without requiring excessive storage (less than 30 MB in all datasets in our experiment when a budget size is 5%). Specifically, the required storage of iCaRL with 5% budget size for each dataset (HHAR, PAMAP2, Skoda, Ninapro (Per Subject), Ninapro (LOUO), and EmotionSense corresponds to 28.84, 12.34, 4.19, 0.51, 3.19, and 0.41 MB, respectively. *This is an interesting finding, making iCaRL a good candidate to perform IL on many embedded devices and smartphones with reasonable storage as only a few samples are required to be stored.*

3.2.4 Discussion

We discuss the potential guidelines ($\mathcal{G}$) for researchers and practitioners in the mobile and embedded systems community based on our findings of this work. The readers should take our results and guidelines with a pinch of salt as we did not compare all the existing IL methods due to reasons mentioned earlier (Section 3.2.1) and these findings are based on a few prominent IL methods we analyzed in our study.

($\mathcal{G}_1$): If storage is not an issue on the device, one can choose to use the iCaRL method since it performs best across all datasets in different sensing applications. As many modern computing platforms including smartphones and embedded devices have large storage capacity, the issue of storing a proportion of training samples can be minor. iCaRL is also not very computationally expensive on the modern embedded devices and the smartphone. Also, the process can be sped up by using GPUs, although it incurs higher peak memory than CPUs.

($\mathcal{G}_2$): GEM, although being a replay with exemplars-based method like iCaRL, should not be preferred over iCaRL as its performance remains inferior to those of iCaRL. Also, GEM is computationally expensive as well as requires more storage than iCaRL.

($\mathcal{G}_3$): In a severely resource-constrained environment, EWC and Online EWC can be a reasonable alternative to iCaRL since these methods require less additional storage. Although EWC is a computationally expensive method, the computational cost can be manageable as the IL process is only performed once per task. One can reduce the number of samples used to compute Fisher matrices, which account for the majority of the IL time.

($\mathcal{G}_4$): LwF and SI should be avoided as they offer minimal protection against CF on mobile sensing applications.

($\mathcal{G}_5$): Suppose the available resources such as storage are constrained on the device. In that case, we suggest using iCaRL with a budget size of 1%–5% of training samples as using a higher budget size does not always provide enough benefits if the training dataset size is large (HHAR PAMAP2, and NinaPro (LOUO)). On the other hand, for datasets having smaller training sizes such as Ninapro (Per Subject) and EmotionSense datasets, having a higher budget of 20%–40% helps to a large extent.

3.3 FastICARL

Drawn from our initial investigation above, we identified the bottlenecks of applying CL on resource-constrained devices. In particular, CL methods are computationally heavy, making it difficult to be deployed on resource-limited mobile and embedded platforms. In addition, the exemplar-based method such as iCaRL achieves high accuracy while it requires storing exemplars which impose resource overheads on-device.

In this section, we propose an efficient CL method, FastICARL, that minimises the computationally heavy CL procedure to reduce the end-to-end latency of CL and employs the quantisation scheme to reduce the burdens of storing exemplars (Section 3.3.1). After that, we present the results to demonstrate the effectiveness of FastICARL in terms of CL performance, end-to-end latency, and storage requirements (Section 3.3.2).

3.3.1 Methodology

We first formulate our problem and then propose our CL method, FastICARL.

Problem Formulation

We focus on Sequential Learning Tasks (SLTs) [21] from the audio sensing tasks, where new classes (e.g., different sounds in ESC) can emerge over time. In this section, we employ the third scenario detailed in Section 3.2.1 as it is practical and challenging. Formally, we are given training samples, $X^1, X^2, ...,$, where X^y is a set of samples of class y. Inspired by

Algorithm 2: Construction and quantisation of exemplar sets for iCaRL/FastICARL

Input: Feature Extractor $\mathcal{F}()$, The number of exemplars to be stored m, Quantisation bit b, IL method
Output: Quantised Exemplar set Q
Data: $X = \{x_1, ..., x_n\}$ of class y

$\mu \leftarrow \frac{1}{n} \sum_{i=1}^{n} \mathcal{F}(x_i)$ // calculate class mean
/* find m exemplars out of n samples */
if IL method is *ICARL* **then**
  for $k = 1, ..., m$ **do**
    $p_k \leftarrow \underset{x \in X}{\operatorname{argmin}} \left\| \mu - \frac{1}{k}(\mathcal{F}(x) + \sum_{i=1}^{k-1} \mathcal{F}(p_i)) \right\|$
if IL method is *FastICARL* **then**
  /* calculate feature distance between each sample and class mean */
  for $i = 1, ..., n$ **do**
    $d_i = \mathcal{F}(x_i) - \mu$
  /* build max heap with size k */
  create max heap H of pair {d, index}
  for $k = 1, ..., m$ **do**
    H.insert(d_k, k)
  /* loop over the remaining samples while updating the max heap */
  for $k = m + 1, ..., n$ **do**
    if $d_k < H$.extractMaxDist() **then**
      H.pop() // delete one item from H
      H.insert(d_k, k)
  /* build a sorted exemplar set P */
  for $k = m, ..., 1$ **do**
    $i \leftarrow H$.extractMaxDistIndex(), H.pop()
    $p_k \leftarrow x_i$
for $k = 1, ..., m$ **do**
  $q_k \leftarrow$ Quantise(p_k, b)
$Q \leftarrow (q_1, ..., q_m)$ // Quantised exemplar set

prior works [114, 128], we first train a model on the first task with $N/2$ classes and then incrementally train the model by adding subsequent tasks with one class ($N/2 + 1$ tasks).

FastICARL

Although iCaRL provides impressive performance, it is limited by high computational costs and large storage requirements to maintain sufficient budget size to perform reasonably well. To begin with, iCaRL's high computational loads come from its herding operation (find an exemplar set that has a min distance between the class mean and exemplars mean in feature space), i.e., exemplar selection procedure which is based on the inefficient double for loops (Lines 2-4), resulting in the $O(nm^2)$ complexity (which takes up 70 - 90% of the total IL time). n is the number of examples in a class, and m represents the target number of exemplars. Note that in this work, training time indicates the usual training time with respect to backpropagation, updating weights, while the rest of the time in learning a new task or adding a new class is considered IL time. Thus, instead of relying on herding, FastICARL employs a k-nearest-neighbour search to identify the representative examples to construct exemplar sets. This enables FastICARL to accelerate the process of exemplar construction without performance degradation, as shown in Section 3.3.2. By jointly utilizing the max heap as in Algorithm 2, FastICARL remarkably reduces the complexity of finding m exemplars out of n samples to $O(n(1+log(m))+mlog(m)) = O(nlog(m))$. In detail, the computation of feature distance and the insertion of max heap cost $1 + log(m)$ which is performed on n samples in total. After that, the sorting on m identified exemplars in a max heap costs another $mlog(m)$.

Furthermore, iCaRL requires as much as 69 MB (see Section 3.3.2). To alleviate this storage demand, we apply quantisation on exemplar sets on the fly. Note that since budget sizes take up 72-99% of the storage requirements of FastICARL, we apply quantisation only on exemplars in this work. While constructing exemplar sets, FastICARL converts 32-bit float data to 16-bit float or 8-bit integer types and store them with a smaller budget. When converting between 32-bit float and 8-bit integer, we use quantisation scheme used in [48] to minimise the information loss in quantisation. The scheme utilises an affine mapping of integers q to real numbers r, i.e.,

$$r = S(q - Z) \tag{3.8}$$

for some constant quantisation parameters S and Z. S denotes the scale of an arbitrary positive real number, and Z denotes zero-point of the same type as quantised values q and corresponds to the real value 0.

3.3.2 Evaluation

Datasets

We experiment with our method, FastICARL, in two audio applications as discussed in 3.1. Also, note that audio tasks were selected due to their resource-intensive nature and

practical importance in mobile applications, making them ideal test cases for demonstrating FastICARL's improvements in computational efficiency and storage optimisation.

EmotionSense: For ER application, we employ the EmotionSense dataset [15], and its preprocessing details are described in Section 3.2.2. This dataset contains diverse class granularity, one of which is clustered into five standard emotion groups: (1) Happy, (2) Sad, (3) Fear, (4) Anger, and (5) Neutral (generally used by social psychologists [239]). To diversify our evaluation on iCaRL and FastICARL, we select 5-groups class granularity in this section.

UrbanSound8K: For environment sound classification (ESC) application, we adopt the UrbanSound8K dataset [240] as it is a large dataset that can test the effectiveness of our method on resource-limited devices. UrbanSound8K contains 9.7 hour-long data with 8,732 labelled urban sounds collected in real-world settings. This dataset consists of 10 audio event classes such as car horn, drilling, street music, etc. Following [96], we extracted four different audio features ((1) Log-mel spectrogram, (2) chroma, (3) tonnets, (4) spectral contrast) for each sound clip, sampled at 22 kHz. Using the first three seconds of sound, we created an input of size 128×85, where 128 represents the number of frames and 85 represents the aggregated feature size of the four audio features.

Experimental Setup

Task: As described in Section 3.3.1, we adopt class-incremental learning. Hence, for EmotionSense, two classes are selected as task 1 for training a base model, and then the other three classes are added to the model one by one sequentially. For UrbanSound8K, five classes are used as the first task, and the other five classes are learned incrementally. Note that all reported results in Section 3.3.2 are averaged over five times of experiments.

Model Architecture: To diversify our evaluation over our initial investigation which relies on LSTMs (Section 3.2), we adopt a convolutional neural networks (CNN) architecture from prior work [96] for constructing ER and ESC models. To identify a high-performing and yet lightweight CNN model to operate on embedded and mobile devices, we conducted hyper-parameter search with different number of convolutional layers {2,3,4}, number of convolutional filters {8,16,32}, pooling layer type {max pooling, average pooling}, number of fully-connected (FC) layers {0,1} and its hidden units {128,512,1024}. A basic convolutional layer consists of 3×3 convolution, batch normalisation, and Rectified Linear Unit (ReLU). We found that although the best performing model is a 4-layered CNN with 32 Conv filters followed by an FC layer (Weighted F1-score of 86% for ER and 90% for ESC), the performance degradation without the FC layer is minimal (see Table 3.7) while the majority of the model parameters are consumed in the FC layer as shown in [96]. Hence, as our final CNN architecture, we use [Conv: {32,32,64,64}] for ER and [Conv: {16,16,32,32}] for ESC. We omit an FC layer in both applications, and average pooling

Table 3.7: Average weighted F1-score of baselines and FastICARL according to the budget size ($\mathcal{B} = 5\%, 10\%, 20\%$) in EmotionSense and UrbanSound8K datasets.

	EmotionSense (ER)			**UrbanSound8K (ESC)**		
	5%	10%	20%	5%	10%	20%
iCaRL (32 bits)	0.57	0.60	0.70	0.67	0.69	0.69
iCaRL (16 bits)	0.55	0.63	0.70	0.66	0.67	0.71
iCaRL (8 bits)	0.59	0.62	0.68	0.65	0.68	0.70
FastICARL (32 bits)	0.57	0.62	**0.67**	0.67	0.69	0.70
FastICARL (16 bits)	0.58	0.65	0.68	0.66	0.69	**0.71**
FastICARL (8 bits)	**0.60**	**0.63**	**0.69**	**0.65**	**0.68**	**0.69**
Joint (Upper Bound)		0.83			0.89	
None (Lower Bound)		0.41			0.02	

layers and a 0.5 dropout probability are adopted for the second and fourth Conv layers. ADAM optimiser [241] and a learning rate of 0.001 are used.

Evaluation Protocol: Following prior works [15, 96], the 10% of each class is used as the test set and the remaining as the training data. Additionally, we report the performance of a model trained up to task k incrementally. Also, we report the results based on a weighted F1-score which is more resilient to class imbalances as the employed datasets are not balanced.

Baselines: To evaluate the effectiveness of FastICARL, we include various baselines in our experiments. First, we include a *Joint* model which represents a scenario when the model is trained with training data of all classes available from the beginning. *Joint* serves as a performance upper bound. Second, a *None* model represents a case where a model is fine-tuned incrementally by adding classes to the model without any IL method. *None* can be regarded as a performance lower bound. Thirdly, we include iCaRL with three quantisation levels (32, 16, and 8 bits). Finally, FastICARL (32, 16, and 8 bits) is compared.

Implementation

To evaluate our framework on resource-constrained devices, we implemented it on an embedded (Jetson Nano) and a mobile device (Google Pixel 4 with Qualcomm SM8150 Snapdragon 855)[3]. The Jetson Nano is an embedded mobile platform with four cores and 4 GB RAM. It is often utilised in mobile robotics. We use PyTorch 1.6 to develop and evaluate FastICARL on Jetson Nano. The Google Pixel 4 phone has eight cores and 6 GB RAM. We develop FastICARL based on C++ on the Android smartphone using

[3]Note that we use Google Pixel 4 as it employs the same chipset as OnePlus 7 Pro for the consistency of evaluation.

Table 3.8: Average Latency (IL Time) in seconds for iCaRL and FastICARL on Jetson Nano and a smartphone (Google Pixel 4) for both datasets according to the budget size ($\mathcal{B} = 5\%, 10\%, 20\%$).

	Embedded Device (Jetson Nano)						**Smartphone (Google Pixel 4)**					
	EmotionSense (ER)			**UrbanSound8K (ESC)**			**EmotionSense (ER)**			**UrbanSound8K (ESC)**		
	5%	10%	20%	5%	10%	20%	5%	10%	20%	5%	10%	20%
iCaRL (32 bits)	6.25	7.24	9.35	102	144	271	1.41	1.98	2.73	41.5	75.5	138
iCaRL (16 bits)	6.30	7.40	9.25	100	144	270	1.48	1.99	2.74	44.6	78.8	139
iCaRL (8 bits)	6.27	7.40	9.25	120	178	292	1.43	1.99	3.04	45.4	77.7	146
FastICARL (32 bits)	5.10	5.18	5.18	**60.6**	60.8	60.7	0.88	0.90	**0.83**	10.5	10.8	**10.4**
FastICARL (16 bits)	**4.96**	4.98	5.22	61.1	61.5	**60.6**	0.87	0.89	0.84	10.7	11.2	10.9
FastICARL (8 bits)	5.01	5.07	5.24	67.1	66.3	61.5	0.90	0.91	0.87	10.7	10.7	10.6

mobile deep learning framework, MNN, and the Android Native Development Kit. Note that our implementation of FastICARL on the smartphone enables complete on-device training of new tasks/classes incrementally, unlike other deep learning frameworks on mobile platforms (e.g., PyTorch Mobile) where only on-device inference is supported. The binary size of our implementation on a mobile platform is only 3.8 MB which drastically reduces the burden of integrating the IL functionality into mobile applications given that iCaRL requires as much as 69 MB for UrbanSound8K.

Results

Performance: We first show the average weighted F1-score across all runs for different baselines and IL methods for the EmotionSense and UrbanSound8K datasets in Table 3.7. For both datasets, we present the performance according to the size of the budgets storing exemplars (5%, 10%, and 20%) to analyse trade-offs between the performance and storage requirement of the studied IL methods. Note that the weighted F1-score of the models after all tasks are trained incrementally is reported.

To begin with, the *None* model allows us to confirm that CF occurs without the IL method. Its weighted F1-score drops sharply to 41% for ER and 2% for ESC. In contrast, the *Joint* model achieves as high as 83% and 89% weighted F1-scores for ER and ESC, respectively. ICARL (32 bits) and our proposed IL method, FastICARL (32 bits), can largely mitigate the CF issues observed in the None model. With a budget size of 20%, iCaRL provides a high weighted F1-score of 70% for ER and 69% for ESC. Likewise, FastICARL achieves a similar performance (67% for ER and 70% for ESC) to that of iCaRL, which stays close to the upper bound performance of the *Joint* model. Furthermore, we find that the impact of the information loss due to the quantisation of the saved exemplars for both iCaRL and FastICARL is minimal. As shown in Table 3.7, all four variants, such as iCaRL (16 and 8 bits) and FastICARL (16 and 8 bits), achieve similar performance to their original counterparts.

Finally, we study the importance of the storage budget parameter. We present the performance of our IL method according to its budgets of 5%, 10%, and 20% of total training samples. In general, the more samples are used as exemplars, the higher the weighted F1-score the IL method can achieve. We also find that our method (FastICARL) needs only a 5% budget size to achieve a weighted F1-score of 60-64% and successfully retain its weighted F1-score even after losing some information by applying quantisation up to 8 bits on its exemplars.

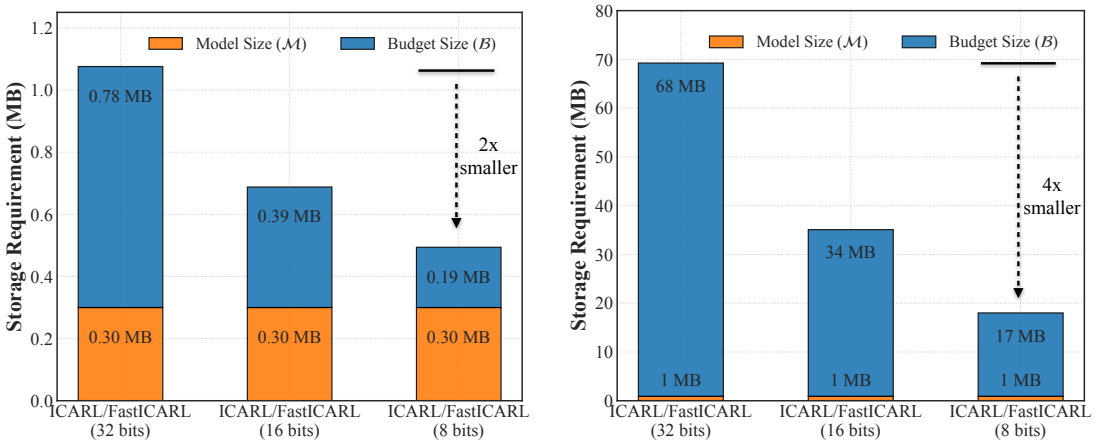
Latency: We measure the computational costs of sequentially learning additional classes based on a pre-trained model. The average IL time to run different IL methods is presented in Table 3.8. The IL time of FastICARL (32, 16, and 8 bits) ranges 4.96-67.1 seconds on Jetson Nano and 0.83-11.2 seconds on Google Pixel 4 depending on the budget and datasets. FastICARL remarkably reduces the IL time by 18-78% on Jetson Nano and 37-92% on Google Pixel 4 compared to iCaRL. Note that the training time of iCaRL and FastICARL is approximately the same (these results are omitted for brevity). Also, FastICARL (16 and 8 bits) shows substantial improvement in IL time: this indicates that the additional operation of quantizing exemplars does not impose a meaningful burden on the system.

Storage: We now show the storage overhead of the IL method. The size of FastICARL is composed of the model parameter size ($\mathcal{M}$) and budget size ($\mathcal{B}$). As FastICARL relies on stored exemplars, its storage demand is primarily driven by the number of exemplars to be stored, i.e., budget size ($\mathcal{B}$). As shown in Figure 3.6, FastICARL requires at most 0.49 MB for the EmotionSense dataset and 18 MB for the UrbanSound8K dataset, decreasing the storage requirement by 2 to 4 folds over iCaRL. Model sizes for EmotionSense and UrbanSound8K datasets are fixed as 0.3 MB and 1 MB, respectively.

Based on the results in this section, we have demonstrated that FastICARL enables faster IL by reducing the IL time and storage requirements by applying quantisation.

3.4 Conclusion

In this chapter, we first studied the CF problem using six prominent CL methods based on three representative sensing applications (i.e., HAR, GR, and ER) in three CL scenarios with varying complexities. With our end-to-end CL framework implemented on Nvidia Jetson Nano and a smartphone (OnePlus 7 Pro), we conducted extensive experiments to investigate CL methods' performance, generalisability, and trade-offs of storage, computational costs, and memory footprints. We first identified that CF occurs in mobile and embedded sensing applications when CL methods are not used. We also found that while most CL methods solve the CF in simple scenarios, only iCaRL among the compared methods can successfully alleviate CF issues in more challenging scenarios across the



(a) EmotionSense (b) UrbanSound8K

Figure 3.6: Comparison of the storage requirement ($\mathcal{M} + \mathcal{B}$) for iCaRL and FastICARL (32, 16, and 8 bits) based on 20% budget size in each dataset.

employed datasets. Furthermore, we demonstrated that the CL approaches incur minor to modest storage, peak memory usage, and latency overheads, thereby saving a considerable amount of computational resources on-device compared to a case when training is done from scratch whenever a new class/task is added to the system. Finally, based on those findings, we discuss potential guidelines for practitioners and researchers interested in applying CL to edge platforms.

Our systematic analysis of CL in mobile and embedded systems allows us to identify the bottlenecks of applying CL in resource-constrained devices. Based on these insights and findings, we developed an end-to-end and on-device CL framework, FastICARL, that enables efficient and accurate CL in mobile sensing applications. We implemented FastICARL on two resource-constrained devices (Jetson Nano and Google Pixel 4) and demonstrated its effectiveness and efficiency. FastICARL decreases the IL time up to 78-92% by optimizing the exemplar construction procedure and also reduces the storage requirements by 2-4 times by quantising its exemplars without sacrificing performance.

Chapter 4

Bringing On-Device ML from Edge to Microcontrollers: YONO

4.1 Introduction

In Chapter 3, we investigated the applicability and trade-offs of various CL methods in mobile sensing applications regarding accuracy, computational costs, memory and storage. Building on this analysis, we proposed an efficient CL method that optimises resource usage of CL on mobile devices. Subsequently, in Chapters 4 and 5, we explore the systematic and algorithmic approaches to seamlessly support efficient execution of CL and on-device training on even more resource-constrained devices such as IoT devices and MCUs.

While ML models are becoming more efficient on resource-constrained IoT devices [127], most existing on-device systems designed for MCUs target one specific application [98, 161, 242]. However, multi-application systems capable of supporting a wide range of applications on-device could be more versatile and useful in practice, potentially facilitating on-device training and CL in multi-user scenarios. Therefore, this chapter aims to design an MCU-powered system capable of recognising users' voice commands, activities, and gestures, as well as identifying everyday objects to understand surrounding environments. Such a system has the potential to boost the utilisation of IoT devices in practice (*e.g.,* assist visually impaired individuals to understand their environments [243]).

However, realising such a multi-tasking system faces three major challenges:

1. **Multiple Dissimilar Tasks:** The system must accommodate multiple tasks based on different modalities of incoming data (*e.g.,* voice recognition (audio), activity recognition (accelerometer signals), object classification (image)) within the same framework. As discussed in [189], conventional multi-task learning (MTL) approaches

cannot effectively address *multiple heterogeneous networks*.

2. **Resource Constraints:** IoT devices based on MCUs are extremely resource-constrained [67, 244]. For example, "high-end" MCUs (*e.g.,* STMF767ZI) have only 512 KB Static Random-Access Memory (SRAM) for intermediate data and 2 MB on-chip embedded flash (eFlash) memory for program storage.

3. **Context Switching Overhead:** In real-world deployment scenarios, context switching of different ML tasks at run-time could incur significant overheads on memory-constrained MTL systems, as demonstrated in [189], where some models must reside in external storage devices due to the limited on-chip memory space. As on-chip memory operations are faster than external disk accesses, frequent model loading/swap between different tasks based on external storage increase the overall latency, exacerbating system usability and responsiveness.

To solve these challenges, one of the common techniques employed is to compress individual models separately using pruning [164, 169] and quantisation [48] as described thoroughly in Section 2.3.1. However, model compression techniques are limited as they require extensive and iterative finetuning to ensure high performance after compression. Additionally, since models are trained independently, they cannot benefit from potential knowledge transfer between different tasks. In the literature, researchers proposed MTL-based approaches to achieve robustness and generalisation of multiple tasks, while increasing the compression rate of the model by sharing network structures (see Section 2.3.2 for further details). However, *sharing/compressing multiple heterogeneous networks* has not been fully examined. Furthermore, prior work [189] attempts to solve the MTL of multiple heterogeneous networks by sharing weights of multiple models via virtualisation. This method is complex, and its compression ratio is constrained to 8.08× (see Section 4.4.2 for detail), restricting the type of IoT devices on which it can operate. Moreover, since only a simplified LeNet architecture is evaluated on an MCU, the system could not achieve high accuracy to be useful in practice (*e.g.,* 59.26% on the CIFAR-10 dataset [1]).

This Work. To address the challenges and limitations of previous approaches, we propose **YONO** (**Y**ou **O**nly **N**eed **O**ne pair of codebooks), that adopts Product Quantisation (PQ) [50] to maximise compression rate and on-chip memory operations while minimising external disk accesses for a heterogeneous multi-tasking system. PQ, originally proposed in the database community, aims to decompose the original high-dimensional space into the Cartesian product of a finite number of low-dimensional subspaces that are independently quantised. A model's weight matrix of any layer can be converted to codeword indexes corresponding to the subvectors of the weight matrix via a codebook.

Inspired by successful applications of PQ on approximate nearest neighbour search out of billions of vectors in the database community [50, 187, 188] and single layer compression in individual models [147, 185, 245, 246, 247], we jointly apply PQ on multiple models instead

of on a layer of a model. We find just one pair of codebooks that are generalisable and thus can be shared across many dissimilar tasks. We then propose a novel optimisation process based on alternating PQ and finetuning steps to mirror the performance of the original models. Further, we introduce heuristics to consider the weight differences between the layers of the original model and the reconstructed layers from the codebooks to maximise the compression rate and accuracy. Finally, we develop an efficient model execution and switching framework to operate multiple heterogeneous models targeted for different tasks, reducing the overhead of context switching (*i.e.,* model swap between tasks) at run-time.

YONO is comprised of two components:

1. An offline phase in which a shared PQ codebook is learned and multiple models are incorporated. We implement this phase on a server.
2. An online phase in which multiple heterogeneous models are deployed on an extremely resource-constrained device (MCUs).

To evaluate YONO, we first evaluated four image datasets and one audio dataset used in SOTA prior work on heterogeneous MTL [189] for a fair comparison. We show that YONO achieves high accuracy of 93.7% on average across the five datasets, which is a 15.4% improvement over [189] due to our usage of the optimised network architecture (see Section 4.4.2 for detail) and is very close to the accuracy of the uncompressed models (0.4% loss in accuracy).

To evaluate the scalability of YONO to other modalities, we include data from modalities such as accelerometer signals from Inertial Movement Units (IMU) for human activity recognition (HAR) and surface electromyography (sEMG) signals for gesture recognition (GR). We then demonstrate that YONO effectively retains the accuracy of the uncompressed models across all the employed datasets of four different modalities (Image, Audio, IMU, sEMG).

To evaluate the generalisability of the learned codebooks of YONO, we apply YONO to compress new models trained on unseen datasets during the codebook learning in the offline phase. Surprisingly, YONO can maintain the accuracy of the uncompressed models and achieve a 12.37× compression ratio (53.1% higher than [189]).

Finally, we evaluate the online component of YONO on the largest model and the smallest model to show the upper bound and lower bound results, respectively. We employ an MCU, STM32H747XI (see Section 4.3 for details), and demonstrate that YONO enables efficient in-memory execution (latency of 16-159 ms and energy consumption of 3.8-37.9 mJ per operation) and model loading/swap framework for task switching (showing reductions of 93.3-94.5% in latency and 93.9-95.0% in energy consumption compared to the method using external storage access).

4.2 YONO

In this section, we first present the overview of our multitasking system, YONO (Section 4.2.1). Then, we introduce the background on PQ and its applications on single model compression (Section 4.2.2). We then explain how we utilise PQ to compress multiple heterogeneous networks into a pair of codebooks. The networks can be of any arbitrary architecture that consists of fully connected layers and convolutional layers. After that, we present our novel network optimisation process to ensure the performance of the compressed networks remain close to original models (Section 4.2.4). On top of that, based on an observation (detailed in Section 4.2.5), we further propose optimisation heuristics to maximise the performance gain with a minimal loss of the compression rate when using PQ-based compression. Finally, we describe our in-memory execution and model swapping framework on MCUs (Section 4.2.6).

4.2.1 Overview

In this subsection, we describe the overview of YONO that learns codebooks to represent the weights of multiple heterogeneous neural networks as well as enable on-chip memory operations on resource-constrained devices. In particular, YONO is composed of two components: (1) an offline phase where YONO learns a pair of codebooks on pretrained neural networks using PQ (will be explained in detail in Section 4.2.2) and (2) an online phase where YONO enables on-chip execution such as model execution and model loading/swapping. Note that we assume that the overall size of multiple neural networks is larger than the operational limit of the on-chip eFlash memory and SRAM of the targeted IoT devices. For example, in Section 4.4, we employ seven different models with a total size of 3.84 MB and evaluate our framework on MCU (STM32H747XI), which strictly has only 512 KB of SRAM and 1 MB of eFlash.

4.2.2 Product Quantisation and Compressing Single Neural Network

We now provide an introduction to PQ and how it is used to compress a single model. PQ can be considered a special case of vector quantisation (VQ) [248], in which it attempts to find the nearest codeword, $\mathbf{c}$, to encode a given vector, $\mathbf{w}$. Suppose we are given a codebook, $\mathbf{C}$, that contains a set of representative codewords, we can reconstruct/approximate the given vector $\mathbf{w}$ by using $\mathbf{c}$ and its associated index in the codebook. Thus, given a vector $\mathbf{w} \in \mathbb{R}^d$ to be encoded, the encoding problem of VQ can be formulated as follows.

$$\underset{b}{\operatorname{argmin}} \|\mathbf{w} - \mathbf{C}b\|^2 \tag{4.1}$$

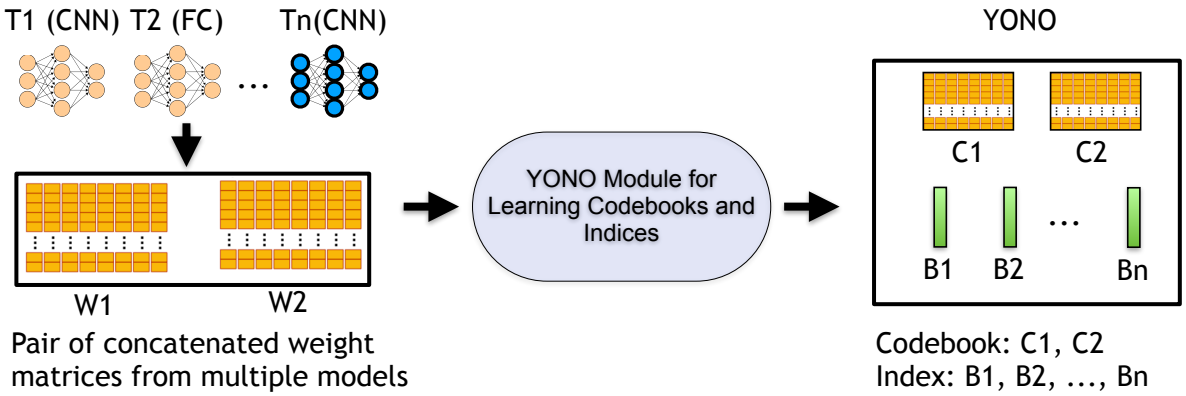


Figure 4.1: Overview of the offline component of YONO. The offline module employs PQ to learn a pair of codebooks and identify indices to represent multiple heterogeneous neural networks. This module incorporates our novel optimisation process and heuristics to minimise the accuracy loss compared to the original models.

where $\mathbf{C}$ is a d-by-K matrix containing K codewords of length d, and b is called a code (i.e., index of codebook pointing to a codeword, $\mathbf{c}$, nearest to the given vector, $\mathbf{w}$). $\|\cdot\|$ is a l_2 norm. Solving Equation 4.1 is equivalent to searching the nearest codeword. Besides, the codebook, $\mathbf{C}$, is learned by running the standard k-means clustering over all the given vectors [50].

The PQ is a particular case of VQ when the learned codebook is the Cartesian product of sub-codebooks. Given that there are two sub-codebooks, the encoding problem of PQ is as follows.

$$\begin{aligned} &\underset{b}{\operatorname{argmin}} \left\|\mathbf{w} - \mathbf{C}b\right\|^2, \\ &s.t. \quad \mathbf{C} = \mathbf{C}_1 \times \mathbf{C}_2 \end{aligned} \tag{4.2}$$

where $\mathbf{C}_1$ and $\mathbf{C}_2$ are two sub-codebooks of $\frac{d}{2}$-by-K matrices. Since any codeword of $\mathbf{C}$ is now the concatenation of a codeword of $\mathbf{C}_1$ and a codeword of $\mathbf{C}_2$, PQ can have K^2 different combinations of codewords. If a vector is divided into M partitions, then PQ can have K^M combinations of codewords. The number of sub-codebooks, M, can be any number between 1 and the length of the given vector, d (e.g., 1, 2, ..., d). When M is set to 1, it is VQ. When M is set to d, it is equivalent to the scalar k-means algorithm.

We now describe how the encoding problem of PQ can be applied to compress a neural network. It is because instead of storing weight matrix $\mathbf{W}$ of any layer in neural networks explicitly, we can learn an encoding $\mathcal{B}(\mathbf{W})$ that needs much less storage space. Using the found encoding $\mathcal{B}$ and a learned codebook $\mathbf{C}$ based on PQ, we can reconstruct $\widehat{\mathbf{W}}$ which approximates the original weight matrix $\mathbf{W}$ of the layer. If we can find $\widehat{\mathbf{W}}$ close enough to

W, the reconstructed layer of a neural network will perform normally as demonstrated in prior works using PQ to compress a single neural network [185, 246].

4.2.3 Compressing Multiple Heterogeneous Networks

As described in Section 4.2.2, PQ is typically used to compress a single model in machine learning literature [245, 247]. In prior works, each layer is replaced by one small-sized codebook (e.g., K=256, D=8, M=1), and a high compression rate and little performance loss are achieved in large computer vision models with more than 10 M parameters (e.g., ResNet50 [249]). However, in small-sized models that are specially designed to be used on MCUs (i.e., the number of parameters is at most around 500K-1M), the same approach (having a codebook for each layer) no longer provides a high compression rate due to the overhead of storing many codebooks. Therefore, in our system, we propose to apply PQ to one or multiple neural networks while only sharing a pair of the learned codebooks to maximise the compression ratio. We will explain how we ensure high performance of the compressed models in the next subsections (Section 4.2.4 and Section 4.2.5).

As in Figure 4.1, we first concatenate weights of all the models of different tasks (i.e., $T_1, T_2, ..., T_n$). Then, we construct two weight matrices, W_1 and W_2, so that YONO takes into account spatial information of convolutional layer kernels as in other prior works [246]. For one weight matrix, W_1, we combine convolutional layers with a kernel size of 3×3. Then, in the other weight matrix, W_2, we concatenate convolutional layers with kernel size 1×1 and fully-connected layers. Then these concatenated weight matrices, W_1 and W_2, are given as an input to learn codebooks, C_1 and C_2, for different kernel sizes, respectively. Note that we also observed that neglecting such information in learning codebooks leads to worse performance. In our system design, we select kernel sizes of 3×3 and 1×1 as those are widely used kernel sizes in many of the optimised network architectures [152, 153, 154]. Also, since FC layers are essentially the same as point-wise convolution operation (i.e., kernel size of 1×1), we combine weights of FC layers together with those of 1×1 kernel convolution layers. Besides, we set M to 2 throughout our evaluation so that YONO can leverage the implicit codebook size of K^M. We observed that when M is 1, the codebook is not generalisable enough to compress multiple neural networks. When M is set to 3, the overhead of the codebooks decreases the compression rate without providing much accuracy benefit.

4.2.4 Network Optimisation

After learning a pair of codebooks for multiple models as in Section 4.2.3, YONO performs finetuning on the reconstructed model in order to adjust the loss of information due to the compression (see Algorithm 3). As studied in [169], weights in the first and last layer of a model are the most important. Thus, in the finetuning stage, we select the first and

last layer of a model and finetune them (Lines 2-4). The finetuning step largely recovers the accuracy of the original model by re-adjusting the first and last layer of the model according to the different weights induced by the codebooks. However, as we will show in our evaluation in Section 4.4 (this incurs 2-8% accuracy loss), a simple extension of PQ to multiple heterogeneous neural networks with a finetuning step cannot ensure high accuracy due to the increased weight differences between original models' weight matrices $\mathbf{W}_{T_1,\ldots,T_n}$ and reconstructed models' weight matrices $\widehat{\mathbf{W}}_{T_1,\ldots,T_n}$ although it shows a high compression rate.

Therefore, we introduce an optimisation process to improve the performance of the decompressed models. As discussed in prior works [185, 245], in general, higher weight differences (i.e., errors) result in increased loss of accuracy. Thus, to minimise the impact of the weight differences, we adopt to use the iterative optimisation procedure, inspired by the Expectation-Maximisation (EM) algorithm [250] and prior work [245]. We iteratively adjust the weight drifts by reassigning indices on the updated weights from finetuning as the E-step (Lines 12-13) and by finetuning several selected layers (e.g., first and last layers) as the M-step (Lines 14-17). Note that our optimisation procedure is novel in that (i) we perform network optimisation across multiple heterogeneous networks and (ii) we do not update codewords in our learned codebooks since we want our codebooks to be generalisable to compress unseen models and datasets during the codebook learning procedure, different from single model compression methods [185, 245, 246, 247]. In Section 4.4, we demonstrate the generalisability of our learned codebooks and our system on new models trained on new datasets that YONO did not see in its codebook learning.

4.2.5 Optimisation Heuristics

In addition, we further propose an optimisation heuristic that can maximise performance improvement while ensuring a high compression rate. We observed that weight differences of each layer ($\mathbf{W}$ and $\widehat{\mathbf{W}}$) are not uniformly distributed. Besides, the number of parameters in each layer is considerably different. For example, MicroNet-KWS-M [242] (we adopt this network architecture in our evaluation. Refer to Section 4.4 for detail) contains 12 convolutional and FC layers. Among them, one convolutional layer has a 4-dimensional weight matrix ($\mathbf{W} \in \mathbb{R}^{C_{cout} \times C_{in} \times k \times k}$) with a size of {140, 1, 3, 3} which has 1,260 parameters, whereas another convolutional layer in the same model can have weight matrix with a size of {196, 112, 1, 1} which has 21,952 parameters. The latter has 17.4 times more parameters than the former. Thus, based on this observation, we propose our novel optimisation heuristic to select layers for finetuning that have the largest weight difference and contain the least number of parameters (refer to Lines 22-24 in Algorithm 3). Hence, given a network $\mathbf{W}$ with L layers, we attempt to find a layer ℓ as follows.

$$\underset{\ell}{\operatorname{argmax}} \left\| \mathbf{W}^{\ell} - \widehat{\mathbf{W}}^{\ell} \right\|^2 / N^{\ell} \tag{4.3}$$

where $\mathbf{W}^{\ell} - \widehat{\mathbf{W}}^{\ell}$ is a weight difference of weight matrices of the layer ℓ, and N^{ℓ} is the number of the parameters of the layer ℓ.

Algorithm 3: YONO's network optimisation and heuristics for a given task t

Input: Model weights $\mathbf{W}$, model indices $\mathbf{b}$, PQ codebooks $\mathbf{C}$, the number of layers L, error threshold ϵ, heuristics

Output: Reconstructed model weights $\widehat{\mathbf{W}}$, model indices $\hat{\mathbf{b}}$

Data: Train data $\mathcal{D}^{TRAIN}$, Test data $\mathcal{D}^{TEST}$

/* Perform an initial finetuning step */

$\widehat{\mathbf{W}} \leftarrow \mathbf{C}(\mathbf{b})$ // reconstruct a model via codebooks and indices

for $\ell = 2, ..., L-1$ **do**

  FreezeWeights($\widehat{\mathbf{W}}^{\ell}$)

// run network training (e.g., BackProp) with loss function

Finetune($\widehat{\mathbf{W}}, \mathcal{D}^{TRAIN}$)

$acc\_orig \leftarrow$ Evaluate($\mathbf{W}, \mathcal{D}^{TEST}$))

$acc\_recon \leftarrow$ Evaluate($\widehat{\mathbf{W}}, \mathcal{D}^{TEST}$))

if $acc\_orig - \epsilon \leq acc\_recon$ **then**

  return $\widehat{\mathbf{W}}, \mathbf{b}$

/* Perform a further network optimisation step */

$S \leftarrow (1, L)$ // finetuning layer set

$\hat{\mathbf{b}} \leftarrow \mathbf{b}$

for $i = 1, ..., L-2$ **do**

  // E-step: code re-assignment

  for $\ell \notin S$ **do**

    $\hat{\mathbf{b}}^{\ell} \leftarrow \underset{b \in \hat{\mathbf{b}}^{\ell}}{\text{argmin}} \left\| \hat{\mathbf{w}}^{\ell} - \mathbf{C}b \right\|^2$

  // M-step: model update

  $\widehat{\mathbf{W}} \leftarrow \mathbf{C}(\hat{\mathbf{b}})$

  for $\ell \notin S$ **do**

    FreezeWeights($\widehat{\mathbf{W}}^{\ell}$)

  Finetune($\widehat{\mathbf{W}}, \mathcal{D}^{TRAIN}$)

  $acc\_orig \leftarrow$ Evaluate($\mathbf{W}, \mathcal{D}^{TEST}$))

  $acc\_recon \leftarrow$ Evaluate($\widehat{\mathbf{W}}, \mathcal{D}^{TEST}$))

  if $acc\_orig - \epsilon \leq acc\_recon$ **then**

    return $\widehat{\mathbf{W}}, \hat{\mathbf{b}}$

  if heuristics is *OURS* **then**

    // choose a layer to finetune based on our heuristics

    $\ell \leftarrow \underset{\ell}{\text{argmax}} \left\| \mathbf{W}^{\ell} - \widehat{\mathbf{W}}^{\ell} \right\|^2 / N^{\ell}$

    $S \leftarrow (S, \ell)$

In summary, through the optimisation heuristics, YONO identifies a layer with the highest weight difference per parameter. After that, YONO finetunes the identified layer using our network optimisation process introduced in Section 4.2.4. The process continues until the reconstructed model's accuracy is recovered to the target accuracy (Lines 20-21), i.e., accuracy loss is less than a given threshold ϵ (e.g., 2-3% in our evaluation). The number of layers to be finetuned is less than or equal to three in most cases. This process helps YONO maximise the compression ratio (small storage overhead) while retaining the accuracy of its compressed models close to their corresponding original (uncompressed) models. Note that the finetuned layers are then quantised into 8-bit integers in the online component of YONO as described in the next subsection.

4.2.6 In-memory Execution and Model Swap Framework on MCUs

Having established the offline component of YONO, we now turn our attention to the online component of our system. At runtime, the online component of YONO enables the fast and efficient in-memory execution and model swap of multiple heterogeneous neural networks. Figure 4.2 illustrates the overview of the online component of YONO.

Data Structure for Deployment on MCUs: To begin with, we describe the data structures that are necessary for deploying ML models on MCUs. First, YONO requires one pair of learned PQ codebooks, model indices, and other relevant information to reconstruct a model. In addition, YONO needs a task executor to run the reconstructed model in-memory and a task switcher to swap an in-memory model to another reconstructed model.

Learned Codebooks: As described in Sections 4.2.2-4.2.5, YONO learns a pair of codebooks by applying PQ on multiple heterogeneous neural networks with our novel optimisation procedure. Since SRAM is a scarce resource on MCUs, the codebooks are stored on eFlash. Also, because the codebooks are shared across different models compressed by YONO and static during runtime, they are stored on the read-only memory of eFlash.

Model Indices and Other Elements: Once a model is compressed through our system, YONO generates model indices that correspond to the weights of an original model via the learned codebooks and other relevant elements necessary to reconstruct the uncompressed model. For example, relevant elements include model architecture, operators, quantisation information, and so on.

Task Executor: We now present the explanation of our task executor. As we adopt TensorFlow Lite for Microcontrollers (TFLM) [127] to run the deployed model on MCUs, YONO also follows its model representation and interpreter-based task execution. As

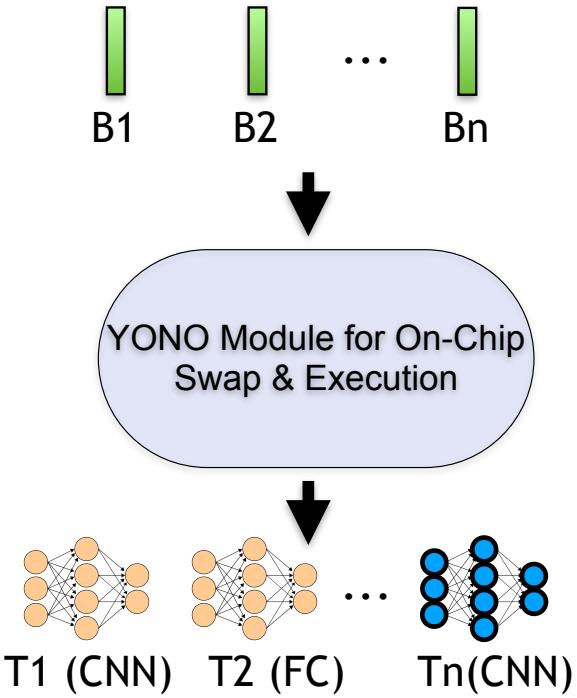


Figure 4.2: Overview of the online component of YONO. The online module enables fast and efficient model loading/swap and in-memory execution.

model representation on MCUs, the stored schema of data and values represent the model. The schema is designed for storage efficiency and fast access on mobile and embedded platforms. Therefore, it has some features that help ease the development of MCUs. For example, operations are in a topologically sorted list instead of a directed-acyclic graph, making conducting calculations be a simple looping through the operation list in order. In addition, YONO adopts interpreter-based task execution by relying on TFLM. Thus, the interpreter refers to the schema of the model representation and loads a model. After that, the interpreter handles operations to execute. Since YONO adopts an interpreter-based task executor and loads a model in the main memory for execution, YONO allows model switching at run time, which is not allowed with the code-generator-based compiler method [251] because this method requires recompilation to switch a model.

Task Switcher: When a task needs to be switched (e.g., the target application is switched from image classification to voice command recognition), YONO replaces the loaded model in the memory with a new model to be executed. Using the same memory space between previous and new models, YONO can operate multiple models within a limited memory budget of SRAM. In addition, since YONO does on-chip memory operations to perform

execution and model swap, YONO improves the response time and end-to-end execution time of different applications. It is because the access time to secondary storage devices is slower than that to internal memory and primary storage. Moreover, a system relying on external storage devices may have unpredictable overheads. For example, disk-writes on storage devices like flash and solid-state drives need to erase an entire block before a write operation.

Model Reconstruction: We now describe our model reconstruction scheme. To reconstruct a model, YONO utilises the PQ codebooks, indices, and relevant elements, such as batch normalisation layer's mean and variance, quantisation information, stored in eFlash. The overall process is as follows. First, YONO retrieves model weights by matching indices of a model to be loaded on the main memory and its corresponding codewords of the PQ codebooks. Secondly, YONO loads relevant elements of the model and then writes this information and model weights to the preallocated memory address for the model on the main memory.

In addition, each value of the learned codewords in the PQ codebooks is stored in 16-bit float instead of 32-bit float type to further reduce the storage requirements on eFlash. In contrast, the weights of the model loaded on the main memory and executed need to be quantised to 8-bit integers. Thus, while loading each layer of the model, YONO converts 16-bit floats to 8-bit integers using the saved quantisation information. Specifically, we use the quantisation scheme used in [48] to minimise the information loss in quantisation. We utilise an affine mapping of integer q to real number r for constant quantisation parameters S and Z, i.e., $r = S(q - Z)$. S denotes the scale of an arbitrary positive real number. Z denotes zero-point of the same type as quantised value q, corresponding to the real value 0. As a result, the reconstructed model in the online component is based on 8-bit integers, and thus the use of codebooks does not affect computations of model execution.

4.3 System Implementation

We introduce the hardware and software implementation of YONO[1].

Hardware. The offline component of our system is implemented and tested on a Linux server equipped with an Intel Xeon Gold 5218 CPU and NVIDIA Quadro RTX 8000 GPU. This component is used to learn PQ codebooks and find indices for each model to be compressed. Then, the online component of our system is implemented and evaluated on an MCU, STM32H747XI, having two cores (ARM Cortex M4 and M7) with 1 MB SRAM and 2 MB eFlash in total. However, our implementation of YONO uses only one core (ARM Cortex M7) since MCUs are typically equipped with one CPU core. We restrict the usage space of SRAM and eFlash to 512 KB and 1 MB, respectively, to enforce stricter

[1] https://github.com/theyoungkwon/YONO

resource constraints.

Software. We use PyTorch 1.6 (deep learning framework) and Faiss (PQ framework) to develop and evaluate the offline component of YONO on the Linux server. At the offline phase, we develop YONO using Python on the server and examine the accuracy of the models. In addition, we develop the online component of YONO using C++ on STM32H7 series MCUs. For running neural networks on MCUs, we rely on TFLM. Since eFlash memory of MCUs is read-only during runtime, YONO loads the model weights on SRAM (read-write during runtime) and swaps the models by replacing the models' weights using PQ codebooks and indices stored on eFlash. The binary size of our implementation on an MCU is only 0.41 MB, and the total size of PQ codebooks, indices, and other information to compress the eight heterogeneous networks evaluated in Section 4.4.4 is 0.35 MB. Note that the memory requirement of the seven models is 4.19 MB, which is 12.05× of what YONO requires and 4.19× of what typical MCUs with 1 MB storage can support.

4.4 Evaluation

We now present the results of the evaluation on our system. Section 4.4.1 describes our experimental setup. We evaluate the effectiveness of our system in the offline phase regarding the performance (i.e., accuracy) and compression rate of the compressed models in an MTL scenario. To make a comparison with prior work [189] that tackles MTL of different neural networks, we begin with evaluating our system with the same datasets used in [189] consisting of five datasets for two modalities (i.e., image and audio) (Section 4.4.2). After that, we evaluate our system to what extent it can address multiple heterogeneous networks trained with different modalities. Thus, we employ four different modalities of data ((1) Image, (2) Audio, (3) IMU, (4) sEMG) by adding two more datasets in order to demonstrate the scalability of YONO on diverse modalities in Section 4.4.3. Further, to demonstrate the generalisabilty of YONO's learned codebooks, we select two additional datasets in each of the four modalities and evaluate our system to compress new models trained on these datasets that YONO did not learn during its codebook learning stage (Section 4.4.4). Finally, we present the results of our online in-memory model execution and swap operations in Section 4.4.5.

4.4.1 Experimental Setup

Task

Our target application scenarios are based on dealing with dissimilar multitask learning. For example, those applications are image classification, keyword spotting, human activity recognition, and gesture recognition.

Evaluation Protocol

Following prior works [15, 96], 10% of data is used as the test set and the remaining as the training set. In addition, to evaluate the effectiveness of the offline phase component of our system, we report the accuracy and compression rate of the compressed models using our system. We also use compressed model's error rate (i.e., accuracy loss) compared to the original model. Then, to evaluate the efficiency of the online phase component of our system, we report the execution time and load/swap time of the models on MCU.

Baseline Systems

To evaluate the effectiveness of our work, **YONO**, we include various baselines in our experiments as follows.

NWV: Neural Weight Virtualisation (NWV) [189] is the state-of-the-art heterogeneous MTL system that treats weights of neural networks as consecutive memory locations which can be virtualised and shared by multiple models. Note that we use reported results of [189] on an MCU, which relies on simplified LeNet architecture.

Scalar Quantisation (Int8): This baseline compresses a single model by quantizing 32-bit floats into low-precision fixed-point representation (e.g., 8-bit) [48, 200]. As in [200], we employ both post-training quantisation and quantisation-aware training schemes. We then report the results of the best-performing scheme in our evaluation. Besides, we only include 8-bit quantisation as sub-byte datatypes (e.g., 4-bit or 2-bit) are not natively supported by MCUs [242]. We leave sub-byte quantisation as future work.

PQ-S: This baseline uses PQ to compress a single model to a pair of the shared codebooks across layers in the model. As this baseline does not share the codebooks across multiple models, this can serve as a baseline for the single model compression and as the lower bound in compression ratio among the PQ variants.

PQ-M: This baseline uses PQ to compress multiple heterogeneous models to a pair of the shared codebooks but does not apply our optimisation process and heuristics as described in Section 4.2. We include this to conduct an ablation study to evaluate the impact of the proposed optimisation in our system.

PQ-MOpt: This baseline uses PQ to compress multiple heterogeneous models to a pair of the shared codebooks and also apply the optimisation process without the heuristics described in Section 4.2. We include this to conduct an ablation study to evaluate the impact of the heuristics in our system.

Uncompressed (Original): An original model before compression. It is pretrained with available training data and serves as the upper bound in terms of the accuracy metric.

Table 4.1: Summary of datasets, model architectures, mobile applications used in Section 4.4.2 and Section 4.4.3.

Modality	Dataset	Architecture	Mobile Application
Image	MNIST	LeNet	Digit recognition
	CIFAR-10	MicroNet-AD	Object recognition
	SVHN	MicroNet-AD	Digit recognition
	GTSRB	MicroNet-AD	Road sign recognition
Audio	GSC	MicroNet-KWS	Keyword spotting
IMU	HHAR	MicroNet-AD	Activity recognition
sEMG	Ninapro DB2	Lightweight CNN	Gesture recognition

4.4.2 Performance

Following [189], we start by evaluating YONO in MTL scenarios on two modalities: images and audio signals which are widely used data modalities in mobile sensing applications.

Datasets. We employ the same datasets used in the prior work [189] to make a fair comparison. First, four image datasets are employed, namely MNIST [252], CIFAR-10 [1], SVHN [70], and GTSRB [74] associated with classifying objects of handwritten digits (grayscale), generic objects, numbers (RGB), and road signs, respectively. Then, one audio dataset of Google Speech Commands V2 (GSC) [253] for keyword spotting is used.

Model Architecture. We adopt optimised neural network architectures, designed to be used in the resource-constrained setting, such as variants of MicroNet [242], simplified LeNet used in [189]. For MNIST, we use the simplified LeNet as it is used in [189] and the accuracy of such LeNet variant is very high at 98%. For other datasets (CIFAR-10, SVHN, GTSRB, GSC), we use variants of MicroNet architecture to construct pretrained models. To identify a high-performing and yet lightweight model to operate on embedded and mobile devices, we conduct a hyper-parameter search based on different variants of MicroNet (e.g., small, medium, large models), lightweight convolutional neural network (CNN) architectures [51], the number of convolutional filters. A basic convolutional layer consists of 3×3 convolution, batch normalisation, and Rectified Linear Unit (ReLU). Then, as our final model architectures, we use MicroNet-KWS-M for GSC and MicroNet-AD-M (with the reduced number of convolutional filters $\{192\}$) for CIFAR-10, SVHN, GTSRB. Throughout model training for all of the datasets, ADAM optimiser [241] and learning rate of 0.001 are used. The datasets, architectures, and applications are summarised in Table 4.1.

Accuracy. We show the accuracy results here. Figure 4.3 shows the accuracy of each baseline so that we can analyse the impact of our proposed techniques in our system.

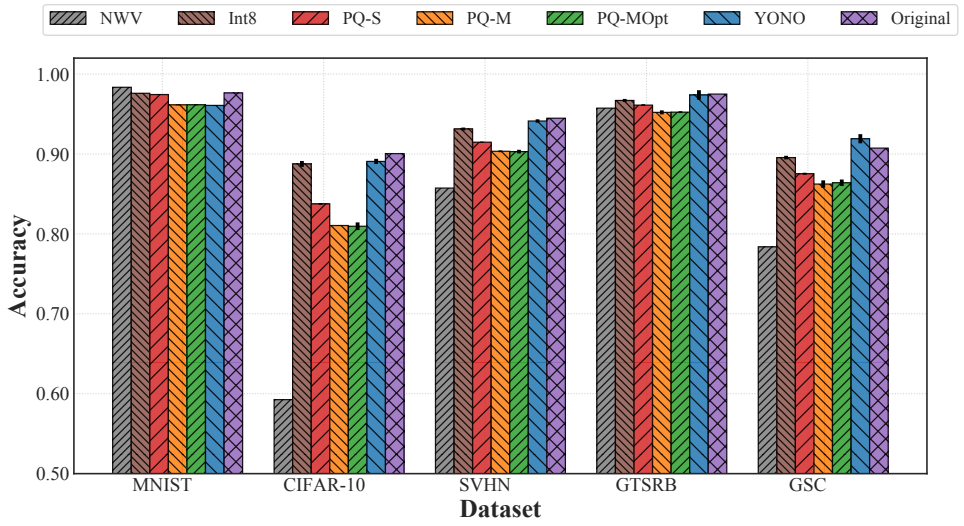


Figure 4.3: The inference accuracy of the heterogeneous MTL systems trained with five datasets of two modalities. Reported results are averaged over five trials, and standard-deviation intervals are depicted.

To begin with, the uncompressed (original) model serves as a performance upper bound. 8-bit quantisation and PQ-S achieve high accuracy close to that of the original model, showing a small average error rate of 0.9% and 2.8%, respectively, between each of the five models after compression and their corresponding original models. However, in the case of the CIFAR-10 dataset, PQ-S shows high error rates of 6.3% on average. This result indicates that the specialised codebooks which target only one model can help retain the performance of the original model in general but sometimes fail to retain it, as shown in the case of CIFAR-10. Besides, PQ-M shows an accuracy loss of 4.3% on average. For CIFAR-10, it shows a high error rate of 9.0%. In addition, although our proposed EM-based iterative network optimisation procedure can help in improving the accuracy, PQ-MOpt still shows a substantial accuracy drop of 4.3% on average. This result indicates that compressing multiple neural networks based on only one pair of codebooks is very challenging. However, YONO shows that its accuracy drop is minimal (i.e., an average error rate of 0.4%). Interestingly, in the case of GSC, YONO outperforms the accuracy of the original model by 1.2% where YONO benefits from sharing weights via PQ codebooks.

This result indicates that YONO can effectively retain the accuracy of original models as observed in the prior work on heterogeneous MTL systems [189] and other techniques focusing on a single model compression [48, 164, 245]. Further, it is an interesting result because YONO can retain the accuracy of multiple heterogeneous models, which is more challenging given that simply performing MTL would lead to the accuracy drop as shown in the prior work, NWV [189]. Also, note that differently from [189] which used LeNet,

Table 4.2: The compression efficiency and average accuracy of the heterogeneous MTL systems trained with five datasets of two modalities. Note that we use reported results in NWV by adjusting its compression rate from 4.04× to 8.08× as original models are based on 32-bit floats. The combined storage overhead for NWV's original models is 1.05 MB as it relies on the simplified LeNet architecture.

	NWV [189]	**Int8**	**PQ-S**	**PQ-M**	**PQ-MOpt**	**YONO**	**Original**
Ratio	8.08×	3.04×	9.47×	12.07×	12.07×	**11.57×**	1×
Size	0.13 MB	0.96 MB	0.31 MB	0.24 MB	0.24 MB	**0.25 MB**	2.91 MB
Accuracy	83.5%	93.2%	91.3%	89.8%	89.8%	**93.7%**	94.1%

we used optimised network architectures such as MicroNet and lightweight CNN that can execute on resource-constrained MCUs (refer to Section 4.4.5) and obtain very high accuracy. To name a few, the pretrained models in our work achieve 90.05%, 94.48%, 90.74% on CIFAR-10, SVHN, GSC compared to 59.26%, 85.74%, 78.38% reported in [189], respectively.

Compression Efficiency. Table 4.2 shows the overall efficiency in compressing heterogeneous networks trained with five datasets of two modalities. First, the combined storage overhead of the five uncompressed models is 2.91 MB which is three times the capacity of our target MCU's storage, which is 1 MB at maximum. However, considering that to perform an inference on MCUs, it is required to have a space for program codes of TFLM, input and output peripherals, input and output buffers, and other variables, etc., the space used to store models needs to be below the storage size of 1 MB. Thus, it is impossible to put those five models on an MCU and run multitask applications using the uncompressed models. 8-bit quantisation shows the lowest compression rate of 3.04× among all the evaluated methods, and its storage size (0.96 MB) is just below the limit of our employed MCU. Then, PQ-S shows a moderate compression rate of 9.47× and decreases the required storage size down to 0.31 MB and thus can reside on an MCU. Other baseline systems, PQ-M and PQ-MOpt, show a high compression rate of 12.07× and reduce the storage requirement to 0.24 MB. This is because PQ-M and PQ-MOpt share the same codebooks across the different applications. However, the savings in storage come at the expense of loss of accuracy, as seen in the accuracy results discussed before. In contrast, YONO achieves the best of both worlds, demonstrating a high compression rate close to PQ-M and PQ-MOpt and negligible accuracy loss compared to the uncompressed models. YONO obtains a 11.57× compression rate and decreases the storage overhead to 0.25 MB, showing a higher compression rate than NWV [189].

Overall, the results indicate that YONO can enable running multi-task applications on MCUs while retaining high accuracy and low storage footprints.

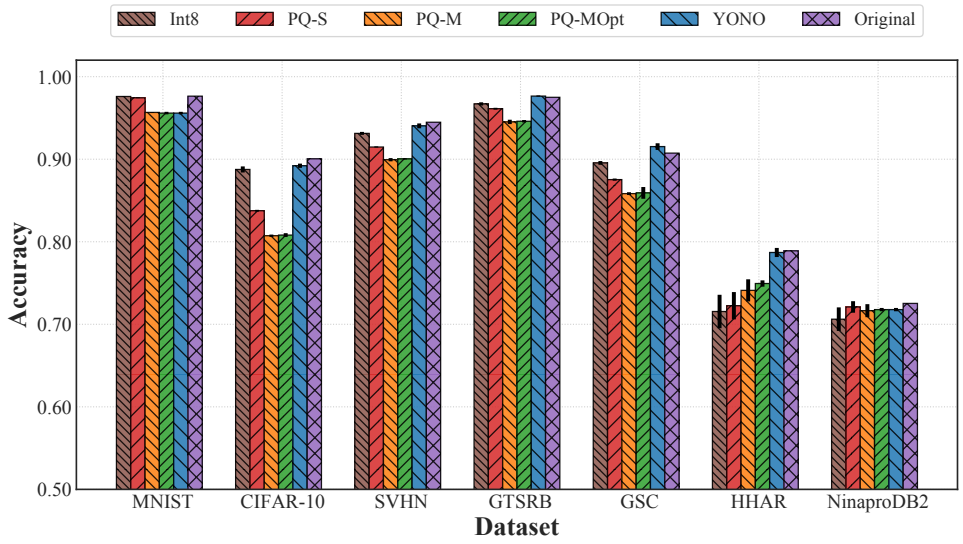


Figure 4.4: The inference accuracy of the heterogeneous MTL systems trained with seven datasets of four modalities. Reported results are averaged over five trials, and standard-deviation intervals are depicted.

4.4.3 Scalability

In this subsection, we apply YONO on seven datasets consisting of four different data modalities ((1) Image, (2) Audio, (3) IMU, (4) sEMG) to investigate to what extent our system can effectively compress multiple networks trained on different data modalities without losing its accuracy and compressive power. We select IMU and sEMG as additional modalities because they are also widely used in mobile sensing applications [10, 228].

Datasets. On top of the five datasets used in the previous subsection, we add two datasets of two additional modalities: HHAR [228] and Ninapro DB2 [234], corresponding to activity recognition (based on IMU) and gesture recognition (based on sEMG), respectively. The HHAR and Ninapro DB2 datasets are some of the most widely used HAR and sEMG datasets, respectively.

Model Architecture. To identify the right model architecture for each dataset, we adopt to use the optimised neural architectures and also conduct a hyper-parameter search as described in Section 4.4.2. Then, we select the model which shows the best performance. As a result, we use MicroNet-AD for HHAR and lightweight CNN architecture for Ninapro DB2 (see Table 4.1 for detail).

Accuracy. Figure 4.4 presents the accuracy results of the seven datasets of four modalities so that we examine the scalability of YONO to various modalities of data. Overall, the accuracy of reconstructed models from baseline systems and YONO is slightly improved

Table 4.3: The compression efficiency and average accuracy of the heterogeneous MTL systems trained with seven datasets of four modalities.

	Int8	**PQ-S**	**PQ-M**	**PQ-MOpt**	**YONO**	**Original**
Ratio	2.96×	9.27×	12.29×	12.29×	**11.77×**	1×
Size	1.27 MB	0.41 MB	0.31 MB	0.31 MB	**0.32 MB**	3.76 MB
Accuracy	86.8%	85.8%	84.6%	84.8%	**88.4%**	88.8%

since the error rates on the new datasets are smaller than those of the other five datasets. Also, accuracy results of baseline systems and YONO reach similar observations as in Section 4.4.2. 8-bit quantisation shows a small average error rate of 2.0% but a relatively high accuracy variance for HHAR. Also, PQ-S achieves high accuracy close to that of the uncompressed models with small error rates of 3.0% on average, whereas it shows high error in CIFAR-10. Then, the PQ-M and PQ-MOpt systems present an average error rate of 4.2% and 4.0% respectively, indicating that our proposed EM-based iterative network optimisation procedure help improve the accuracy but still falls short of achieving the original model's accuracy. Also, in this setting, YONO performs the best and shows an negligible accuracy loss of 0.5% on average.

Compression Efficiency. Table 4.3 shows the overall efficiency in compressing heterogeneous models trained with seven datasets of four modalities. Similar to the compression results in Section 4.4.2, the total size of the seven uncompressed models (3.76 MB) is larger than the storage budget for our target MCU. In the case of 8-bit quantisation, the required storage size of the seven compressed models is 1.27 MB, larger than our storage budget of 1 MB. This result indicates that 8-bit quantisation is not suitable for operating many heterogeneous neural networks simultaneously on our target MCU. However, YONO requires at most 0.32 MB. Since our system can effectively compress multiple heterogeneous models (showing 11.77× compression ratio), the incurred storage requirement is minimal. For example, when two additional models (for HHAR and Ninapro DB2) are included in an MTL system, YONO incurs only 0.07 MB additional overhead, whereas the original models' storage size increases by 0.85 MB.

To summarise, our results show that YONO is scalable as it can accommodate many applications utilizing different input modalities while achieving high performance and small storage overhead.

4.4.4 Generalisability

We now investigate the generalisability of our multitasking system on new models/datasets and different network architectures unseen during the codebook learning phase of the offline component. Specifically, we evaluate whether YONO can achieve high accuracy on

Table 4.4: Summary of datasets, model architectures, mobile applications used in Section 4.4.4.

Modality	Dataset	Architecture	Mobile Application
Image	FashionMNIST	DS-CNN	Object recognition
	STL-10	DS-CNN	Object recognition
Audio	EmotionSense	Lightweight CNN	Emotion recognition
	UrbanSound	DS-CNN	Sound classification
IMU	PAMAP2	MicroNet-AD	Activity recognition
	Skoda	MicroNet-AD	Activity recognition
sEMG	Ninapro DB3	Lightweight CNN	Gesture recognition
	Ninapro DB6	MicroNet-AD	Gesture recognition

the unseen models from new datasets using the same codebooks that are learned previously (Section 4.4.3). This can be particularly useful since the learned codebooks of YONO can still be utilised to compress unseen models in different network architectures from new datasets without learning new codebooks again whenever a user wants to incorporate a new task/dataset into the system. Also, note that since the codebooks are not modified, the reported results in Section 4.4.3 are not affected, ensuring high accuracy on previous datasets. Then, in Section 4.4.4, we select two new datasets in each of the four modalities for a robust evaluation.

Datasets. In total, we add eight new datasets: two image datasets (1) FashionMNIST [254], (2) STL-10 [255], and two audio datasets (3) EmotionSense [15], (4) UrbanSound [240], and two HAR datasets (5) PAMAP2 [230], (6) Skoda [231], and lastly two sEMG datasets (7) Ninapro DB3 [234] and (8) Ninapro DB6 [256]. These are widely used real-world application datasets corresponding to classification problem as follows: (1) ten fashion items, (2) ten generic objects, (3) five emotions, (4) ten environmental sounds, (5) 12 activities, (6) ten activities, (7) ten gestures of amputees, (8) seven gestures of ordinary people, respectively.

Model Architecture. To demonstrate that YONO can effectively address new network architectures that were not shown during the offline codebook learning phase, we include another widely used architecture, DS-CNN [98], in our work. Then, we follow the same hyper-parameter search process as described in Section 4.4.2. Table 4.4 summarises the identified network architectures for each dataset and its associated mobile application.

Accuracy. Note that we exclude PQ-S as it needs to learn PQ codebooks on a given dataset and then perform network finetuning on the given dataset. However, in this scenario, the system needs to adapt to new (unseen) datasets. This point makes the

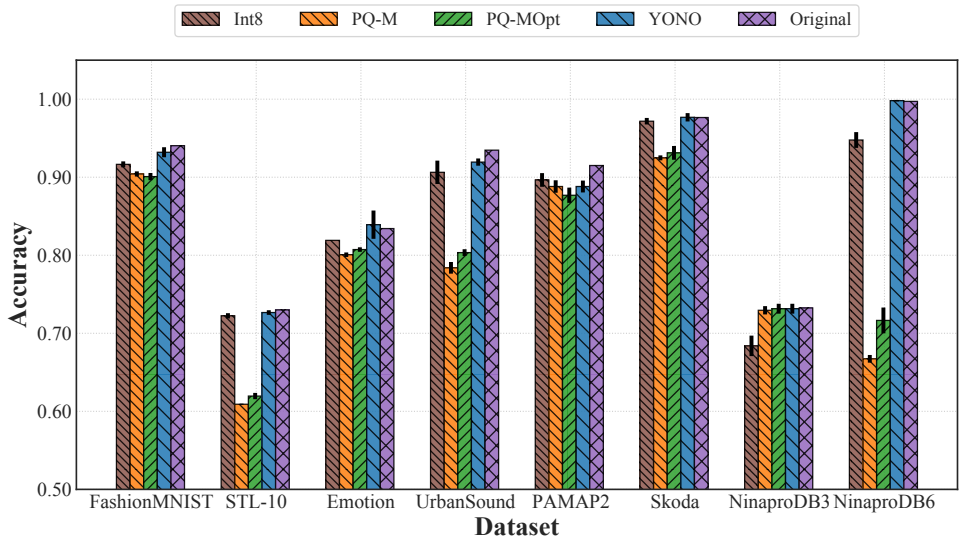


Figure 4.5: The inference accuracy of the heterogeneous MTL systems applied to unseen datasets of four modalities. Reported results are averaged over five trials, and standard-deviation intervals are depicted.

scenario particularly challenging since an MTL system needs to incorporate unseen datasets and network architectures. Nonetheless, an MTL system that can address this challenge could become very useful in practice since it is adaptable.

To begin with, Figure 4.5 shows the accuracy results on the eight unseen datasets with diverse network architectures. 8-bit quantisation presents a moderate error rate of 2.5% similar to the results in Section 4.4.2 and Section 4.4.3 as the current evaluation setup does not make a difference for the single model compression approach. Conversely, PQ-M shows a substantial accuracy drop (9.4%) compared to the original model, which is worse than the previous two scenarios where it obtained error rates of 4.3% and 4.2%. In fact, on one dataset (Ninapro DB6), PQ-M shows a 33.0% error rate. Although PQ-MOpt improves upon PQ-M, the amount of improvement is small. PQ-MOpt shows a 8.4% accuracy drop on average compared to the original model. Also, for Ninapro DB6, the accuracy of PQ-MOpt shows a sharp decrease of 28.1% compared to the original model, demonstrating the difficulty of this scenario. Surprisingly, however, YONO does not experience a considerable accuracy loss. It shows only 0.6% accuracy loss on average. Besides, YONO shows a low variance of accuracy loss across the employed datasets. In fact, YONO even improves upon the accuracy of uncompressed models for some datasets such as EmotionSense, Skoda, and Ninapro DB6. These results highlights that YONO is capable of retaining the accuracy of original models even in the most challenging scenario of incorporating unseen datasets and architectures.

Table 4.5: The compression efficiency and average accuracy of the heterogeneous MTL systems applied to unseen datasets of four modalities.

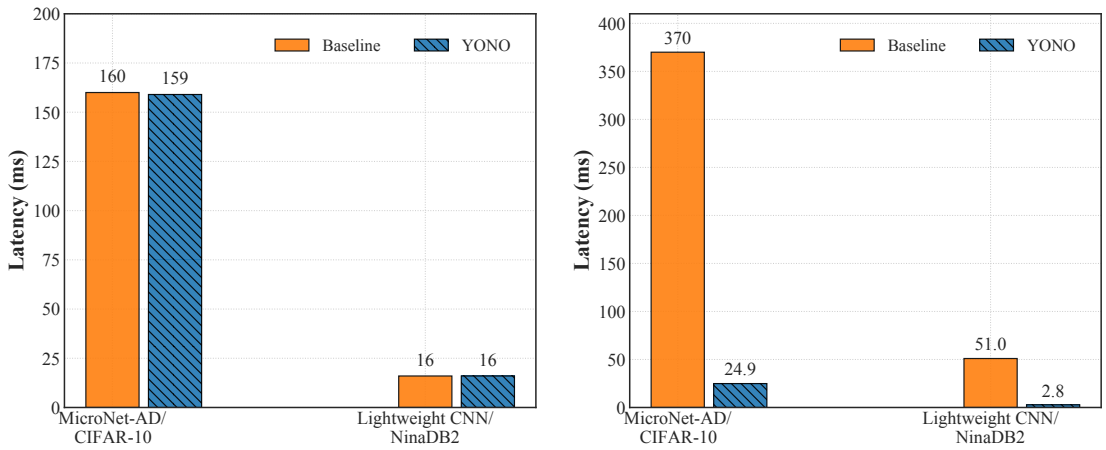
	Int8	**PQ-M**	**PQ-MOpt**	**YONO**	**Original**
Ratio	2.80×	13.60×	13.60×	**12.37**×	1×
Size	1.47 MB	0.30 MB	0.30 MB	**0.33 MB**	4.11 MB
Accuracy	85.8%	78.8%	79.8%	**87.6%**	88.3%

Compression Efficiency. The compression results for heterogeneous models with eight unseen datasets are shown in Table 4.5. The size of the uncompressed models is the largest, 4.11 MB, in this setup compared to Section 4.4.2 and Section 4.4.3. YONO shows an impressive compression ratio of 12.37× and require storage size of 0.33 MB after compressing eight heterogeneous networks. It is worth noting that we included a new network architecture that YONO did not learn during its offline codebook learning phase. Yet, YONO successfully compress different architectures with an even higher compression rate (11.57× in Section 4.4.2 and 11.77× in Section 4.4.3) without loss of accuracy on all the unseen datasets.

In summary, the results here hint that YONO can effectively compress different heterogeneous models trained on unseen datasets without losing accuracy and demonstrate the generalisability of YONO's codebooks and the effectiveness of the proposed network optimisation and optimisation heuristics.

4.4.5 Evaluation on In-Memory Execution and Model Swapping Framework on MCUs

We finally examine the run-time performance of the online component of YONO, the in-memory execution and model swapping framework, introduced in Section 4.2.6. In specific, we evaluate the latency and energy consumption of model execution and model swapping of YONO on an MCU. Also, we include an alternative approach to YONO as a baseline that relies on an external SD card as a secondary storage device for storing heterogeneous networks and on in-memory execution similar to YONO. We employ the same datasets used in the previous subsections. In Figures 4.6 and 4.7, we report the results of upper bound (i.e., slowest or the most energy-consuming) and lower bound (i.e., fastest or the least energy-consuming) to show the range of latency and energy consumption of YONO and the baseline based on the identified network architectures trained on the datasets in Section 4.4.2-Section 4.4.4 (see Tables 4.1 and 4.4). We use a MicroNet-AD model based on CIFAR-10 as upper bound and a lightweight CNN model based on Ninapro DB2 as lower bound. Although results for other models and datasets are omitted, they reside within the reported latency and energy consumption as in Figures 4.6 and 4.7.



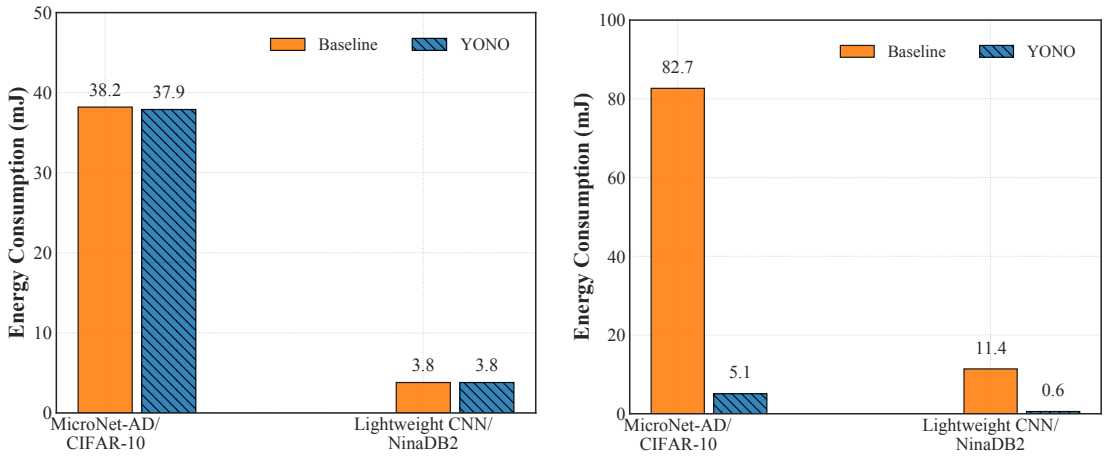
(a) Execution (b) Loading/Switching

Figure 4.6: The model execution and loading/switching time of YONO and the baseline.

Latency. We measure the latency of the model execution and model loading/swap by using MBed Timer API, as shown in Figure 4.6. In terms of execution time, both YONO and the baseline show a swift execution time (16-160 ms per inference) that can be useful in practice, and there is no meaningful latency difference between them since both rely on in-memory execution. However, for model loading/swap time, YONO accelerates the model switching. YONO reduces model loading/swap time by 93.3% (370 ms vs. 24.9 ms) in a MicroNet-AD model based on CIFAR-10 and 94.5% (51.0 ms vs. 2.8 ms) in a lightweight CNN model based on Ninapro DB2 compared to the baseline. Note that we did not conduct a direct comparison on-device with the prior work [189] since its source code is not shared and the used MCUs for experiments are not the same.

Energy Consumption. We measure the energy consumption of model execution and loading/swap on the MCU using YONO and the baseline, as shown in Figure 4.7. We use the Tenma 72-7720 digital multimeter to measure the power consumption and then compute the energy consumption over time taken for each operation (i.e., inference and model loading). Similar to the latency result, the energy consumption for executing models does not show the difference as explained above. However, for the model loading/swap task, YONO decreases energy consumption by at minimum 93.9% (82.7 mJ vs. 5.1 mJ in a MicroNet-AD model on CIFAR-10) and at maximum 95.0% (11.4 mJ vs. 0.6 mJ in a lightweight CNN model on Ninapro DB2) compared to the baseline.

To summarise, the results demonstrate that YONO enables fast (low latency) and efficient (low energy footprints) model execution and loading/swap on an extremely resource-limited IoT device, MCU.



(a) Execution (b) Loading/Switching

Figure 4.7: The energy consumption of model execution and loading/switching of YONO and the baseline.

4.5 Discussion

Impact on Heterogeneous MTL Systems. YONO represents the first framework that can compress multiple heterogeneous models and be applicable to unseen datasets. Also, YONO ensures negligible or no loss of accuracy in compressing many different models (architecture) on multiple datasets. This is achieved by only one pair of PQ-based codebooks, our novel optimisation procedure, and heuristics. Thus, we envisage that YONO could become a practical system to deploy heterogeneous MTL systems on various embedded devices and platforms in many real-world applications in the future. We leave the wide deployment and performance evaluation of YONO on other embedded platforms under real-world application scenarios as future work.

Application Scenario. Let us consider an example of a real-world application. Given an intelligent authentication system for a smart home, the system would need to detect tenants' identification based on images and voice (image classification and voice recognition). Then, the system could take voice commands as inputs from the identified tenant (e.g., keyword spotting). This simple application scenario already needs three different models, which could satisfy the necessity of a heterogeneous MTL system, YONO.

Generalisability of YONO. In Section 4.4, we have demonstrated that YONO can incorporate heterogeneous models and datasets (four different modalities) consisting of 15 datasets (i.e., seven datasets for learning codebooks in Section 4.4.3 and the other eight unseen datasets in Section 4.4.4), which shows that YONO is a generalisable framework. Other datasets and network architectures (e.g., LSTMs [8] and CNNs with large-sized

kernels like 5x5 or 7x7) that can be employed and tested on YONO are left as future work.

Limitation. To enable model switching during the runtime, we design YONO to load the model in the main memory instead of the storage of an MCU. However, since SRAM is a limited on-chip resource and typically smaller than eFlash, our design choice may limit the applicability of YONO, especially for low-end MCUs with smaller SRAM sizes such as 128 KB. Therefore, it would be worthwhile to further investigate memory-efficient ways to reduce the required main memory space for model execution while enabling the model switching at run time. Better usage of FlatBuffer serialisation format to hold model weights can be interesting future work since the weights of a model takes the majority of the space.

4.6 Conclusions

In this chapter, we have presented an efficient MTL system, YONO, that compresses multiple heterogeneous models through PQ codebooks, our novel network optimisation and heuristics. First, we implemented YONO's offline component on a server and its online component on a critically resource-constrained MCU. Then, we demonstrated its effectiveness and efficiency. YONO compresses multiple heterogeneous models up to 12.37× with minimal or near to no accuracy loss. Interestingly, YONO can successfully compress models trained with datasets unseen during its offline codebook learning phase. Finally, YONO's online component enables an efficient in-memory model execution and loading/swap with low latency and energy footprints on an MCU. We envision that methods developed for YONO and our research findings could pave the way to deploy practical heterogeneous multi-task deep learning systems on various embedded devices in the near future.

Chapter 5

Bringing On-Device ML from Edge to Microcontrollers: TinyTrain

5.1 Introduction

In the previous chapter, we have designed and developed the ML system tailored for MCUs with extreme resource constraints by enabling efficient execution of different ML models for real-world applications supporting multiple tasks. In this chapter, we propose the joint optimisation of data, memory, and computation to enable on-device training on (extremely) resource-constrained devices. Specifically, on-device training of DNNs on edge devices has the potential to enable diverse real-world applications to *dynamically adapt* to new tasks [29] and different (*i.e.* cross-domain/out-of-domain) data distributions from users (*e.g.* personalisation) [18], without jeopardising privacy over sensitive data (*e.g.* healthcare) [22].

Despite its benefits, several challenges hinder the broader adoption of on-device training (refer to Chapter 1 for more details). **Firstly,** labelled user data are neither abundant nor readily available in real-world IoT applications. **Secondly,** edge devices are often characterised by severely limited memory. With the *forward* and *backward passes* of DNN training being significantly memory-hungry, there is a mismatch between memory requirements and memory availability at the edge. Even architectures tailored to microcontroller units (MCUs), such as MCUNet [251], require almost 1 GB of peak training-time memory (see Table 5.2), which far exceeds the RAM size of widely used embedded devices, such as Raspberry Pi Zero 2 (512 MB), and commodity MCUs (1 MB). **Lastly,** on-device training is limited by the constrained processing capabilities of edge devices, with training requiring at least 3× more computation (*i.e.* multiply-accumulate (MAC) count) than inference [24]. This places an excessive burden on tiny edge devices that host less powerful processors,

compared to server-grade CPUs and GPUs [44].

Despite the growing effort towards on-device training, the current methods have important limitations as we discussed thoroughly in Section 2.4.

1. The common approach of *fine-tuning only the last layer* [37, 38] leads to considerable accuracy loss (≥10%) that far exceeds the typical drop tolerance.
2. *Recomputation*-based memory-saving techniques [22, 41, 42, 43] that trade-off more operations for lower memory usage incur significant computation overhead, further aggravating the already excessive on-device training time.
3. *Sparse-update* methods [44, 215, 216, 217, 218] selectively update only a subset of layers (and channels) during on-device training, reducing both memory and computation loads. Nonetheless, as shown in Section 5.3.2, these approaches show either drastic accuracy drops up to 7.7% for *SparseUpdate* [44] or small memory/computation reduction of 1.5-3× for *p-Meta* [217] and *TinyTL* [215] over fine-tuning the entire DNN when applied at the edge where data availability is low. Additionally, these methods require running *a few thousands of* computationally heavy searches [44], pruning processes [216], or pre-selecting layers to be updated [218] on powerful GPUs to identify important layers/channels for each target dataset during the offline stage before deployment, and they are hence unable to dynamically adapt to the characteristics of user data.

To address the aforementioned challenges and limitations, we present *TinyTrain*, the first approach that fully enables compute-, memory-, and data-efficient on-device training on constrained edge devices. TinyTrain departs from the static configuration of the sparse-update policy, *i.e.* with the subset of layers and channels to be fine-tuned remaining fixed, and proposes *task-adaptive sparse update*. Our task-adaptive sparse update requires running *only once* for each target dataset and can be efficiently executed on resource-constrained edge devices. This enables us to adapt the layer/channel selection in a task-adaptive manner, leading to better on-device adaptation and higher accuracy.

Specifically, we introduce a novel resource-aware *multi-objective criterion* that captures both the importance of channels and their computational and memory cost to guide the layer/channel selection process. Then, at run time, we propose *dynamic layer/channel selection* that dynamically adapts the sparse update policy using our multi-objective criterion. Considering both the properties of user data, and the memory and processing capacity of the target device, TinyTrain enables on-device training with a significant reduction in memory and computation while ensuring high accuracy over the SOTA [44].

To further address the drawbacks of data scarcity, TinyTrain enhances the conventional on-device training pipeline by means of a few-shot learning (FSL) pre-training scheme; this step meta-learns a reasonable global representation that allows on-device training to

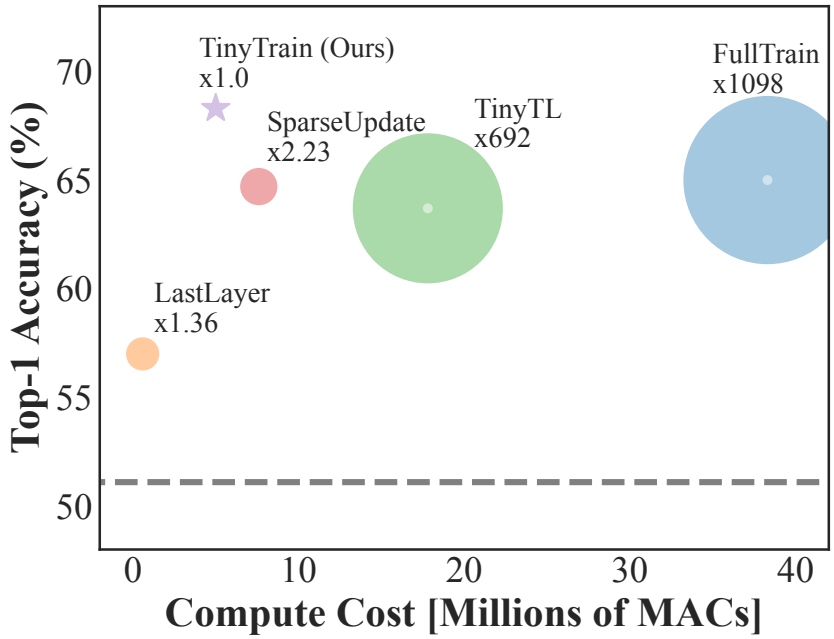


Figure 5.1: Cross-domain accuracy (y-axis) and compute cost in MAC count (x-axis) of TinyTrain and existing methods, targeting ProxylessNASNet on Meta-Dataset. The radius of the circles and the corresponding text denote the increase in the memory footprint of each baseline over TinyTrain. The dotted line represents the accuracy without on-device training.

be sample-efficient and reach high accuracy despite limited and cross-domain target data.

Figure 5.1 presents a comparison of our method's performance with existing on-device training approaches. TinyTrain achieves the highest accuracy, with gains of 3.6-5.0% over fine-tuning the entire DNN, denoted by *FullTrain*. On the compute front, TinyTrain significantly reduces the memory footprint and computation required for backward pass by up to 1,098× and 7.68×, respectively. TinyTrain further outperforms the SOTA *SparseUpdate* method in all aspects, yielding: (a) 2.6-7.7% accuracy gain across nine datasets; (b) 1.59-2.23× reduction in memory; and (c) 1.52-1.82× lower computation costs.

Finally, we demonstrate how our work makes important steps towards efficient training on highly constrained edge devices by deploying TinyTrain on Raspberry Pi Zero 2 and Jetson Nano. We show that our multi-objective criterion can be efficiently computed within 20-35 seconds on both target edge devices (*i.e.* 3.4-3.8% of the total training time of TinyTrain), eliminating the need of an expensive offline search process for layers/channel selection. Also, TinyTrain achieves end-to-end on-device training in 10 minutes, an order of magnitude speedup over the two-hour training of FullTrain on Pi Zero 2.

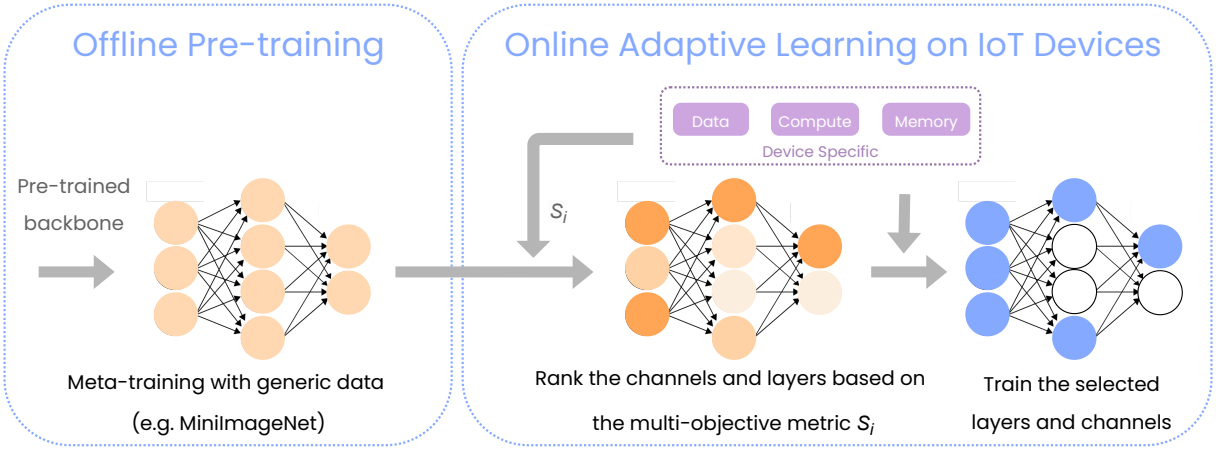


Figure 5.2: Overview of TinyTrain. It consists of (1) the offline pre-training and (2) the online adaptive learning stages. In (1), TinyTrain pre-trains and meta-trains DNNs to improve the attainable accuracy when only a few data are available for adaptation. Then, in (2), TinyTrain performs task-adaptive sparse update based on the multi-objective criterion and dynamic layer/channel selection that co-optimises both memory and computations.

This work paves the way, for the first time, to performing on-device training with acceptable performance on a variety of resource-constrained devices, such as MCU-grade IoT frameworks.

5.2 Methodology

Problem Formulation. From a learning perspective, on-device DNN training at the data-scarce edge imposes unique characteristics that the model needs to address during deployment, primarily: (1) unseen target tasks with different data distributions (cross-domain), (2) scarce labelled user data (Section 5.2.1), and (3) minimisation of compute and memory resource consumption (Section 5.2.2). To formally capture this setting, in this work, we cast it as a cross-domain few-shot learning (CDFSL) problem. In particular, we formulate it as *K-way-N-shot* learning [138] which allows us to accommodate more general scenarios instead of optimising towards one specific CDFSL setup (*e.g.* 5-way 5-shots). This formulation requires us to learn a DNN for K classes given N samples per class. To further push towards realistic scenarios, we learn *one* global DNN representation from various K and N, which can be used to learn novel tasks (see Section 5.3.1 for details).

Our Pipeline. Figure 5.2 shows the processing flow of TinyTrain comprising two stages. The first stage is *offline learning* (Section 5.2.1). By means of pre-training and meta-training, TinyTrain aims to find an informed weight initialisation, such that subsequently the model can be rapidly adapted to the user data with only a few samples

(5-30), drastically reducing the burden of manual labelling and the overall training time compared to state of the art methods. The second stage is *online learning* (Section 5.2.2). This stage takes place on the target edge device, where TinyTrain utilises its task-adaptive sparse-update method to selectively fine-tune the model using the limited user-specific, cross-domain target data, while minimising the memory and compute overhead.

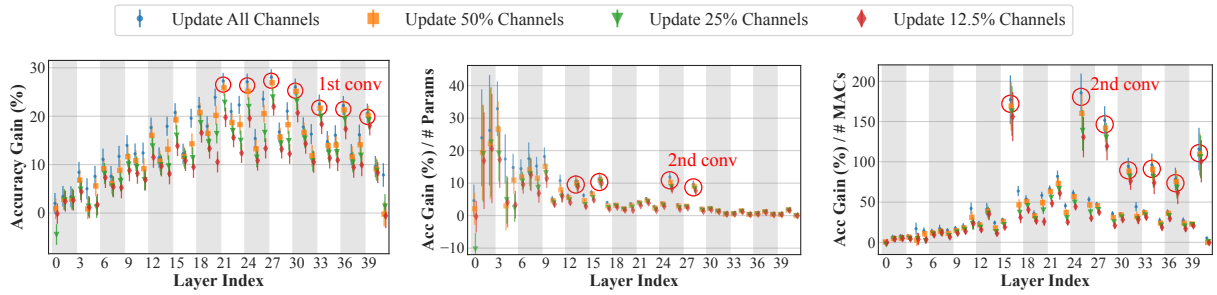
5.2.1 Few-Shot Learning-Based Pre-training

The vast majority of existing on-device training pipelines optimise certain aspects of the system (*i.e.* memory or compute) via memory-saving techniques [22, 41, 42, 43] or fine-tuning a small set of layers/channels [37, 38, 44, 215, 216]. However, these methods neglect the aspect of sample efficiency in the low-data regime of tiny edge devices. As the availability of labelled data is severely limited at the edge, existing on-device training approaches suffer from insufficient learning capabilities under such conditions.

In our work, we depart from the transfer-learning paradigm (*i.e.* DNN pre-training on source data, followed by fine-tuning on target data) of existing on-device training methods that are unsuitable to the very low data regime of edge devices. Building upon the insight of recent studies [36] that transfer learning does not reach a model's maximum capacity on unseen tasks in the presence of only limited labelled data, we augment the *offline* stage of our training pipeline as follows. Starting from the *pre-training* of the DNN backbone using a large-scale public dataset, we introduce a subsequent *meta-training* process that meta-trains the pre-trained DNN given only a few samples (5-30) per class on simulated tasks in an episodic fashion. As shown in Section 5.3.3, this approach enables the resulting DNNs to perform more robustly and achieve higher accuracy when adapted to a target task despite the low number of examples, matching the needs of tiny edge devices. As a result, our few-shot learning (FSL)-based pre-training constitutes an important component to improve the accuracy given only a few samples for adaptation, reducing the training time while improving data and computation efficiency. Thus, TinyTrain alleviates the drawbacks of current work, by explicitly addressing the lack of labelled user data, and achieving faster training and lower accuracy loss.

Pre-training. For the backbones of our models, we employ feature extractors of different DNN architectures as in Section 5.3.1. These feature backbones are pre-trained with a large-scale image dataset, *e.g.* ImageNet [257].

Meta-training. For the meta-training phase, we employ the metric-based ProtoNet [133], which has been demonstrated to be simple and effective as an FSL method. ProtoNet computes the class centroids (*i.e.* prototypes) for a given support set and then performs nearest-centroid classification using the query set. Specifically, given a pre-trained feature backbone f that maps inputs x to an m-dimensional feature space, ProtoNet first computes the prototypes c_k for each class k on the support set as $c_k = \frac{1}{N_k} \sum_{i:y_i=k} f(x_i)$, where



(a) Accuracy Gain (b) Accuracy Gain / Params (c) Accuracy Gain / MACs

Figure 5.3: Memory- and compute-aware analysis of MCUNet by updating four different channel ratios on each layer. (a) Accuracy gain per layer is generally highest on the first layer of each block. (b) Accuracy gain per parameter of each layer is higher on the second layer of each block. (c) Accuracy gain per MACs of each layer has peaked on the second layer of each block. These observations show accuracy, memory footprint, and computes in a trade-off relation.

$N_k = \sum_{i:y_i=k} 1$ and y are the labels. The probability of query set inputs x for each class k is then computed as:

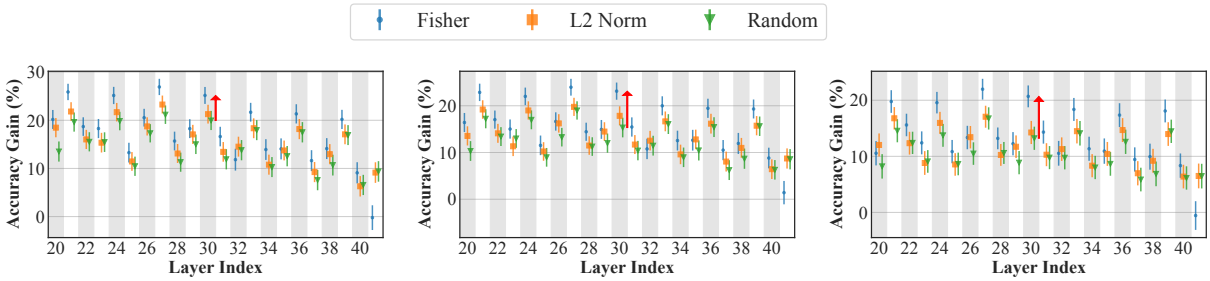
$$p(y = k|x) = \frac{exp(-d(f(x), c_k))}{\sum_j exp(-d(f(x), c_j))} \tag{5.1}$$

We use cosine distance as the distance measure d similarly to Hu et al. [36]. Note that ProtoNet enables the *various-way-various-shot* setting since the prototypes can be computed regardless of the number of ways and shots. The feature backbones are meta-trained with MiniImageNet [2], a commonly used source dataset in CSFSL, to provide a weight initialisation generalisable to multiple downstream tasks in the subsequent online stage (see Section 5.3.3 for meta-training cost analysis).

5.2.2 Task-Adaptive Sparse Update

Existing FSL pipelines generally focus on data and sample efficiency and attend less to system optimisation [36, 129, 130, 133, 138], rendering most of these algorithms undeployable for the edge, due to high computational and memory costs. In this context, sparse update [44, 216], which dictates that only a subset of essential layers and channels are to be trained, has emerged as a promising paradigm for making training feasible on resource-constrained devices.

Two key design decisions of sparse-update methods are *i)* the scheme for determining the *sparse-update policy*, *i.e.* which layers/channels should be fine-tuned, and *ii)* how often should the sparse-update policy be modified. In this context, a SOTA method, such as *SparseUpdate* [44], is characterised by important limitations. First, it casts the



(a) 50% channels selected (b) 25% channels selected (c) 12.5% channels selected

Figure 5.4: The pairwise comparison between our dynamic channel selection and static channel selections (*i.e.* Random and L2-Norm) on MCUNet. The dynamic channel selection consistently outperforms static channel selections as the accuracy gain per layer differs by up to 8%. Also, the gap between dynamic and static channel selections increases as fewer channels are selected for updates.

layer/channel selection as an optimisation problem that aims to maximise the accuracy gain subject to the memory constraints of the target device. However, as the optimisation problem is combinatorial, *SparseUpdate* solves it offline by means of a heuristic evolutionary algorithm that requires *a few thousand trials*. Second, as the search process for a good sparse-update policy is too costly, it is practically infeasible to dynamically adjust the sparse-update policy whenever new target datasets are given, leading to performance degradation.

Multi-Objective Criterion. With resource constraints being at the forefront in tiny edge devices, we investigate the trade-offs among accuracy gain, compute and memory cost. To this end, we analyse each layer's contribution (*i.e. accuracy gain*) on the target dataset by updating a single layer at a time, together with cost-normalised metrics, including *accuracy gain per parameter* and *per MAC operation* (*i.e.* Accuracy gain divided by the number of parameters and MACs of each layer). Figure 5.3 shows the results of MCUNet [251] on the Traffic Sign [258] dataset. We make the following observations: (1) the peak point of accuracy gain occurs at the first layer of each block (pointwise convolutional layer) (Figure 5.3a), (2) the accuracy gain per parameter and computation cost occurs at the second layer of each block (depthwise convolutional layer) (Figures 5.3b and 5.3c). These findings indicate a non-trivial trade-off between accuracy, memory, and computation, demonstrating the necessity for an effective and resource-aware layer/channel selection for on-device training that jointly considers all the aspects.

To encompass both accuracy and efficiency aspects, we design a multi-objective criterion for the layer selection process of our task-adaptive sparse-update method. To quantify the importance of channels and layers on the fly, we propose the use of Fisher information on activations [39, 259, 260], often used to identify *less important* channels/layers for

pruning [259]. In addition, Turner et al. [261] demonstrated that the summation of the Fisher information on channel activations for a whole block (consisting of several layers) is a useful metric in identifying effective blocks in architecture search, whereas we use it as a proxy for identifying *with fine granularity the more important* layers/channels for weight update. Formally, given N examples of target inputs, the Fisher information Δ_o can be calculated after backpropagating the loss L with respect to activations a of a layer:

$$\Delta_o = \frac{1}{2N} \sum_{n}^{N} (\sum_{d}^{D} a_{nd} g_{nd})^2 \tag{5.2}$$

where gradient is denoted by g_{nd} and D is feature dimension of each channel (*e.g.* $D = H \times W$ of height H and width W). We obtain the Fisher potential P for a whole layer by summing Δ_o for all activation channels as: $P = \sum_o \Delta_o$.

Having established the importance of channels in each layer, we define a new multi-objective metric s that jointly captures importance, memory footprint and computational cost:

$$s_i = \frac{P_i}{\frac{\|W_i\|}{\max\limits_{l \in \mathcal{L}}(\|W_l\|)} \times \frac{M_i}{\max\limits_{l \in \mathcal{L}}(M_l)}} \tag{5.3}$$

where $\|W_i\|$ and M_i represent the number of parameters and multiply-accumulate (MAC) operations of the i-th layer and are normalised by the respective maximum values $\max\limits_{l \in \mathcal{L}}(\|W_l\|)$ and $\max\limits_{l \in \mathcal{L}}(M_l)$ across all layers $\mathcal{L}$ of the model. This multi-objective metric enables TinyTrain to rank different layers and prioritise the ones with higher Fisher potential per parameter and per MAC during layer selection. Further, since TinyTrain can obtain multi-objective metric efficiently by calculating the Fisher potential *only once* for each target dataset as detailed below, TinyTrain effectively alleviates the burdens of running the computationally heavy search processes *a few thousand times*.

Dynamic Layer/Channel Selection. We now present our *dynamic layer/channel selection* scheme, the second component of our task-adaptive sparse update, that runs at the *online* learning stage (*i.e.* deployment and meta-testing phase). Concretely, with reference to Algorithm 4, when a new on-device task needs to be learned (*e.g.* a new user or dataset), the sparse-update policy is modified to match its properties (lines 1-4). Contrary to the existing layer/channel selection approaches that remain fixed across tasks, our method is based on the key insight that different features/channels can play a more important role depending on the target dataset/task/user. As shown in Section 5.3.3, effectively tailoring the layer/channel selection to each task leads to superior accuracy compared to the pre-determined, static layer selection scheme of *SparseUpdate*, while further minimising system overheads.

As an initialisation step, TinyTrain is first provided with the memory and computation budget determined by hardware and users, *e.g.* around 1 MB and 15% of total MACs can be given as backward-pass memory and computational budget. Then, we calculate the Fisher potential for each convolutional layer using the given inputs of a target task (lines 1-2). Then, based on our multi-objective criterion (Eq. (5.3)) (line 3), we score each layer and progressively select as many layers as possible without violating the memory constraints (imposed by the memory usage of the model, optimiser, and activations memory) and resource budgets (imposed by users and target hardware) on an edge device (line 4). Formally, our dynamic layer selection aims to find layer indices i that optimise the following:

$$\begin{gathered}\max|\mathcal{L}_{\text{sel}}|, \quad \text{where} \quad \mathcal{L}_{\text{sel}} \subset \mathcal{L} \\ \text{s.t.} \quad s_i \geq s_j \qquad \forall i \in \mathcal{L}_{\text{sel}} \quad \forall j \in \mathcal{L} \\ \text{MemoryCost}(\mathcal{L}_{\text{sel}}) \leq B_{\text{mem}}, \\ \text{ComputeCost}(\mathcal{L}_{\text{sel}}) \leq B_{\text{compute}}\end{gathered}$$

where $\mathcal{L}$ is the set of all layers in the target neural network, $\mathcal{L}_{\text{sel}}$ is the set of selected layers, s_i is the value of our multi-objective metric (Section 5.2.2) for the i-th layer, and B_{compute} and B_{mem} are the compute and memory budgets, respectively, also shown in Algorithm 4. The overall objective is to find the maximum number of layer indices i with the highest multi-objective score s_i with respecting the compute and memory constraints.

After having selected layers, within each selected layer, we identify the top-K most important channels to update. Formally, our dynamic channel selection aims to find indices c for each layer $i \in \mathcal{L}_{\text{sel}}$ that optimise the following:

$$\begin{aligned}&\max_{\mathcal{C}_{i,\text{sel}} \subset \mathcal{C}_i} \sum_{c \in \mathcal{C}_{i,\text{sel}}} \Delta_{o,c} \\ &\text{s.t.} \quad |\mathcal{C}_{i,\text{sel}}| = K\end{aligned}$$

where $\mathcal{C}_i$ is the set of channel indices for the i-th layer, $\mathcal{C}_{i,\text{sel}}$ is the set of selected channels, $\Delta_{o,c}$ is the Fisher information for the c-th channel that was precomputed during the initialisation step (line 4). The overall objective is, for each selected layer $i \in \mathcal{L}_{\text{sel}}$, to find the top-$K$ channels with the highest Fisher information. Note that the overhead of our dynamic layer/channel selection is minimal, which takes only 20-35 seconds on edge devices (refer to Sections 5.3.2 and 5.3.3 for more analysis). Having finalised the layer/channel selection, we proceed with their sparse fine-tuning of the meta-trained DNN during meta-testing (see below for detailed procedures). As in Figure 5.4 (MCUNet on Traffic Sign), dynamically identifying important channels for an update for each target task outperforms the static channel selections such as random- and L2-Norm-based selection.

Algorithm 4: Online learning stage of TinyTrain

Input: Meta-trained backbone weights W, Iterations k, Train data D_{train}, Test data D_{test}, Memory and compute budgets B_{mem}, B_{compute}

/* - - - *Dynamic Layer / Channel Selection* - - - */
Compute the gradient using the given samples D_{train}
Compute the Fisher potential using Eq. (5.2) from the Fisher information
Compute our multi-objective metric s using Eq. (5.3)
Perform the dynamic layer & channel selection using $\{W, s, B_{\text{mem}}, B_{\text{compute}}\}$

/* - - - *Perform sparse fine-tuning* - - - */
for $t = 1, ..., k$ **do**
    Update the selected layers/channels using D_{train}
Evaluate the fine-tuned backbone using D_{test}

Fine-tuning Procedure during Meta-Testing Our fine-tuning procedure during the meta-testing phase is similar to that of [36, 137, 262]. First of all, as the support set is the only labelled data during meta-testing, prior works [137, 262] fine-tune the models using only the support set. For [36], it first uses data augmentation with the given support set to create a pseudo query set. After that, it uses the support set to generate prototypes and the pseudo query set to perform backpropagation using Eq. 5.1. Differently from [137, 262], the fine-tuning procedure of [36] does not need to compute prototypes and gradients using the same support set using Eq. 5.1. However, Hu et al. [36] simply fine-tune the entire DNNs without memory-and compute-efficient on-device training techniques, which becomes one of our baselines, FullTrain requiring prohibitively large memory footprint and computation costs to be done on-device during deployment. In our work, for all the on-device training methods including TinyTrain, we adopt the fine-tuning procedure introduced in [36]. However, we extend the vanilla fine-tuning procedure with existing on-device training methods (*i.e.* LastLayer, TinyTL, SparseUpdate, which serve as the baselines of on-device training in our work) so as to improve the efficiency of on-device training on the extremely resource-constrained devices.

Overall, our task-adaptive sparse update facilitates TinyTrain to achieve superior accuracy, while further minimising the memory and computation cost by co-optimising both system constraints, thereby enabling memory- and compute-efficient training at the data-scarce edge.

Table 5.1: Top-1 accuracy results of TinyTrain and the baselines. TinyTrain achieves the highest accuracy with three DNN architectures on nine cross-domain datasets.

Model	Method	Traffic	Omniglot	Aircraft	Flower	CUB	DTD	QDraw	Fungi	COCO	Avg.
MCUNet	None	35.5	42.3	42.1	73.8	48.4	60.1	40.9	30.9	26.8	44.5
	FullTrain	**82.0**	72.7	75.3	90.7	66.4	74.6	64.0	40.4	36.0	66.9
	LastLayer	55.3	47.5	56.7	83.9	54.0	72.0	50.3	36.4	35.2	54.6
	TinyTL	78.9	73.6	74.4	88.6	60.9	73.3	67.2	41.1	36.9	66.1
	SparseUpdate	72.8	67.4	69.0	88.3	67.1	73.2	61.9	41.5	37.5	64.3
	TinyTrain (Ours)	79.3	**73.8**	**78.8**	**93.3**	**69.9**	**76.0**	**67.3**	**45.5**	**39.4**	**69.3**
Mobile NetV2	None	39.9	44.4	48.4	81.5	61.1	70.3	45.5	38.6	35.8	51.7
	FullTrain	75.5	69.1	68.9	84.4	61.8	71.3	60.6	37.7	35.1	62.7
	LastLayer	58.2	55.1	59.6	86.3	61.8	72.2	53.3	39.8	36.7	58.1
	TinyTL	71.3	69.0	68.1	85.9	57.2	70.9	**62.5**	38.2	36.3	62.1
	SparseUpdate	77.3	**69.1**	72.4	87.3	62.5	71.1	61.8	38.8	35.8	64.0
	TinyTrain (Ours)	**77.4**	68.1	**74.1**	**91.6**	**64.3**	**74.9**	60.6	**40.8**	**39.1**	**65.6**
Proxyless NASNet	None	42.6	50.5	41.4	80.5	53.2	69.1	47.3	36.4	38.6	51.1
	FullTrain	78.4	73.3	71.4	86.3	64.5	71.7	63.8	38.9	37.2	65.0
	LastLayer	57.1	58.8	52.7	85.5	56.1	72.9	53.0	38.6	38.7	57.0
	TinyTL	72.5	**73.6**	70.3	86.2	57.4	71.0	65.8	38.6	37.6	63.7
	SparseUpdate	76.0	72.4	71.2	87.8	62.1	71.7	64.1	39.6	37.1	64.7
	TinyTrain (Ours)	**79.0**	71.9	**76.7**	**92.7**	**67.4**	**76.0**	**65.9**	**43.4**	**41.6**	**68.3**

5.3 Evaluation

5.3.1 Experimental Setup

We briefly explain our experimental setup in this subsection.

Datasets. We use *MiniImageNet* [2] as the *meta-train dataset*, following the same setting as prior works on cross-domain FSL [36, 138]. For *meta-test datasets* (*i.e.* target datasets of different domains than the source dataset of MiniImageNet), we employ all nine out-of-domain datasets of various domains from Meta-Dataset [138], excluding ImageNet because it is used to pre-train the models before deployment, making it an in-domain dataset. Extensive experimental results with nine different *cross-domain* datasets showcase the robustness and generality of our approach to the challenging CDFSL problem.

Architectures. Following Lin et al. [44], we employ three DNN architectures: *MCUNet* [251], *MobileNetV2* [154], and *ProxylessNAS* [159]. The models are pre-trained with ImageNet and optimised for resource-limited IoT devices by adjusting width multipliers. Specifically, the backbones of MCUNet (using the 5FPS ImageNet model), MobileNetV2 (with the 0.35 width multiplier), and ProxylessNAS (with a width multiplier of 0.3) have 23M, 17M, 19M MACs and 0.48M, 0.25M, 0.33M parameters, respectively. Note that MACs are calculated based on an input resolution of 128×128 with an input channel dimension of 3.

Evaluation. To evaluate the CDFSL performance, we sample 200 tasks from the test split

for each dataset. Then, we use testing accuracy on unseen samples of a new-domain target dataset. Following Triantafillou et al. [138], the number of classes and support/query sets are sampled uniformly at random regarding the dataset specifications. On the computational front, we present the computation cost in MAC operations and the memory usage. We measure latency and energy consumption when running end-to-end DNN training on actual edge devices.

Baselines. We compare *TinyTrain* with the following five baselines: (1) *None* does not perform any on-device training; (2) *FullTrain* [18] fine-tunes the entire model, representing a conventional transfer-learning approach; (3) *LastLayer* [37, 38] updates the last layer only; (4) *TinyTL* [215] updates the augmented lite-residual modules while freezing the backbone; and (5) *SparseUpdate* of MCUNetV3 [44], is a SOTA method for on-device training that statically pre-determines which layers and channels to update before deployment and then updates them online.

System Implementation[1] The *offline* component of our system is built on a Linux server equipped with an Intel Xeon Gold 5218 CPU and NVIDIA Quadro RTX 8000 GPU. This component is used to obtain the pre-trained model weights, *i.e.* pre-training and meta-training. Then, the *online* component of our system is implemented and evaluated on Pi Zero 2 and Jetson Nano. Pi Zero 2 is equipped with a quad-core 64-bit ARM Cortex-A53 and limited 512 MB RAM. Jetson Nano has a quad-core ARM Cortex-A57 processor with 4 GB of RAM. Also, we do not use sophisticated memory optimisation methods or compiler directives between the inference layer and the hardware to decrease the peak memory footprint; such mechanisms are orthogonal to our algorithmic innovation and may provide further memory reduction on top of our task-adaptive sparse update.

5.3.2 Main Results

Accuracy. Table 5.1 summarises accuracy results of TinyTrain and various baselines after adapting to cross-domain target datasets, averaged over 200 runs. *None* attains the lowest accuracy among all the baselines, demonstrating the importance of on-device training when domain shift in train-test data distribution is present. *LastLayer* improves upon *None* with a marginal accuracy increase, suggesting that updating the last layer is insufficient to achieve high accuracy in cross-domain scenarios, likely due to final layer limits in the capacity. *FullTrain*, serving as a strong baseline as it assumes unlimited system resources, achieves high accuracy. *TinyTL* also yields moderate accuracy. However, as both *FullTrain* and *TinyTL* require prohibitively large memory and computation for training (as shown below), they remain unsuitable to operate on resource-constrained devices.

TinyTrain achieves the best accuracy on most datasets and the highest average accuracy

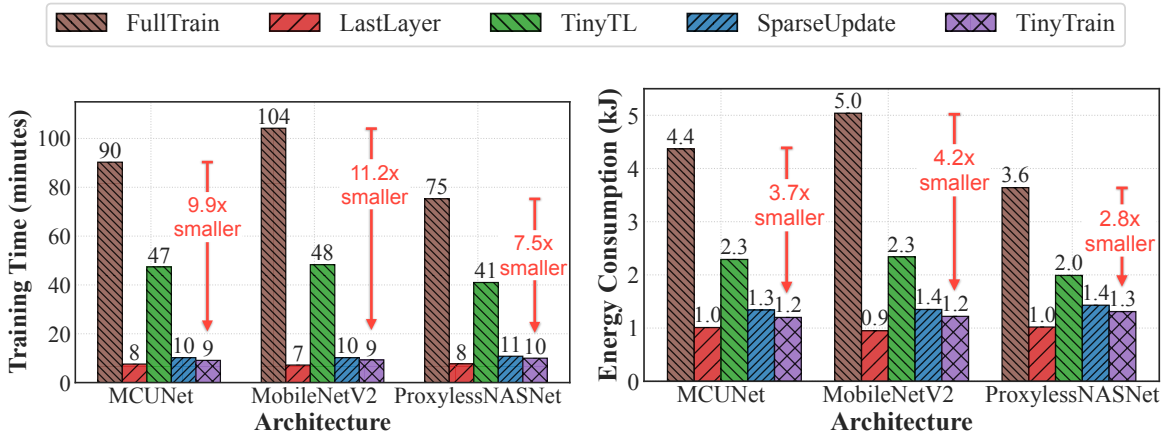
[1] https://github.com/theyoungkwon/TinyTrain

Table 5.2: Comparison of the memory footprint and computation cost for a backward pass.

Model	Method	Memory	Ratio	Compute	Ratio
MCUNet	FullTrain	906 MB	1,013×	44.9M	6.89×
	LastLayer	2.03 MB	2.27×	1.57M	0.23×
	TinyTL	542 MB	606×	26.4M	4.05×
	SparseUpdate	1.43 MB	**1.59**×	11.9M	**1.82**×
	TinyTrain (Ours)	**0.89 MB**	1×	**6.51M**	1×
Mobile NetV2	FullTrain	1,049 MB	987×	34.9M	7.12×
	LastLayer	1.64 MB	1.54×	0.80M	0.16×
	TinyTL	587 MB	552×	16.4M	3.35×
	SparseUpdate	2.08 MB	**1.96**×	8.10M	**1.65**×
	TinyTrain (Ours)	**1.06 MB**	1×	**4.90M**	1×
Proxyless NASNet	FullTrain	857 MB	**1,098**×	38.4M	7.68×
	LastLayer	1.06 MB	1.36×	0.59M	0.12×
	TinyTL	541 MB	**692**×	17.8M	3.57×
	SparseUpdate	1.74 MB	**2.23**×	7.60M	**1.52**×
	TinyTrain (Ours)	**0.78 MB**	1×	**5.00M**	1×

across them, outperforming all the baselines including *FullTrain*, *LastLayer*, *TinyTL*, and *SparseUpdate* by 3.6-5.0 percentage points (pp), 13.0-26.9 pp, 4.8-7.2 pp, and 2.6-7.7 pp, respectively. This result demonstrates the effectiveness of our pipeline of FSL-based pre-training and task-adaptive sparse updates. Also, it indicates that given the limited available samples, fine-tuning the whole DNN (*i.e. FullTrain*) does not necessarily guarantee higher performance in CDFSL tasks as similarly observed in prior work [137]. Instead, our approach of identifying important parameters on the fly in a task-adaptive manner and updating them could be more effective in preventing overfitting than *FullTrain* [263], leading to superior accuracy.

Memory & Compute. We investigate the memory and computation costs to perform a backward pass, which takes up the majority of the memory and computation of training [24, 205]. As shown in Table 5.2, we first observe that *FullTrain* and *TinyTL* consume significant amounts of memory, ranging between 857-1,049 MB and 541-587 MB, respectively, *i.e.* up to 1,098× and 692× more than TinyTrain, which exceeds the typical RAM size of IoT devices, such as Pi Zero (*e.g.* 512 MB). Note that a batch size of 100 is used for these two baselines as their accuracy degrades catastrophically with smaller batch sizes. Conversely, the other methods, including *LastLayer*, *SparseUpdate*, and TinyTrain, use a batch size of 1 and yield a smaller memory footprint and computational cost. Importantly, compared to *SparseUpdate*, TinyTrain enables on-device training with 1.59-2.23× less memory



(a) Training Time (b) Energy Consumption

Figure 5.5: End-to-end latency and energy consumption of the on-device training methods on three architectures.

and 1.52-1.82× less computation. This gain can be attributed to the multi-objective criterion of TinyTrain's sparse-update method, which co-optimises both memory and computation. Note that evaluating our multi-criterion objective does not incur excessive memory overhead.

End-to-End Latency and Energy Consumption. We now examine the run-time system efficiency by measuring TinyTrain's end-to-end training time and energy consumption. To this end, we deploy TinyTrain and the baselines on constrained edge devices, Pi Zero 2 (Figure 5.5) and Jetson Nano (Table 5.4). To measure the overall on-device training cost (excluding offline pre-training and meta-training), we include the time and energy consumption: (1) to load a pre-trained model, and (2) to perform training using all the samples (*e.g.* 25) for a certain number of iterations (*e.g.* 40), and (3) to perform dynamic layer/channel selection for task-adaptive sparse update (only for TinyTrain).

TinyTrain yields 1.08-1.12× and 1.3-1.7× faster on-device training than SOTA on Pi Zero 2 and Jetson Nano, respectively (see Tables 5.3 and 5.4). Also, Figure 5.5 shows that TinyTrain completes an end-to-end on-device training process within 10 minutes, an order of magnitude speedup over the two-hour training of conventional transfer learning, *FullTrain*, on Pi Zero 2. Moreover, the latency of TinyTrain is shorter than all the baselines except for that of *LastLayer* which only updates the last layer but suffers from high accuracy loss. In addition, TinyTrain shows a significant reduction in the energy consumption (incurring 1.20-1.31kJ) compared to all the baselines, except for *LastLayer*, similarly to the latency results.

Summary. *Our results demonstrate that TinyTrain can effectively learn cross-domain tasks requiring only a few samples, i.e. it generalises well to new samples and classes unseen*

Table 5.3: The end-to-end latency breakdown of TinyTrain and SOTA on **Pi Zero 2**.

Model	Method	Fisher Calculation (s)	Run Time (s)	Total (s)	Ratio
MCUNet	SparseUpdate	0.0	607	607	1.12×
	TinyTrain (Ours)	18.7	526	**544**	1×
MobileNetV2	SparseUpdate	0.0	611	611	1.10×
	TinyTrain (Ours)	20.1	536	**556**	1×
ProxylessNASNet	SparseUpdate	0.0	645	645	1.08×
	TinyTrain (Ours)	22.6	575	**598**	1×

Table 5.4: The end-to-end latency breakdown of TinyTrain and SOTA on **Jetson Nano**.

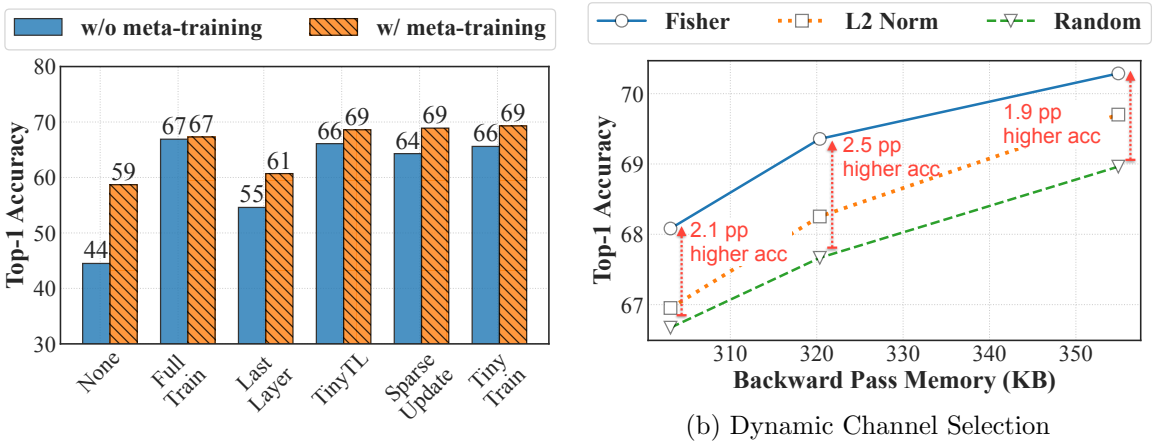
Model	Method	Fisher Calculation (s)	Run Time (s)	Total (s)	Ratio
MCUNet	SparseUpdate	0.0	1,189	1,189	1.3×
	TinyTrain (Ours)	35.0	892	**927**	1×
MobileNetV2	SparseUpdate	0.0	1,282	1,282	1.5×
	TinyTrain (Ours)	32.2	815	**847**	1×
ProxylessNASNet	SparseUpdate	0.0	1,517	1,517	1.7×
	TinyTrain (Ours)	26.8	869	**896**	1×

during the offline learning phase. Furthermore, TinyTrain enables fast and data-efficient on-device training on constrained IoT devices with significantly reduced memory footprint and computational load.

5.3.3 Ablation Study and Analysis

Efficiency of Task-Adaptive Sparse Update. Our dynamic layer/channel selection process takes only 20-35 seconds on our employed edge devices (*i.e.* Pi Zero 2 and Jetson Nano), accounting for only 3.4-3.8% of the total training time of TinyTrain (see Tables 5.3 and 5.4). Our *online* selection process is 30× faster than *SparseUpdate*'s server-based *offline* search, taking 10 minutes with abundant compute resources. *This demonstrates the efficiency of our task-adaptive sparse update (Section 5.2.2).*

Impact of Meta-Training. We compare the accuracy between pre-trained DNNs with and without meta-training using MCUNet. Figure 5.6a shows that meta-training improves the accuracy by 0.6-31.8 pp over the DNNs without meta-training across all the methods. For TinyTrain, offline meta-training increases accuracy by 5.6 pp on average. Note that meta-training does not incur excessive overhead (see below for cost analysis of meta-training). *This result shows the impact of meta-training compared to conventional transfer*



(a) Meta-training

(b) Dynamic Channel Selection

Figure 5.6: The effect of (a) meta-training and (b) dynamic channel selection on MCUNet averaged over nine cross-domain datasets.

learning, demonstrating the effectiveness of our FSL-based pre-training (Section 5.2.1).

Cost Analysis of Meta-Training. We analyse the cost of meta-training, one of the major components of our FSL-based pre-training, in terms of the overall latency to perform meta-training. TinyTrain 's meta-training stage takes place offline (as illustrated in Figure 5.2) on a server equipped with sufficient computing power and memory prior to deployment on-device. In our experiments, the offline meta-training on MiniImageNet takes around 5-6 hours across three architectures. However, note that this cost is small as meta-training needs to be performed *only once* per architecture. Furthermore, this cost is amortised by being able to reuse the same resulting meta-trained model across multiple downstream tasks (different target datasets) and devices, *e.g.* Raspberry Pi Zero 2 and Jetson Nano, while achieving significant accuracy improvements (refer to Table 5.1 and Figure 5.6a).

Robustness of Dynamic Channel Selection. We compare the accuracy of TinyTrain with and without dynamic channel selection, with the same set of layers to be updated within strict memory constraints using MCUNet. This comparison shows how much improvement is derived from dynamically selecting important channels based on our method at deployment time. Figure 5.6b shows that dynamic channel selection increases accuracy by 0.8-1.7 pp and 1.9-2.5 pp on average compared to static channel selection based on L2-Norm and Random, respectively. In addition, given a more limited memory budget, our dynamic channel selection maintains higher accuracy than static channel selection. *Our ablation study reveals the robustness of the dynamic channel selection of our task-adaptive sparse-update (Section 5.2.2).*

Table 5.5: Top-1 accuracy results of TinyTrain based on different multi-objective criteria and L2-Norm-based layer selection scheme. Three DNN architectures are used and accuracy is averaged over nine cross-domain datasets.

Method	MCUNet	MobileNetV2	ProxylessNASNet
L2 Norm	67.9	62.5	62.6
Fisher Only	69.2	64.3	68.2
Fisher / Memory	68.6	63.5	67.5
Fisher / Compute	65.0	62.2	67.5
TinyTrain (Ours)	**69.3**	**65.7**	**68.3**

Impact of Each Component of Multi-Objective Criterion. We experimented with a task-adaptive sparse update based on different versions of our multi-objective criterion: (1) when only Fisher information is used, (2) Fisher information with memory overhead, (3) Fisher information with computation overhead, (4) our final form, *i.e.* Fisher information with memory and computation overheads. Table 5.5 shows that using metrics based on (1) Fisher Only produces strong performance, achieving higher accuracy than (2) and (3) and slightly lower accuracy than (4) our final metric form. This result indicates that Fisher information is very effective in identifying important layers/channels. The other metrics that consider one type of resource, *i.e.* either memory or computation, show slightly lower final accuracy compared to (1) or (4) as they optimise primarily towards one aspect of resource consumption. *Finally, our proposed metric - leveraging all three Fisher information, memory and computation - outperforms the other three metrics, demonstrating the effectiveness of considering both the importance of layers/channels and system resources.*

Impact of Layer Selection Scheme. We compare a top-k layer selection scheme such as an L2-Norm-based selection and TinyTrain. For L2-Norm-based layer selection, a layer with the highest L2-norm of its weights is selected. We set the same memory constraint used for TinyTrain (in Section 5.3.2) and compare their performance. As shown in Table 5.5, compared to the L2-Norm-based layer selection scheme, our proposed method improves the average accuracy by up to 2.0 pp, 5.1 pp, and 9.2 pp on average for nine cross-domain datasets based on MCUNet, MobileNetV2, and ProxylessNASNet, respectively. *This demonstrates the effectiveness of our layer selection scheme.*

5.4 Conclusion

In this chapter, we have developed the first realistic on-device training framework, TinyTrain, solving practical challenges in terms of data, memory, and compute constraints

for edge devices. TinyTrain meta-learns in a few-shot fashion during the offline learning stage and dynamically selects important layers and channels to update during deployment. As a result, TinyTrain outperforms all existing on-device training approaches by a large margin enabling fully on-device training on unseen tasks at the data-scarce edge. It allows applications to generalise to cross-domain tasks using only a few samples and adapt to the dynamics of the user devices and context.

Chapter 6

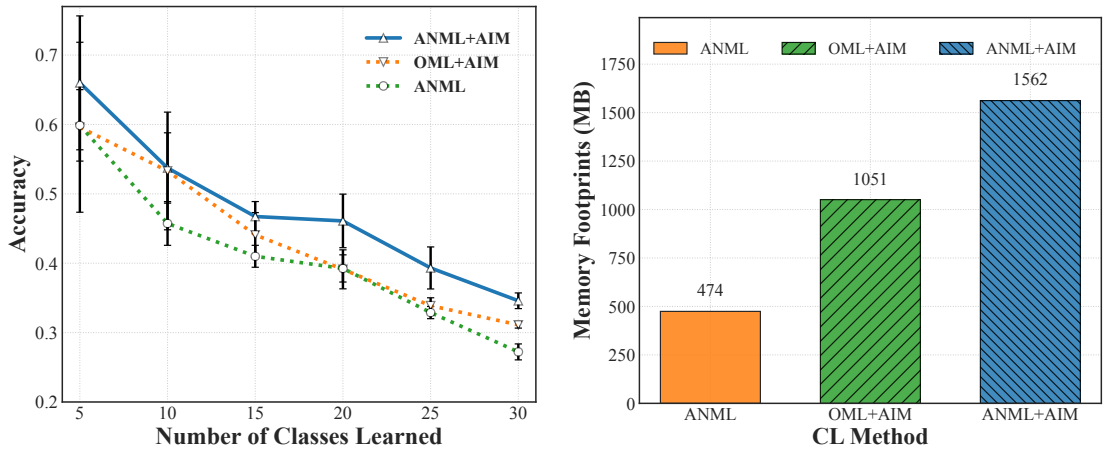
Efficient Continual and On-Device Training on Edge and Microcontrollers

6.1 Introduction

Building on our thorough analysis of CL in mobile applications in Chapter 3, where we identified rehearsal-based methods as superior for mobile sensing applications and developed FastICARL to reduce computational costs and storage requirements, along with our efficient techniques developed for resource-constrained MCUs in Chapters 4 and 5–specifically YONO's model compression and TinyTrain's pre-training and adaptive learning expertise–we sought to advance CL in mobile computing. In this chapter, we introduce several new innovations beyond these previous works, including a novel rehearsal-based Meta CL approach, a CL-tailored compression module utilising both lossless and lossy compression, and a hardware-aware system implementation optimised for devices ranging from edge devices to MCUs with extremely limited resources. This integrated approach enables efficient CL and on-device training across computing platforms previously considered unsuitable for such advanced learning capabilities.

To this end, we investigate *CL-tailored algorithm/software co-design* and *hardware-aware systems* that comprehensively address on-device resource requirements such as data, memory, and computation. However, in real-world setups where deployed models need to dynamically learn new tasks (*i.e.,* new classes or inputs) from users [114] and adapt to changing input distributions [18], existing learning approaches often fail due to resources constraints on edge devices and the phenomenon of *CF* [27] as discussed in Chapter 3.

Recently, many CL approaches have been proposed as described in Section 2.2. As discussed



(a) Performance

(b) Memory Overhead

Figure 6.1: Preliminary analysis of the prior Meta CL methods (i.e., ANML, OML+AIM, ANML+AIM). (a) shows the CL accuracy degradation of the Meta CL methods after learning c number of classes on CIFAR-100 [1]. (b) shows the memory footprint needed to run the Meta CL methods on MiniImageNet [2] with a batch size of 8.

in Chapter 3 and prior works [7, 103, 104], rehearsal-based methods largely alleviate the forgetting issue of learned models. Nonetheless, they are excessively data-hungry as they require a large number of labelled samples to learn new information and to be stored as rehearsal samples [29], incurring high computational and memory overheads to achieve high CL performance.

Another research stream has recently attempted to utilise meta-learning [129] in CL to address the problem of the scarce labelled data. A number of Meta CL methods [35, 139, 140] relying on a few samples of new classes to adapt and learn have been proposed. However, Meta CL's performance degrades when many classes are added during deployment, leading to low scalability (refer to Figure 6.1a). Additionally, SOTA Meta CL methods, OML+AIM and ANML+AIM [35], exhibit large memory footprint, easily exceeding the RAM size on many embedded devices (*e.g.,* 1 GB) (refer to Figure 6.1b). Furthermore, we observed that the end-to-end latency of SOTA Meta CL methods to continually learn multiple classes is computationally expensive. These aspects render prior Meta CL methods unsuitable for deployment on resource-constrained devices. As such, there is an emerging need for novel system design approaches that facilitate the broader deployment of CL systems on various edge and IoT devices by reducing resource requirements of CL methods without jeopardizing their accuracy.

To address the aforementioned limitations, we develop *LifeLearner*, the first hardware-aware system that fully enables data-and memory-efficient CL on constrained edge and

IoT devices. **First,** contrary to the existing Meta CL methods that primarily rely on regularisation and suffer from accuracy loss, we introduce *rehearsal-based Meta CL*; we co-design meta-learning with an efficient rehearsal strategy. This enables LifeLearner to rapidly learn new classes using only a few samples while alleviating CF of already learned classes upon deployment (Section 6.2.1). **Second,** we propose a *CL-tailored algorithm/software co-design approach* that minimises on-device resource overheads of CL. At the algorithmic level, we design a latent replay scheme, where rehearsal samples are extracted from an intermediate layer of the target DNN instead of holding copies of raw inputs. By strategically selecting the rehearsal layer for high compressibility, we facilitate the subsequent compression of rehearsal samples, enabling their efficient storage on-device. Besides, based on an observation that latent replays are sparse, we further design a novel *Compression Module* that combines lossless compression to utilise sparsity and lossy compression to yield a high compression rate, fast encoding and decoding, and minimal resource usage (Section 6.2.2). **Finally,** we develop our *hardware-aware system* by employing hardware-friendly optimisation techniques and considering the unique characteristics of hardware (*e.g.,* costly write operation on Flash memory of IoT devices during runtime) to optimise the runtime efficiency of CL operations on-device (Section 6.3).

Our key contributions are as follows:

- A novel Meta CL method comprising a rehearsal strategy that alleviates CF and a deployment-time inner-and outer-loop training structure. This approach achieves both fast adaptation to new classes and refreshing of already learned classes. LifeLearner achieves previously unattainable levels of on-device accuracy, outperforming all existing Meta CL methods by 4.1-16.1% on image and audio datasets, while remaining within 2.8% of an oracle.
- A new algorithm/software co-design method that co-optimises the rehearsal strategy and the compression pipeline to significantly reduce the resource requirements of CL and rehearsal samples. As a result, LifeLearner requires only 3.40–15.45 MB of memory and obtains a compression rate of 11.4–178.7× compared to the SOTA Meta CL method, ANML+AIM. This allows LifeLearner to run on edge devices, which is impossible for current SOTA methods due to their large memory requirements (>1.05 GB).
- Our hardware-aware system implementation successfully deploys LifeLearner on two embedded devices (Jetson Nano and Raspberry Pi 3B+) and an MCU (STM32H747). Through extensive experiments, we demonstrate that LifeLearner outperforms existing CL and Meta CL baselines in terms of latency and energy consumption. Specifically, compared to ANML+AIM, LifeLearner achieves 80.8-94.2% lower end-to-end latency and 80.9-94.2% lower energy consumption on Jetson Nano. Also, we developed LifeLearner on an extremely resource-constrained IoT device, STM32H747

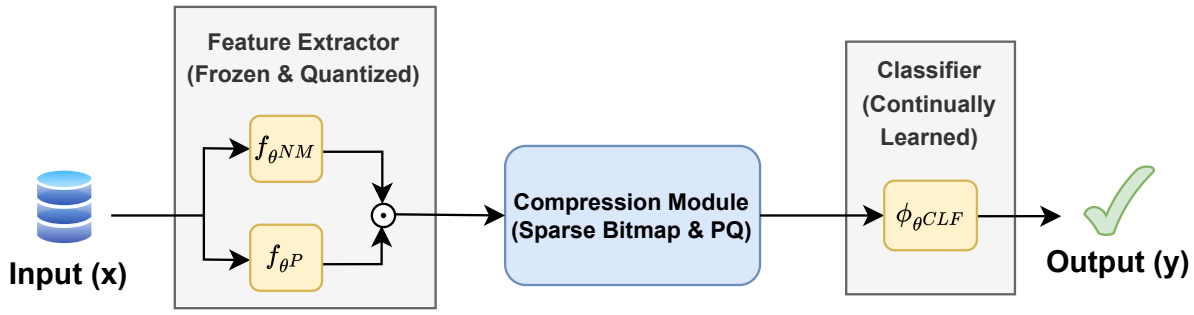


Figure 6.2: The system overview. LifeLearner consists of the frozen/quantised feature extractor, the continually learned classifier, and the compression module based on sparse bitmap and PQ. The compression module takes the feature extractor's outputs (activations) as inputs and compresses them to be saved as latent replay samples.

with 512 KB of SRAM (2,000× smaller memory than Pi 3B+ with 1 GB RAM). To our knowledge, this is the first implementation of a CL framework on this constrained and challenging platform, opening the door for the ubiquitous deployment of learning systems that can continually adapt to users and environments over time.

6.2 LifeLearner

LifeLearner leverages the idea of Meta CL and rehearsal-based learning and minimises the system overheads on embedded devices. LifeLearner consists of two phases. The first phase, i.e., meta-training, is performed on a server to obtain a good weight initialisation by utilizing meta-learning in the CL setup with a few samples. The second phase is meta-testing: a meta-trained model is deployed on embedded devices and learns new classes continually without forgetting previously learned classes. Additionally, as shown in Figure 6.2, LifeLearner has two components to ensure superior performance and efficiency when it is deployed on resource-constrained devices: (1) co-utilisation of Meta CL and rehearsal strategy together with a deployment-time inner- and outer-loop optimisation to resolve the accuracy degradation issue, (2) a design scheme that co-optimises LifeLearner's rehearsal strategy and compression pipeline (*Compression Module* in Figure 6.2) to minimise the memory footprint, compute cost, and energy consumption when running CL.

6.2.1 Co-utilisation of Meta-Learning and Rehearsal Strategy

Current Meta CL methods rely on regularisation in order to minimise radical changes to the already trained weights when learning new classes. As such, given a small set of training data from a stream of classes, all samples are discarded once they have been used.

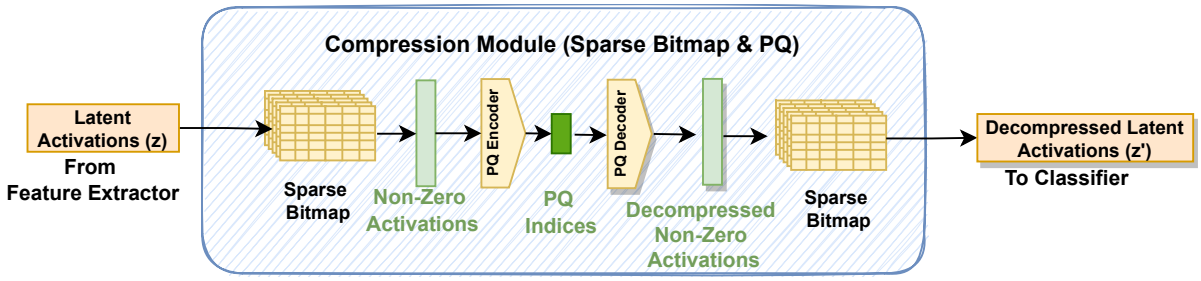


Figure 6.3: The overview of our compression module. It consists of (1) a sparse bitmap to filter out zero from activations or to reconstruct decompressed activations from non-zero activations, (2) a PQ encoder that further compresses non-zero activations into PQ indices, and (3) a PQ decoder that decompresses PQ indices back into decompressed non-zero activations.

However, recent results from the CL literature [104] indicate that the alternative approach of rehearsal-based methods often outperforms regularisation-based CL. Moreover, we opt to use a rehearsal-based approach over dynamic architecture-based methods [121, 122, 123] to avoid the dynamic expansion of the model architecture during deployment, allowing us to apply system optimisations on the static computation graph of the model (see last paragraph of Section 6.3 for details). Driven by this rationale, we design our Meta CL method, called *rehearsal-based Meta CL*, which introduces a rehearsal strategy into the Meta CL to improve CL performance. Concretely, we introduce a Replay Buffer that stores informative samples from already learned classes; these serve as additional training samples when learning new classes, form a mechanism for refreshing the weights of the model, and avoid CF.

In addition, existing Meta CL systems are limited by their sole use of inner-loop optimisation during meta-testing. Instead, we construct a variant of the learning fast and slow weights approach: we utilise the samples of new classes during inner-loop updates to enable rapid adaptation to new classes, followed by outer-loop iterations with the rehearsal samples of the previously learned classes to alleviate CF.

System Overhead. Despite the learning benefits of our rehearsal-based Meta CL method (see Section 6.4.2 for details), it comes at a system cost. With respect to memory, the Replay Buffer has to store a number of representative samples for each of the already encountered classes, so that they can be fetched during meta-testing. With respect to computation, the samples have to be processed by the DNN with both forward and backward passes to perform CL. Unless alleviated, these overheads can lead to a sharp increase in storage and computational requirements, hindering its deployment on mobile and embedded devices, where continual learning is most needed. In the next section, we present LifeLearner's co-design approach for alleviating these system costs.

6.2.2 CL-tailored Algorithm/Software Co-Design

To alleviate the system costs of rehearsal-based Meta CL and enable its deployment on resource-constrained devices, we present an algorithm-software co-design method, optimised for Continual Learning. At the algorithmic level, we design a *rehearsal strategy* that minimises the computational overhead while maximizing the compressibility of the rehearsal samples. At the software level, we design a two-stage *Compression Module* that enables the efficient compression, storage and decompression of rehearsal samples, while inducing minimal on-device resource usage.

Rehearsal Strategy.

Key design decision in rehearsal-based methods constitutes the form of the rehearsal samples. A standard approach followed by many CL methods [31, 112, 114] is *native rehearsal (i.e., raw data replay)*, which stores and replays the input data in their raw format, e.g., images are stored for computer vision tasks and MFCC features for audio tasks. Under this scheme, a random subset of the given classes is stored as rehearsal samples, which are later replayed to mitigate the forgetting issue. The drawbacks of this approach are the significant computational overhead, as the samples have to be processed from the full model, and the compression variability as compressibility varies substantially in a per-sample manner.

To counteract these drawbacks, we introduce *latent replay* into our rehearsal strategy. Under this scheme, instead of holding copies of raw inputs, we store their latent representations, i.e., intermediate activations at the output of a selected layer of the target DNN. In LifeLearner, we employ two techniques in order to enable the utilisation of latent replay: *i)* select the last layer of the model's feature extractor as the rehearsal point; and *ii)* we freeze the feature extractor upon deployment and perform CL only on the classifier. With the feature extractor frozen, we render latent replay functionally equivalent to raw data replay. On the computational front, the forward pass of the feature extractor can be omitted when replaying latent representations and the backward propagation is performed until the last layer, inducing significant computational gains.

On the memory front, we make the following observation. In DNN training, the activations for each layer are saved during the forward propagation so that those activations are utilised for computing the gradients during the backward propagation. As in [205], storing activations requires a large memory footprint depending on the batch size used for training. However, commonly used ReLU non-linearity in many DNN models results in sparse activations in the successive layers. Also, we observe that more than 90% of the activation values of the latent layer are zero due to the usage of ReLU from our analysis of the network architecture on all three datasets. By strategically selecting the rehearsal layer in the DNN and treating ReLU activations as the rehearsal samples, LifeLearner's rehearsal

strategy facilitates their compression and subsequent efficient storage on-device.

Compression Module for Latent Replays

We now introduce the *Compression Module* that is responsible for *i)* compressing rehearsal samples (i.e., latent activations in our work) when new classes are encountered and storing them in the Replay Buffer, and *ii)* fetching and decompressing them to perform CL at runtime. This component comprises two stages: sparse bitmap compression and product quantisation (PQ).

Sparse Bitmap Compression. To leverage the sparsity of our latent replays for efficient storage, we employ sparse bitmap compression [264]. This scheme enables the Compression Module in LifeLearner to filter out the majority of zero values (typically 90% or more) in latent activations and save the remaining non-zero values to increase the compression rate for saving latent activations.

Figure 6.3 depicts the compression and decompression processes. For compression, when latent activations are given to our system, a bitmap with the same dimensions as the latent activations sets a bit to 1 for non-zero values' indices and 0 for the remainders. Then, non-zero values and the sparse bitmap are stored in 32-bit floats and the bitmap format, respectively. For decompression, we traverse all elements of the bitmap and a vector containing the stored non-zero values, reconstructing in this process the latent activations by using either the saved non-zero value or zero if a bitmap element is 1 or 0, respectively. The compression and decompression processes are linear in runtime: $O(n)$, where n is the total number of elements of latent activations. With respect to memory, the footprint is reduced from $(4n)$ when a dense format is used for storing latent activations to $(4 \times \text{number of non-zero values} + \frac{1}{8}n)$ with the bitmap.

Product Quantisation. To further minimise the resource overhead of rehearsal samples, we introduce a second stage to our compressor (Figure 6.3) utilizing PQ [50]. The output of the sparse bitmap compressor contains a vector of non-zero values. With PQ being a vector compression method that can compress a given vector $\mathbf{v} \in \mathbb{R}^d$ into s number of PQ indices using a PQ codebook with s columns, it is suitable to further reduce the size of the encoded rehearsal samples. Each column of the PQ codebook contains a set of representative vectors that well approximate s sub-vectors of $\mathbf{v}$ when $\mathbf{v}$ is partitioned into s sub-vectors.

For compression, the PQ encoder applies PQ to the non-zero activations $\mathbf{v} \in \mathbb{R}^d$ that are already filtered out by the first-stage sparse bitmap compression. We use 1 byte to store each PQ index and set $d/s = \{128, 32, 8\}$ (length of each sub-vector). Then, each sub-vector of length d/s containing 32-bit floats is encoded to a 1-byte PQ index via our PQ encoder for more analysis regarding hyper-parameters). LifeLearner learns the PQ codebook offline using the latent activations during the meta-training phase, which is then

Algorithm 5: Meta-Training Procedure of LifeLearner

Require: N sequential classes $\mathcal{T}$; learning rates (LR) α, β; inner-loop iterations k; modules f_θ, ϕ_θ; given samples S

// Outer-loop starts here

1 **for** $t = 1, ..., N$ **do**

2 $S_{traj} \sim \mathcal{T}_t,\ S_{rand} \sim \mathcal{T}$

// Inner-loop starts here

3 **for** $i = 1, ..., k$ **do**

4 Update fast weights using S_{traj} ▷ LR: α

/* OML(+AIM):$\phi_{\theta^{PLN}}(f_{\theta^W})$, ANML(+AIM):$f_{\theta^P}, \phi_{\theta^{CLF}}(f_{\theta^W})$, LifeLearner:$\phi_{\theta^{CLF}}$ */

5 Update slow weights using $\{S_{traj}, S_{rand}\}$ ▷ LR: β

/* OML(+AIM):$f_{\theta^{RLN}}$, ANML(+AIM):$f_{\theta^{NM}}, f_{\theta^P}, \phi_{\theta^{CLF}}$, LifeLearner:$f_{\theta^{NM}}, f_{\theta^P}, \phi_{\theta^{CLF}}$ */

stored on-device. For decompression, the PQ decoder reconstructs the non-zero activations $\mathbf{v}'$ using the stored PQ indices and the PQ codebook.

Finally, as in Algorithm 6 (see Lines 7, 9, and 10), our compression module is seamlessly incorporated in the inner- and outer-loop optimisation of LifeLearner, enabling on-the-fly compression of the latent activations during deployment.

6.2.3 Putting It All Together

Having described the main components of LifeLearner we now present the complete meta-training and meta-testing procedures that take place offline and online, respectively.

Meta-Training Procedure. Algorithm 5 shows the procedure of meta-training of Rehearsal-based Meta CL, LifeLearner. Firstly, the meta-training process of rehearsal-based Meta CL is the same as that of Meta CL [139]. In detail, it is comprised of an inner loop inside an outer loop of optimisation. In the inner loop, the classifier part is updated (fast weights, e.g., θ^{PLN} for OML and $\theta^{P,CLF}$ for ANML, $\theta^{PLN,W}$ for OML+AIM, and $\theta^{P,CLF,W}$ for ANML+AIM) (Lines 4-5). The number of weight update iterations is determined by the number of samples k (e.g., 10-30) of a given sample set, S_{traj}, of a new class, $\mathcal{T}_t$. After the k sequential updates, the meta-loss in the outer loop (Line 6) is computed using all the given samples on the new class (S_{traj}) and randomly sampled samples from all the meta-training classes (S_{rand}). All the weights of DNN are updated through outer-loop gradient updates using an Adam optimiser [241]. The learning rates, α for the inner loop and β for the outer loop, are used as hyper-parameters.

Meta-Testing Procedure. After executing the meta-training phase on a server, our system is deployed on resource-constrained devices and evaluated on its ability to learn

Algorithm 6: Meta-Testing Procedure of LifeLearner

Require: N sequential unseen classes $\mathcal{T}$; learning rates (LR) α, β; inner-loop iterations k; modules f_θ, ϕ_θ, $BitPQ_{compress,decompress}$; samples S

1 $S_{train} = \{\}$, $S_{rehearsal} = \{\}$

// Outer-loop starts here

2 **for** $t = 1, ..., N$ **do**

3 $S_{traj} \sim \mathcal{T}_t$

4 $S_{train} = \{S_{train}, S_{traj}\}$

// Inner-loop starts here

5 **for** $i = 1, ..., k$ **do**

6 Update fast weights using S_{traj} ▷ LR: α

/* OML(+AIM):$\phi_{\theta^{PLN}}(f_{\theta^W})$, ANML(+AIM):$f_{\theta^P}, \phi_{\theta^{CLF}}(f_{\theta^W})$, LifeLearner:$\phi_{\theta^{CLF}}$ */

// Get latent activations from compressed rehearsal samples

7 $S_{latent} = BitPQ_{decompress}(S_{rehearsal})$

8 Update slow weights using $\{S_{traj}, S_{latent}\}$ ▷ LR: β

/* OML(+AIM):$f_{\theta^{RLN}}$, ANML(+AIM):$f_{\theta^{NM}}, f_{\theta^P}, \phi_{\theta^{CLF}}$, LifeLearner:$\phi_{\theta^{CLF}}$ */

// Get latent activations

9 $S_{latent} = f_{\theta^{NM}}(S_{traj}) \odot f_{\theta^P}(S_{traj})$

// Store compressed activations for rehearsal

10 $S_{rehearsal} = \{S_{rehearsal}, BitPQ_{compress}(S_{latent})\}$

11 $S_{test} = \mathcal{T} - S_{train}$ // Held-out test set

12 Evaluate on S_{train}, S_{test} // Eval on training/test set

unseen classes in the meta-testing phase. Algorithm 6 shows the meta-testing phase of the rehearsal-based Meta CL. In prior Meta CL, the meta-testing procedure contains only inner-loop optimisation without outer-loop optimisation, i.e., finetuning only fast weights excluding slow weights. In contrast, LifeLearner leverages the full potential of meta-learning by using both inner- and outer-loop optimisation in the meta-testing phase. Specifically, our proposed meta-testing procedure starts with the inner-loop weight updates to learn new classes swiftly using a few samples (Lines 5-6), followed by the outer-loop weight updates to retain the knowledge of the previously learned classes using the replayed samples plus the new samples (Line 8). Note that although the outer-loop iteration could run multiple epochs, the performance converges after one or two epochs (refer to Section 6.4.4 for more analysis). Also, LifeLearner integrates the compression module that compresses (Lines 9-10) and decompresses (Line 7) the latent activations during outer-loop optimisation, as described in Section 6.2.2.

Our Contribution. Our method conceptually leverages existing concepts. We solve the challenge of incorporating these concepts in a coordinated, efficient end-to-end system . We

achieve higher accuracy than baselines while reducing the memory footprint drastically. Our key contributions are (1) co-designing the algorithmic innovation (rehearsal strategy) with an intelligent combination of lossless (bitmap) and lossy (PQ) compression to significantly reduce the resource requirements of CL and latent replay samples (Section 6.2), (2) successfully deploying LifeLearner end-to-end on two embedded devices and MCU on which many prior works fail to run (Section 6.3).

6.3 Hardware-Aware System Implementation

We develop the first phase, meta-training, of Meta CL methods on a Linux server to initialise the neural weights that can enable fast adaptation during deployment scenarios. After that, for the second phase, meta-testing, (i.e., actual deployment scenarios), we implemented our hardware-aware system by considering the hardware capacity and unique runtime characteristics of our target devices: (1) embedded and mobile systems such as Jetson Nano and Raspberry Pi 3B+, and (2) a microcontroller unit such as STM32H747. To further optimise the system efficiency, we adopt hardware-friendly optimisation techniques in our implementation[1].

Embedded Device. Jetson Nano has a quad-core ARM Cortex-A57 processor, and 4 GB of RAM, while Pi 3B+ contains a quad-core ARM Cortex-A53 processor with 1 GB of RAM. Note that the free memory space of Jetson Nano and Pi 3B+ during idle time is roughly 1.7 GB and 600 MB, respectively, due to the memory footprints pre-occupied by background, concurrent applications, and an operating system. As software platforms, we employ Faiss (PQ Framework) [265] and PyTorch 1.8 (Deep Learning Framework) [266] to develop and evaluate the meta-training and meta-testing phases on embedded systems.

Microcontroller Unit (MCU). To demonstrate the feasibility of the broader deployment of CL systems at the extreme edge, we further optimised and developed LifeLearner on MCUs. We implemented the online component of LifeLearner using C++ on an STM32H747 device equipped with ARM Cortex M4 and M7 cores with 1MB SRAM and 2 MB eFlash in total. However, we only utilise one core (ARM Cortex M7), as most MCUs have one CPU core. Also, we restrict the usage space of SRAM and eFlash to 512 KB and 1 MB, respectively, to enforce stricter resource constraints (an order of magnitude smaller memory space than other embedded devices with larger than 1 GB RAM).

To deploy LifeLearner on MCUs effectively and efficiently, we addressed many technical challenges and considered hardware characteristics. First of all, the memory requirements of the MetaCL methods developed on embedded devices, including LifeLearner, far exceed the hardware capacity of a "high-end" MCU such as STM32H747 (refer to Section 6.4.2). Hence, we first searched for a smaller yet accurate architecture for MCUs by experimenting

[1] https://github.com/theyoungkwon/LifeLearner

with various width modifiers [154, 251, 267] (see Section 6.4.5 for details).

We then implemented our Compression Module (sparse bitmap compression and PQ) to reduce memory usage of latent replay samples on SRAM. In particular, we consider hardware characteristics and constraints: (1) the write operation on the storage (Flash) of MCUs is costly [268], and (2) Flash is read-only during runtime [52, 242]. Hence, in our MCU implementation of LifeLearner, to minimise the memory footprint and energy consumption required for latent replay, we first compress latent replay samples using our Compression Module and then store them on SRAM, which has more limited memory but is faster and cheaper to perform read/write operations on than Flash. Note that our learned PQ codebook, used to encode and decode the latent replay samples after sparse bitmap compression, is stored on Flash to leave more space for scarce resources of SRAM. Also, PQ codebooks are static once deployed; they can be stored on the read-only memory of Flash.

In addition, we rely on the TFLM framework [127] to perform inference of the feature extractor on MCUs. However, TFLM does not support training (i.e., backpropagation). We developed our Backpropagation Engine based on C/C++ using Eigen [269] as a data structure and matrix multiplication library. Based on our Backpropagation Engine, we construct the classifier part on the fly whose weights are allocated on SRAM and can be continually learned during deployment whenever more data for new classes become available. Our lightweight Backpropagation Engine enables the implementation of the first CL system on MCUs.

Lastly, the binary size of our Compression Module and Backpropagation Engine, excluding C++ Standard Library (STL) on an MCU, is only 80 KB, introducing minimal overhead on storage.

Hardware-friendly Optimisation. We further optimise LifeLearner's CL operations on-device. By freezing the model's feature extractor during deployment, LifeLearner significantly reduces the computational cost for the already learned classes during replay by omitting the forward and backward passes. In addition, we utilise the hardware-friendly 8-bit integer arithmetic [270] by reducing the precision of weights/activations of the feature extractor from 32-bit floats to 8-bit integers, increasing the computation throughput and minimizing latency and energy. The scalar quantisation scheme [48, 200] is used to minimise the information loss in quantisation. Then, we utilise the QNNPACK [271] backend engine and TFLM to execute the quantised model on two embedded devices and MCUs, respectively.

6.4 Evaluation

6.4.1 Experimental Setup

We briefly describe our experimental setup in this subsection.

Metrics

As in [139], we use testing accuracy on unseen samples of all the new classes learned continually as a key performance metric, representing the generalisation ability of CL systems. In addition, we measure the memory footprint (model parameters, optimisers, activations, and rehearsal samples), end-to-end training latency and energy consumption to continually learn all the given classes for a deployed DNN on embedded devices.

Datasets

We employ three datasets of two different data modalities in our evaluation.

CIFAR-100 [1]: Following [35], we employ CIFAR-100 in our evaluation as it is widely used dataset. CIFAR-100 consists of 60,000 images of 100 classes. Each class has 500 train images and 100 test images. 70 classes are used for meta-training and the remaining 30 for meta-testing. During both meta-training and meta-testing, up to only 30 training images are sampled for training in each class, which holds for both MiniImageNet and GSCv2 datasets. Then, during meta-testing, a total of 900 samples are given to perform CL.

MiniImageNet [2]: Following [35], we employ MiniImageNet containing 64 classes for meta-training and 20 classes for meta-testing. Each class has 540 images for training and 60 images for testing. During meta-testing, a total of 600 samples are given.

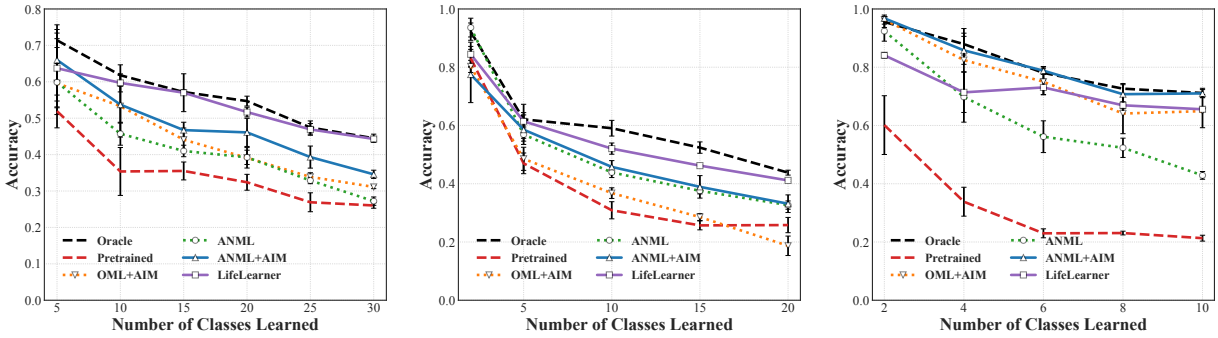
GSCv2 [253]: To generalise our results to another data modality, we include Google Speech Command V2 (GSCv2) as it is a widely used audio dataset. GSCv2 consists of a total of 35 classes of different keywords. We use 25 classes for meta-training and 10 classes for meta-testing. Each class has 2,424 and 314 input data for training and testing, respectively. During meta-testing, 300 samples in total are given for CL.

Baselines

We compare our system, *LifeLearner*, with five baseline systems as follows.

Oracle: The CL performance of Oracle represents the upper bound performance of the experiments. It is because Oracle has access to all the classes at once in an i.i.d. fashion and performs DNN training for many epochs until the performance converges.

Pretrained: This baseline initialises the model weights based on conventional DNN



(a) CIFAR-100 (b) MiniImageNet (c) GSCv2

Figure 6.4: The accuracy of the CL systems on the three datasets of two different modalities. Reported results are averaged over three trials, and standard-deviation intervals are depicted.

training without the meta-learning procedure. Then, it finetunes the weights using given samples in the meta-test phase, similar to prior Meta CL methods.

OML+AIM [35]: This is a Meta CL method based on OML with an Attentive Independent Mechanisms (AIM) module, capturing independent concepts to learn new knowledge.

ANML [139]: It is the representative Meta CL method. As this method is often reported to outperform OML [140], we only employ ANML in our evaluation. Also, note that the proposed components of LifeLearner build on top of ANML.

ANML+AIM [35]: ANML+AIM is a Meta CL method based on ANML with an AIM module. This baseline serves as the SOTA Meta CL method as it often outperforms other Meta CL methods including OML+AIM.

Model Architecture

LifeLearner employs the network architecture used in the prior CL works for a fair comparison [35, 139]. As in Figure 6.2, it consists of the feature extractor and the final classifier. For ANML-based model architectures, the feature extractor consists of a neuromodulatory network, $f_{\theta^{NM}}$, and a prediction network, $f_{\theta^{P}}$, followed by the classifier part, $f_{\theta^{CLF}}$. The neuromodulatory and prediction networks are 3-layer convolutional networks with 112 and 256 channels, respectively. The classifier has a single fully-connected layer. In this case, LifeLearner utilises the last layer of the feature extractor as the latent

replay layer, following the natural structure of the ANML architecture.[2] The SOTA method, ANML+AIM, adds AIM layers $f_{\theta W}$ between the feature extractor and the classifier, which alleviates forgetting and helps learn new classes. In addition, for OML and OML+AIM, the feature extractor has a 6-layer convolutional network with 112 channels, followed by the classifier of two fully-connected layers with an AIM module between the feature extractor and the classifier. Note that the model architectures deployed on embedded devices (i.e., Jetson Nano and Pi 3B+) and an MCU (i.e., STM32H747) are different due to the strict resource constraint on the MCU. Thus, a smaller version of the model architecture described above is adopted for the MCU deployment (see Section 6.4.5 for details).

Training Details

We followed the meta-training procedure used in prior Meta CL works [35, 139, 140]. For instance, we used a batch size of 1 and 64 for the inner- and outer-loop updates over 20,000 steps, respectively. We experimented with different learning rates for the inner loop and outer loop to obtain the meta-trained DNN that provides the best accuracy on a validation set. As a result, for CIFAR-100 and GSCv2 datasets, the inner-loop learning rate (α) is set to 0.001, and the outer-loop learning rate (β) is also set to 0.001. For the MiniImageNet dataset, the optimal settings are $\alpha = 0.001$ and $\beta = 0.0005$. During the meta-testing phase, ten different learning rates are tried for all the methods, and the best-performing results are reported. Besides, to obtain the accuracy results of systems that perform replays, we experimented with batch sizes of 8 and 16 and observed little difference in CL performance. Thus, we employ a batch size of 8, as a smaller batch size reduces the memory footprint.

6.4.2 Experimental Results

Accuracy. We start by evaluating the CL performance (testing accuracy) of LifeLearner compared to the baselines on the employed datasets. Figure 6.4 presents the accuracy results of the meta-testing phase. Pretrained serves as the lower bound. The low accuracy (24.4% on average for three datasets) of Pretrained demonstrates that the conventional transfer learning approach cannot address the challenging scenarios of learning new classes with only a few samples. ANML improves upon Pretrained, however, the improvement is marginal (i.e., average 9.9% accuracy gain compared to Pretrained but 18.9% accuracy drop on average compared to Oracle which shows the upper bound accuracy). Note that it is very challenging to achieve high testing accuracy even for Oracle as the number of available samples is very limited during meta-testing: all evaluated systems are given only

[2]When targeting a different model architecture, the latent replay layer selection is a configurable design decision. We leave this investigation as future work.

Table 6.1: The required memory footprint and the compression ratio for the baselines and our system to perform CL during the meta-testing phase on the three datasets.

Dataset	Metrics	Pretrained	ANML	OML+AIM	ANML+AIM	Oracle	LifeLearner
CIFAR-100	Memory	39.69MB	39.69MB	834.1MB	1,093MB	39.93MB	**15.45MB**
	Ratio	27.5×	27.5×	1.3×	1.0×	27.4×	**70.8**×
Mini-ImageNet	Memory	474.5MB	474.5MB	1,051MB	1,562MB	475.0MB	**136.7MB**
	Ratio	3.3×	3.3×	1.5×	1.0×	3.3×	**11.4**×
GSCv2	Memory	10.16MB	10.16MB	135.2MB	608.2MB	10.20MB	**3.40MB**
	Ratio	59.9×	59.9×	4.5×	1.0×	59.6×	**178.7**×

30 samples per class, accounting for only 2.57%, 1.74%, and 0.5% of all training samples during meta-training of CIFAR-100, MiniImageNet, and GSCv2, respectively.

LifeLearner achieves near-optimal CL performance, falling short by only 2.8% accuracy compared to Oracle. Also, LifeLearner outperforms the Meta CL methods employed in this thesis with accuracy gains of 4.1-16.1% on average for the three datasets. Specifically, LifeLearner shows almost no loss of accuracy, i.e., 0.2% for CIFAR-100 and 2.7% for MiniImageNet compared to Oracle. In contrast, ANML+AIM (i.e., the previous SOTA Meta CL method) shows notable accuracy drops (9.9% for CIFAR-100 and 10.7% for MiniImageNet). In the case of GSCv2, LifeLearner reveals a slight accuracy decline of 5.6% compared to Oracle, while ANML+AIM shows a minor 0.2% drop in accuracy relative to Oracle.

Although LifeLearner shows a slightly lower accuracy for GSCv2 than ANML+AIM, it still outperforms ANML+AIM by 4.1% on average over all datasets. In addition, LifeLearner is essentially designed for edge devices to require drastically lower system resources (memory, latency, and energy) than the previous SOTA. As explained in the following, the excessive resource overhead of ANML+AIM makes it unsuitable to operate on resource-constrained devices.

Peak Memory Footprint. We investigate the peak memory footprint required to perform CL. Precisely, we measure the memory space required to perform backpropagation and to store rehearsal samples. The memory requirement to perform backpropagation consists of three components: (1) model memory that stores model parameters, (2) optimiser memory that stores gradients and momentum vectors, and (3) activation memory that is comprised of the intermediate activations (stored for reuse during backpropagation). Then, the memory requirement for rehearsal samples is included.

Table 6.1 shows the peak memory footprint for various baselines and our system. First, the AIM variants (OML+AIM and ANML+AIM) require an enormous memory footprint of 135.2-1,051 MB and 608.2-1,562 MB, respectively, as their AIM module has many

parameters. This required memory easily exceeds the RAM size of embedded devices such as Pi 3B+ (i.e., 1 GB) and barely fits on Jetson Nano. Conversely, baseline systems such as Pretrained, ANML, and Oracle show modest memory requirements, which are around 10.16-10.20 MB for GSCv2, 39.7-39.9 MB for CIFAR-100, and 474.5-475.0 MB for MiniImageNet. However, as shown earlier, Pretrained and ANML methods are not highly accurate, and Oracle does not support CL. In contrast, LifeLearner shows the impressive results that it only requires 15.45 MB for CIFAR-100, 136.7 MB for MiniImageNet, and 3.40 MB for GSCv2, demonstrating a very high compression rate of 70.8×, 11.4×, and 178.7× compared to ANML+AIM, respectively. Compared to Oracle, LifeLearner shows a tight range of the compression (2.5-3.5×), indicating that we can estimate the compression gain within this range agnostic to the dataset.

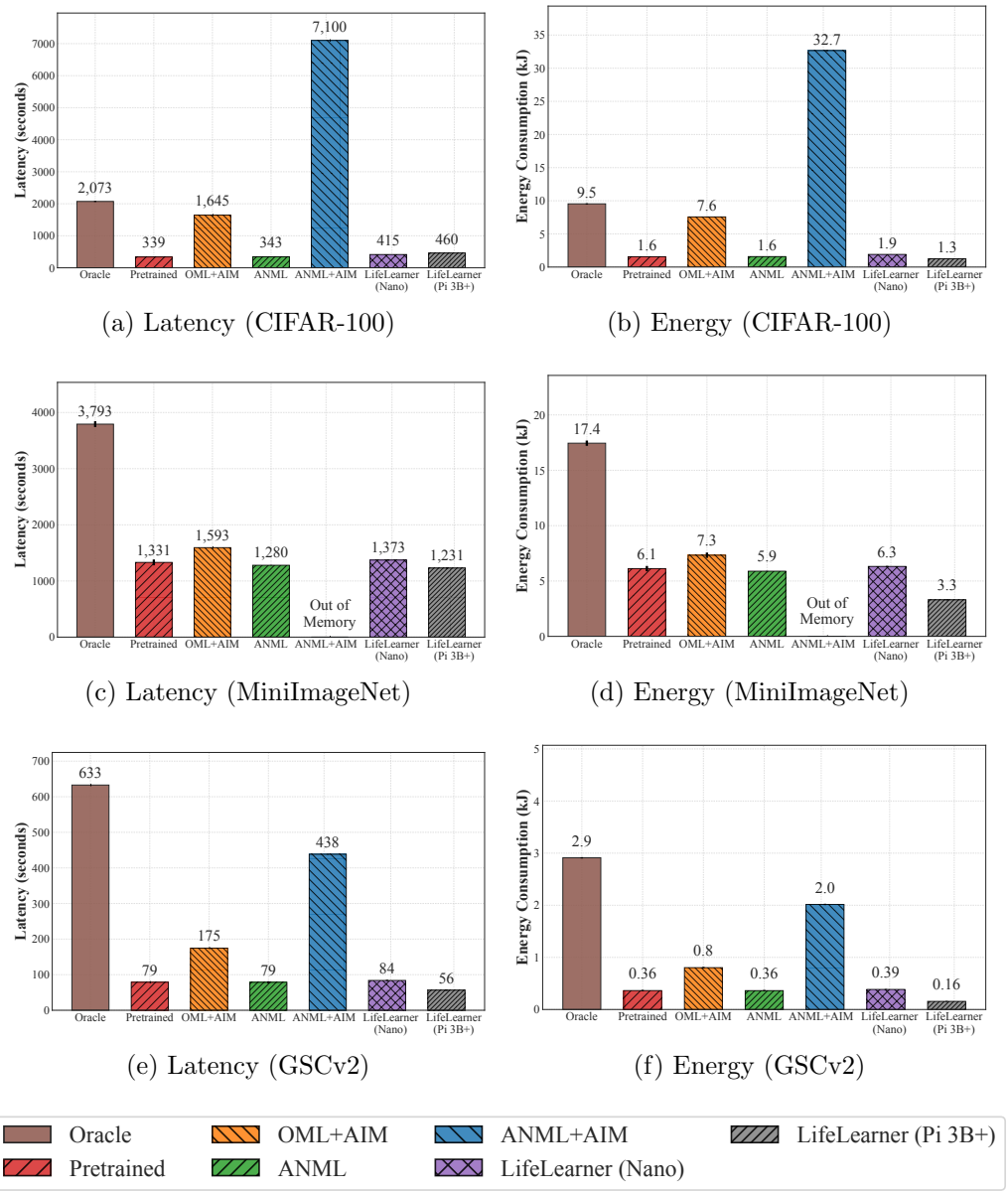
End-to-end Latency & Energy Consumption. We now examine the run-time system efficiency, i.e., end-to-end latency and energy consumption for the entire CL process, of our system and the baselines when deployed on the two embedded devices - Jetson Nano and Pi 3B+ as shown in Figure 6.5. To obtain the end-to-end latency, we include: (1) the time to load a pretrained model, (2) the time to train the model continually over all the given classes one by one, and (3) the time to compress and decompress the latent representations using our compression method (i.e., sparse bitmap compression and PQ).

We first measure the end-to-end latency of our system and the baselines on Jetson Nano CPU to perform CL over all the given classes with 30 samples per class. As shown in Figures 6.5a, 6.5c, and 6.5e, LifeLearner enables a fast end-to-end latency (415 seconds for CIFAR-100, 1,373 seconds for MiniImageNet, and 84 seconds for GSCv2), which is 80.8-94.2% reduction of latency compared to ANML+AIM (e.g., 7,100 seconds for CIFAR-100 and 438 seconds for GSCv2). Note that ANML+AIM often crashes from running out of memory on Jetson Nano due to its excessive memory requirements (as shown in Figures 6.5c and 6.5d). Furthermore, compared to ANML which shares the same network architecture, LifeLearner introduces negligible overheads in terms of the overall latency (343s vs. 415s for CIFAR-100, 1,280s vs. 1,373s for MiniImageNet, and 79s vs. 84s for GSCv2). It is because although there exist some overheads on LifeLearner to perform the compression techniques like the sparse bitmap compression and PQ, the speed gains derived from using quantised neural weights and activations offset the overheads of compression techniques (refer to Section 6.4.3 for details). After having demonstrated the efficiency of LifeLearner on the Jetson Nano, we deployed our system on an even more resource-constrained device, Pi 3B+ (600-700 MB available memory). The end-to-end latency on Pi 3B+ largely stays similar to that on Jetson Nano as shown in Figure 6.5.

To measure the energy consumption, we first use Tegrastats on Jetson Nano to measure the power consumption. Then, we calculate the energy consumption by multiplying power consumption and the elapsed time for each end-to-end CL trial. Similar to the latency results, Figures 6.5b, 6.5d, and 6.5f show that LifeLearner remarkably reduces the energy

consumption by 80.9-94.2% (1.9kJ vs. 32.7kJ for CIFAR-100 and 0.4kJ vs. 2.0kJ for GSCv2) compared to ANML+AIM. Moreover, compared to ANML, LifeLearner shows small overheads of the additional energy consumption (1.6kJ vs. 1.9kJ for CIFAR-100, 5.9kJ vs. 6.3kJ for MiniImageNet, and 0.36kJ vs. 0.39kJ for GSCv2). In the case of Pi 3B+, it consistently consumes less energy than Jetson Nano. It is because while the end-to-end latency of the two embedded devices is similar, the power consumption profile on Pi 3B+ is lower than that on Jetson Nano, making Pi 3B+ a more energy-efficient option. A YOTINO USB power meter is used to obtain the power consumption on Pi 3B+.

Summary. *Our result demonstrates that LifeLearner can effectively learn new classes in a continual manner based on only a few samples without experiencing catastrophic forgetting, i.e., it generalises well to new samples of many classes unseen during the offline learning phase. Moreover, LifeLearner enables fast and energy-efficient CL on edge devices with significantly reduced memory footprint.*



(a) Latency (CIFAR-100) (b) Energy (CIFAR-100)

(c) Latency (MiniImageNet) (d) Energy (MiniImageNet)

(e) Latency (GSCv2) (f) Energy (GSCv2)

Figure 6.5: The end-to-end latency and energy consumption of the baselines and LifeLearner to perform CL over all the given classes. All results are averaged over three runs with standard deviations.

6.4.3 Ablation Study

We perform an ablation study to investigate the role of each component of our system by incrementally adding our proposed components on top of the baseline system (ANML): (1) rehearsal strategy with inner-and outer-loop optimisation (Latent), (2) sparse bitmap compression (Latent+Bit), (3) PQ (Latent+PQ), and (4) quantisation (LifeLearner).

Effect of Rehearsal with Double-Loop Optimisation. As shown in Table 6.2, we find that our proposed rehearsal strategy with double-loop optimisation drastically improves the accuracy (compare ANML vs Latent). For example, Latent increases the accuracy of ANML by 10.6-28.4% across all the datasets. Yet, Latent causes resource overheads on memory footprint, latency, and energy consumption compared to ANML, as Latent is a baseline CL system without our Compression Module.

Effect of Compression and Hardware-aware Implementation. The results of various CL systems such as Latent+Bit, Latent+PQ, and Latent+Bit+PQ show that the proposed compression techniques for latent representations do not sacrifice the accuracy of the CL systems but reduce the overall memory footprint compared to Latent. Moreover, our Compression Module incurs small resource overheads in end-to-end latency and energy. Then, LifeLearner, which combines quantisation of weights and activations accelerating the CL execution on hardware by exploiting efficient integer-based operations, shows excellent performance in all aspects: (1) outperforms ANML by a large margin (8.4-22.7%) with a minor accuracy drop compared to Latent (0.9-5.7%), (2) drastically reduces the memory footprint by 61.0-71.2% compared to ANML and by 71.2-73.3% compared to Latent, and (3) incurs minimal overheads of latency and energy over ANML (costs additional 56.6s and 0.3kJ on average, respectively) but still shows lower latency and energy than Latent (saves 47.9s and 0.2kJ on average, respectively).

Overall, the ablation study reveals that the co-utilisation of the rehearsal strategy with double-loop optimisation, Compression Module, and hardware-friendly implementation effectively makes LifeLearner more accurate and efficient.

6.4.4 Parameter Analysis

Next, we study the impact of the various hyper-parameters that could affect the performance of our system (see Figure 6.6).

The Number of Given Samples. We first examine the accuracy of LifeLearner according to the number of given samples per class (ranging from 10 to 30) as it would directly affect labeling effort of users (see Figure 6.6a). Apparently, the more samples are given for training, the higher the accuracy, which holds for both LifeLearner and Oracle. Even when only 10 samples per class are given to conduct training, the accuracy degradation of LifeLearner is relatively low (7-14%), indicating that LifeLearner can still perform

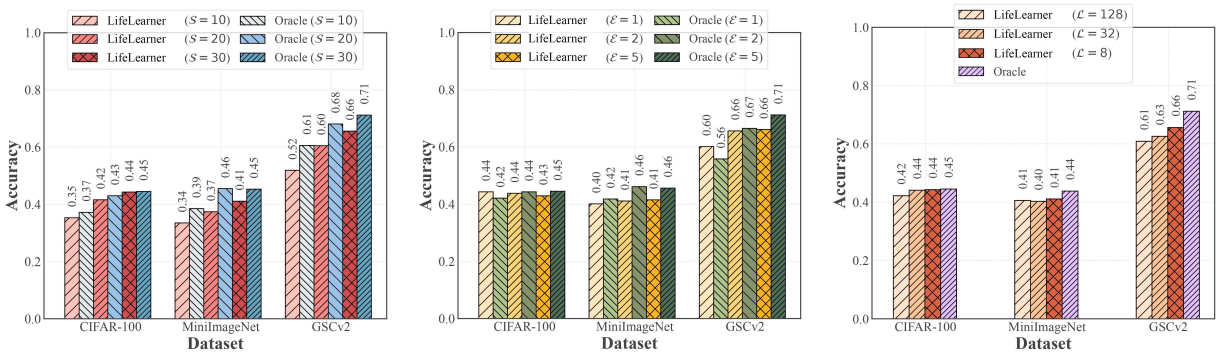
Table 6.2: The comparison of LifeLearner and variants of rehearsal-based Meta CL methods for ablation study.

Dataset	System	Accuracy	Memory	Latency	Energy
CIFAR-100	ANML	0.272	39.7 MB	343.2s	1.58kJ
	Latent	0.452	53.9 MB	432.5s	1.99kJ
	Latent+Bit	0.452	41.2 MB	466.9s	2.15kJ
	Latent+PQ	0.448	41.8 MB	437.1s	2.01kJ
	Latent+Bit+PQ	0.446	40.4 MB	471.4s	2.17kJ
	LifeLearner	**0.443**	**15.5 MB**	**414.7s**	**1.91kJ**
Mini-ImageNet	ANML	0.327	474.5 MB	1,280s	5.89kJ
	Latent	0.433	512.5 MB	1,492s	6.86kJ
	Latent+Bit	0.433	477.7 MB	1,551s	7.14kJ
	Latent+PQ	0.430	483.0 MB	1,501s	6.90kJ
	Latent+Bit+PQ	0.423	476.4 MB	1,560s	7.18kJ
	LifeLearner	**0.411**	**136.7 MB**	**1,373s**	**6.32kJ**
GSCv2	ANML	0.429	10.2 MB	78.6s	0.36kJ
	Latent	0.713	12.0 MB	90.6s	0.42kJ
	Latent+Bit	0.713	10.4 MB	90.8s	0.42kJ
	Latent+PQ	0.708	11.0 MB	95.0s	0.44kJ
	Latent+Bit+PQ	0.707	10.3 MB	95.2s	0.44kJ
	LifeLearner	**0.656**	**3.40 MB**	**83.8s**	**0.39kJ**

reasonably well under extreme data scarcity. Also, the accuracy differences between LifeLearner and Oracle are small (e.g., 1-2% for CIFAR-100, 1-3% for MiniImageNet, and 5-9% for GSCv2), demonstrating that LifeLearner achieves the similar accuracy of Oracle. With 30 given samples, the accuracy difference is minimal: 2.8% on average (ranging from 1 to 5%).

The Number of Replay Epochs. We study to what extent the number of replay epochs affects the CL performance as more epochs incur larger latency and energy consumption. Figure 6.6b shows that the accuracy of LifeLearner converges after the first or the second replay epoch. However, Oracle requires at least two to five epochs to reach the convergence accuracy, which consumes much more training time and energy than our system (see Figure 6.5). This result benefits us since replaying the rehearsal samples over one or two epochs is enough for LifeLearner to reach the converging accuracy, which helps decrease the system overheads.

PQ Codebook's Sub-vector Length. We investigate the accuracy of LifeLearner according to the sub-vector length of the PQ codebook (the number of values per index) ranging from 8 to 128 as it affects the compression ratio of rehearsal samples. For CIFAR-



(a) Number of Samples per Class ($\mathcal{S}$) (b) Number of Replay Epochs ($\mathcal{E}$) (c) Sub-Vector Length ($\mathcal{L}$)

Figure 6.6: The parameter analysis of LifeLearner for all the datasets according to the three parameters.

100 and MiniImageNet, there is little difference according to the sub-vector length. In contrast, for GSCv2, we observe that the shorter the length of the sub-vector (i.e., lower compression rate), the higher the accuracy. These results inform us to select the largest sub-vector length that does not degrade accuracy.

These results show that with only 10-30 samples per class, LifeLearner achieve similar CL performance to Oracle, exhibit rapid convergence with small replay epochs (at most two), and accomplish a high compression rate for rehearsal samples.

6.4.5 MCU Deployment

TinyANML Architecture. For the extremely resource constrained IoT devices like MCUs where on-chip memory of SRAM and Flash are typically a few hundred KB or 1 MB at most (an order of magnitude smaller than Jetson Nano and Pi 3B+ in terms of memory), the memory requirements of the MetaCL methods, including LifeLearner, are prohibitively large. Thus, we propose a small and accurate TinyANML architecture designed for MCUs with tiny memory by experimenting with various width modifiers [154, 251, 267]. We identified widths of 0.2, 0.05, and 0.4 for the ANML architecture of CIFAR-100, MiniImageNet, and GSCv2, respectively.

MCU Implementation and Results. Backbone represents an inference-only feature extractor based on TFLM. On top of that, our hardware-aware systems are added incrementally: (1) Backpropagation Engine (Tiny ANML) and (2) Compression Module (Tiny LifeLearner). Table 6.3 shows the MCU deployment results based on STM32H747 in terms of accuracy, SRAM, Flash, latency, and energy consumption to learn a class with ten samples when continually learning ten classes.

Table 6.3: MCU deployment of the Backbone, tiny ANML, and tiny LifeLearner on STM32H747.

Dataset	System	Accuracy	SRAM	Flash	Latency	Energy
CIFAR-100	Backbone	-	75kB	428kB	561ms	128mJ
	Tiny ANML	0.176	185kB	691kB	579ms	134mJ
	Tiny LifeLearner	0.393	236kB	825kB	832ms	195mJ
Mini-ImageNet	Backbone	-	119kB	329kB	926ms	221mJ
	Tiny ANML	0.112	224kB	591kB	944ms	218mJ
	Tiny LifeLearner	0.301	281kB	725kB	1204ms	282mJ
GSCv2	Backbone	-	81kB	475kB	956ms	218mJ
	Tiny ANML	0.209	181kB	738kB	968ms	223mJ
	Tiny LifeLearner	0.534	212kB	806kB	1160ms	271mJ

Backpropagation Engine. As shown with Tiny ANML compared to inference-only Backbone, our Backpropagation Engine enables on-device CL with extremely small latency/energy overheads (e.g., 579ms vs. 561ms and 134mJ vs. 128mJ for CIFAR-100) while requiring only an additional 100KB SRAM and 260KB Flash.

Co-design of Our Algorithm and Hardware-aware System Implementation. Tiny LifeLearner not only largely prevents accuracy degradation compared to its original LifeLearner (see Table 6.2) but also maintains higher accuracy than ANML despite Tiny LifeLearner's model size being 24.1-1839× smaller than ANML. Tiny LifeLearner achieves significantly higher accuracy than Tiny ANML while having minimal resource requirements (e.g., 181-281kB SRAM, 725-825kB Flash, 832-1,204ms latency, and 195-282mJ energy consumption), demonstrating the effectiveness of our proposed algorithm and hardware-aware system implementation on such an extremely resource-constrained device.

Note that it is infeasible to perform the ablation study to quantify the benefits of our design as in Section 6.4.3. This is because other baselines with rehearsal strategy and prior works exhibit out-of-memory problems and only tiny LifeLearner could run on MCUs with severely limited memory.

6.5 Discussion

Impact on Continual Learning. We envision that LifeLearner could make CL a practical reality on embedded and IoT devices by leveraging meta-learning and rehearsal strategy with only a few samples. Such CL systems will allow DNNs to add new classes (e.g., adding new objects to an image recognition system, adding new keywords to a voice assistant) or new modalities (e.g., adding image recognition on top of a voice recognition authentication system) on the fly without relying on the cloud (i.e., no communication

costs). As one future direction, further optimizing LifeLearner to use stricter quantisation such as 1, 2, or 4 bits will be interesting.

Generalisability of LifeLearner. LifeLearner successfully works on three different datasets operating on two different modalities: image and audio, showing the generalisability of our framework. With the proliferation of smart spaces, such as smart homes and offices, LifeLearner can be used to learn the personal habits and preferences of users in order to control environmental conditions, such as temperature, humidity and lighting, with readings coming from thermometers, motion sensors and cameras on IoT devices. LifeLearner would enable this personalisation and space adaptivity to happen in a data-efficient manner and to stay local to ensure privacy. Moreover, LifeLearner could be used on robot vacuum cleaners to enhance their adaptability, e.g., to continually learn to visually recognise new objects and thus avoid collisions.

The evaluation of other datasets and potentially other modalities, including various other sensor signals [59, 272] as mentioned above to further test the applicability of LifeLearner for learning continually for other real-world applications, is left as future work.

Scalibility over Many Classes. The sample-wise compression ratio of LifeLearner is about 30×, significantly reducing the memory overhead of adding many classes. It incurs only 1.68 MB, 6.16 MB, and 0.66 MB of memory when adding 100 classes with 30 samples per class for CIFAR-100, MiniImageNet, and GSCv2, respectively. Also, our scalar quantisation and selective layer updates resolve scalability issues of latency as it incurs minimal latency overhead over ANML with fixed latency to learn new classes (see Tables 6.2 and 6.3).

Feasibility of Labelling Samples. One of the key challenges of enabling realistic applications for CL is annotation difficulty by users. As conventional CL typically demands a few thousand labelled samples, it becomes almost infeasible for users to perform labelling Instead, LifeLearner ameliorates this labelling burden by enabling data-efficient CL with 10-30 samples per class which are not impractical to label.

Other Considerations. In this work, our evaluation demonstrated that LifeLearner achieves near-optimal CL performance, falling short by only 2.8% accuracy compared to the upper bound system (Oracle). However, a higher accuracy (over 80-90%) given fewer samples (less than 10-30 samples) would be desirable. Thus, it is worth investigating larger and more advanced model architectures specializing in the target problem and task, such as Transformers [273, 274], to push the envelope of the upper bound testing accuracy of the challenging CL problem.

6.6 Conclusions

In this chapter, we proposed LifeLearner, a hardware-aware meta CL system with adaptive fast-slow weights and resource-optimised compression for embedded and IoT platforms. LifeLearner outperforms all existing Meta CL methods by a large margin (approximating the upper bound method that performs training in i.i.d. setting) and demonstrates its potential applicability in real-world deployments. Our efficient CL system opens the door to adaptive applications to run on embedded and IoT devices by allowing them to learn new tasks and adapt to the dynamics of the user and context.

Chapter 7

Final Remarks and Reflections

In this dissertation, we presented five original pieces of work investigating foundational challenges in efficient and adaptive CL systems in mobile computing. Our primary objective was to design algorithms and systems that enable CL and on-device training for mobile and IoT devices by drastically minimising resource requirements while maintaining high precision. This research addresses the growing need for adaptive capabilities in resource-constrained environments, paving the way for more intelligent and responsive mobile and IoT ecosystems.

7.1 Summary of Contributions

Our research has made several significant contributions to the field of CL and on-device training in mobile computing. First of all, we performed a comprehensive analysis of various representative CL methods in terms of performance, efficiency, and generalisability to have a better understanding of applying CL on multiple modalities of data on resource-constrained devices. Based on this analysis, we developed an efficient CL method, FastICARL, to minimise the resource overheads of CL on a smartphone (Chapter 3). Furthermore, we proposed YONO, a novel compression method. YONO reduces model size and enables efficient model execution and switching that can benefit multi-application setups (Chapter 4). In addition, we proposed TinyTrain, an efficient on-device training approach. TinyTrain drastically minimises the data, memory, and computation that can support efficient CL on extremely resource-constrained devices (Chapter 5). Finally, we developed LifeLearner, an efficient CL system that optimises data, memory, and compute efficiency operated on both embedded and IoT devices. (Chapter 6).

These contributions collectively address the research questions posed at the beginning of this dissertation, demonstrating the feasibility and effectiveness of CL and on-device

training in mobile computing environments. Our findings, methods, and systems can be used for the ubiquitous deployment of efficient CL and on-device training systems that are adaptive to users and environments continually over time within strict resource constraints.

7.2 Key Insights and Broader Implications

Beyond the specific technical contributions, this dissertation has produced several important insights that have broader implications for the field of mobile computing.

7.2.1 The Value of Co-Design in Resource-Constrained AI Systems

A fundamental insight emerging from our work is the critical importance of co-designing algorithms and systems with explicit awareness of hardware constraints. Throughout this dissertation, we consistently observed that approaches which naively apply traditional ML techniques to resource-constrained environments fail to achieve acceptable performance. Instead, each of our successful solutions—YONO, TinyTrain, and LifeLearner—was built on the principle of hardware-algorithm co-design.

This co-design principle manifested in different ways: task-adaptive sparse update in TinyTrain, efficient model compression and hardware-aware task switching in YONO, and hardware-aware system optimisation in LifeLearner. The success of these approaches demonstrates that treating resource constraints as primary design parameters, rather than secondary considerations, is essential for creating effective mobile AI systems. This insight extends beyond CL and on-device training to the broader field of mobile computing, suggesting that future advances in this field will come from holistic approaches that jointly optimise across the entire computing stack.

7.2.2 Balancing the Trilemma: Data, Memory, and Computation

Our research reveals a fundamental trilemma in mobile computing: simultaneously optimising for data efficiency, memory footprint, and computational cost. Throughout our work, we found that traditional approaches often excel in one dimension while sacrificing others. For instance, many CL methods achieve high accuracy but at the cost of prohibitive memory usage or computational overhead. Moreover, the scarcity of labelled data emerged as a particularly significant challenge in real-world deployments, as existing CL approaches often assume access to substantial amounts of labelled data – unrealistic in mobile scenarios where user labelling is limited.

The success of our TinyTrain and LifeLearner systems demonstrates that this trilemma can be addressed through techniques that adaptively balance these three dimensions based

on the specific characteristics of the task and device:

1. Task-adaptive sparse updates that reduce memory and computational requirements.
2. Efficient meta-learning strategies that enable high accuracy with fewer labelled examples.
3. Hardware-aware design principles that adapt to the target device capabilities.

These approaches make CL practical in resource-constrained environments where extensive user annotation is infeasible. Future mobile AI systems should adopt more flexible and balanced optimisation strategies rather than prioritising any single dimension of efficiency.

7.2.3 The Feasibility of Truly Ubiquitous Adaptive AI

When we began this research, the notion of deploying CL systems on extremely resource-constrained devices like MCUs seemed ambitious, if not impossible. Traditional wisdom held that such adaptive capabilities would require computational resources far beyond what is available in these tiny devices.

One of the most significant insights from our work is that, with careful system design and algorithmic innovation, truly ubiquitous adaptive AI is not only possible but practical. The successful deployment of LifeLearner on an MCU with only 512 KB of SRAM represents a significant milestone, demonstrating that even the most constrained edge devices can incorporate adaptability.

This finding has profound implications for the future of ubiquitous, mobile computing. It suggests that we are approaching an era where intelligence and adaptive capability can be embedded in virtually any connected device, from wearables to home appliances to industrial sensors. This *"democratisation"* of adaptive AI could lead to more personalised, responsive, and efficient systems across all domains of life.

7.3 Future Research Directions

While this dissertation has made significant steps in enabling CL and on-device training for mobile and IoT devices, several exciting avenues for future research remain:

- **Federated Continual Learning:** Investigating how our techniques can be extended to federated learning scenarios, where multiple devices collaboratively learn while preserving privacy. Note that this scenario involves challenges derived from federated setup on top of the system challenges addressed in this dissertation: limited system resources and user data. Thus, this could involve developing communication-efficient protocols for sharing model updates and exploring how to balance local adaptation with global model consistency.

- **Cross-Modal Continual Learning**: Extending our work to scenarios where systems need to continually learn across different data modalities (*e.g.,* vision, audio, and sensor data) simultaneously. This could be particularly relevant for multimodal mobile and IoT applications and could involve developing new techniques for efficient cross-modal knowledge transfer and representation learning.
- **Further Optimising Training Efficiency of Continual Learning and On-device Training:** While our work enhances the training efficiency of CL and on-device training, there exist many promising techniques that can be explored and incorporated to maximise the system efficiency. For instance, techniques such as Mixed Precision Training (MPT) [275], quantisation using even lower precision (such as using 2 or 3 bits) for weights and activations during the CL process, and applying pruning dynamically to a model during the CL process to reduce the model parameters might help improve the training efficiency in terms of its computational costs, memory footprints and latency.
- **Combining Binary Neural Networks with Continual Learning:** An interesting avenue for future work is the exploration of combining binary neural networks with CL methods. This approach could potentially offer extreme efficiency in terms of both storage and computation, while still allowing for continuous adaptation. Research in this area could involve developing novel training algorithms that maintain the benefits of binary networks while enabling effective CL.
- **Extending TinyTrain to Different Architectures and Applications**: Our evaluation of TinyTrain is currently limited to CNN-based architectures on vision tasks. Future work should aim to extend TinyTrain to different architectures such as Transformers and RNNs, and to a broader range of applications including segmentation and processing of audio or biological data. Additionally, exploring the application of TinyTrain to mobile-grade large language models on the edge could open up new possibilities for on-device natural language processing. This extension would validate the generalisability of TinyTrain's approach and potentially lead to breakthroughs in efficient on-device training across various domains.
- **Sustainable On-Device Training:** While on-device training avoids the excessive electricity consumption and carbon emissions associated with centralised training, it has traditionally been a significant drain on the battery life of edge devices. TinyTrain's energy efficiency improvements pave the way towards more sustainable on-device training. Future research should focus on further optimising energy consumption, potentially exploring techniques such as adaptive power management or energy-aware training scheduling. This could lead to the development of truly sustainable AI systems that can learn and adapt without compromising device longevity or environmental impact.

- **Automated Hardware-aware Optimisation:** Developing frameworks that can automatically adjust CL and training algorithms based on the specific hardware constraints of target devices. This could involve using reinforcement learning or other adaptive techniques to dynamically optimise the trade-offs between accuracy, memory usage, and energy consumption.

7.4 Closing Thoughts

This dissertation demonstrates a significant advancement in mobile and IoT computing–the shift from static, pre-trained models to adaptive, continually learning systems. Our work shows that devices can become learning companions that evolve with users' changing needs while operating within strict resource constraints.

Most significantly, we have demonstrated that these adaptive capabilities need not be limited to high-end devices. By addressing the core challenges of memory, computational, and data efficiency, we have shown that adaptive intelligence can be democratised across the spectrum of computing devices, from mobile/embedded systems to microcontrollers.

As the IoT continues to grow and mobile computing becomes increasingly ubiquitous, the techniques and systems developed in this dissertation will play a crucial role in creating more intelligent, efficient, and adaptive mobile and IoT ecosystems. The future of mobile computing lies in devices that can continuously learn and adapt to their environments and users, and this research represents a significant step towards realising that vision.

Bibliography

[1] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[2] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, koray kavukcuoglu, and Daan Wierstra. Matching Networks for One Shot Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*. 2016.

[3] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S. Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, April 2016.

[4] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent Trends in Deep Learning Based Natural Language Processing [Review Article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75, August 2018.

[5] Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T. Campbell. A Survey of Mobile Phone Sensing. *IEEE Communications Magazine*, 48(9):140–150, September 2010.

[6] Henry Friday Nweke, Ying Wah Teh, Mohammed Ali Al-garadi, and Uzoma Rita Alo. Deep learning algorithms for human activity recognition using mobile and wearable sensor networks: State of the art and research challenges. *Expert Systems with Applications*, 105:233–261, September 2018.

[7] Young D Kwon, Jagmohan Chauhan, Abhishek Kumar, Pan Hui, and Cecilia Mascolo. Exploring System Performance of Continual Learning for Mobile and Embedded Sensing Applications. In *ACM/IEEE Symposium on Edge Computing*. Association for Computing Machinery (ACM), 2021.

[8] Yu Guan and Thomas Plötz. Ensembles of Deep LSTM Learners for Activity Recognition Using Wearables. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(2):11:1–11:28, June 2017.

[9] Jindong Wang, Yiqiang Chen, Shuji Hao, Xiaohui Peng, and Lisha Hu. Deep

learning for sensor-based activity recognition: A survey. *Pattern Recognition Letters*, 119:3–11, March 2019.

[10] Junjun Fan, Xiangmin Fan, Feng Tian, Yang Li, Zitao Liu, Wei Sun, and Hongan Wang. What is That in Your Hand?: Recognizing Grasped Objects via Forearm Electromyography Sensing. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(4):161:1–161:24, December 2018.

[11] Yifei Jiang, Xin Pan, Kun Li, Qin Lv, Robert P. Dick, Michael Hannigan, and Li Shang. ARIEL: Automatic Wi-fi Based Room Fingerprinting for Indoor Localization. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 441–450, 2012.

[12] Hong Lu, Denise Frauendorfer, Mashfiqui Rabbi, Marianne Schmid Mast, Gokul T. Chittaranjan, Andrew T. Campbell, Daniel Gatica-Perez, and Tanzeem Choudhury. StressSense: detecting stress in unconstrained acoustic environments using smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, page 351–360, New York, NY, USA, 2012.

[13] Akhil Mathur, Nadia Berthouze, and Nicholas D. Lane. Unsupervised Domain Adaptation Under Label Space Mismatch for Speech Classification. In *Proc INTERSPEECH*, pages 1271–1275, October 2020.

[14] Ashish Mittal, Samarth Bharadwaj, Shreya Khare, Saneem Chemmengath, Karthik Sankaranarayanan, and Brian Kingsbury. Representation Based Meta-Learning for Few-Shot Spoken Intent Recognition. In *Proc INTERSPEECH*, pages 4283–4287, October 2020.

[15] Kiran K. Rachuri, Mirco Musolesi, Cecilia Mascolo, Peter J. Rentfrow, Chris Longworth, and Andrius Aucinas. EmotionSense: a mobile phones based adaptive platform for experimental social psychology research. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, UbiComp '10, page 281–290, New York, NY, USA, 2010.

[16] Hendrik Purwins, Bo Li, Tuomas Virtanen, Jan Schlüter, Shuo-Yiin Chang, and Tara Sainath. Deep Learning for Audio Signal Processing. *IEEE Journal of Selected Topics in Signal Processing*, 13(2):206–219, May 2019.

[17] Jagmohan Chauhan, Young D. Kwon, Pan Hui, and Cecilia Mascolo. ContAuth: Continual Learning Framework for Behavioral-Based User Authentication. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(4), December 2020.

[18] Sinno Jialin Pan and Qiang Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(10):1345–1359, 2010.

[19] Taesik Gong, Yeonsu Kim, Jinwoo Shin, and Sung-Ju Lee. MetaSense: Few-shot Adaptation to Untrained Conditions in Deep Mobile Sensing. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, pages 110–123, New York, NY, USA, 2019. ACM. event-place: New York, New York.

[20] Ian J. Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks. *arXiv:1312.6211 [cs, stat]*, December 2013.

[21] B. Pfülb and A. Gepperth. A comprehensive, application-oriented study of catastrophic forgetting in DNNs. In *ICLR*, 2019.

[22] In Gim and JeongGil Ko. Memory-Efficient DNN Training on Mobile Devices. In *Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2022.

[23] Young D. Kwon, Rui Li, Stylianos Venieris, Jagmohan Chauhan, Nicholas Donald Lane, and Cecilia Mascolo. TinyTrain: Resource-Aware Task-Adaptive Sparse Training of DNNs at the Data-Scarce Edge. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 25812–25843. PMLR, 21–27 Jul 2024.

[24] Daliang Xu, Mengwei Xu, Qipeng Wang, Shangguang Wang, Yun Ma, Kang Huang, Gang Huang, Xin Jin, and Xuanzhe Liu. Mandheling: Mixed-Precision On-Device DNN Training with DSP Offloading. In *Annual International Conference on Mobile Computing And Networking (MobiCom)*, 2022.

[25] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine Learning at the Network Edge: A Survey. *ACM Comput. Surv.*, 54(8), October 2021.

[26] Jiasi Chen and Xukan Ran. Deep Learning With Edge Computing: A Review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[27] Michael McCloskey and Neal J. Cohen. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. In *Psychology of Learning and Motivation*, volume 24, pages 109–165. January 1989.

[28] James L. McClelland, Bruce L. McNaughton, and Randall C. O'Reilly. Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory. *Psychological Review*, 102(3):419–457, 1995.

[29] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan

Wermter. Continual Lifelong Learning with Neural Networks: A Review. *Neural Networks*, 113:54–71, 2019.

[30] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proc. National Academy of Sciences*, 114(13):3521–3526, March 2017.

[31] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. iCaRL: Incremental Classifier and Representation Learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[32] Tyler L. Hayes, Kushal Kafle, Robik Shrestha, Manoj Acharya, and Christopher Kanan. REMIND Your Neural Network to Prevent Catastrophic Forgetting. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 466–483, Cham, 2020. Springer International Publishing.

[33] Nils Y. Hammerla, Shane Halloran, and Thomas Plötz. Deep, Convolutional, and Recurrent Models for Human Activity Recognition Using Wearables. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 1533–1540, 2016.

[34] Young D. Kwon, Kirill A. Shatilov, Lik-Hang Lee, Serkan Kumyol, Kit-Yung Lam, Yui-Pan Yau, and Pan Hui. MyoKey: Surface Electromyography and Inertial Motion Sensing-based Text Entry in AR. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 1–4, March 2020.

[35] Eugene Lee, Cheng-Han Huang, and Chen-Yi Lee. Few-Shot and Continual Learning With Attentive Independent Mechanisms. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9455–9464, October 2021.

[36] Shell Xu Hu, Da Li, Jan Stühmer, Minyoung Kim, and Timothy M. Hospedales. Pushing the Limits of Simple Pipelines for Few-Shot Learning: External data and fine-tuning make a difference. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[37] Seulki Lee and Shahriar Nirjon. Learning in the Wild: When, How, and What to Learn for On-Device Dataset Adaptation. In *International Workshop on Challenges in*

Artificial Intelligence and Machine Learning for Internet of Things (AIChallengeIoT), 2020.

[38] Haoyu Ren, Darko Anicic, and Thomas A. Runkler. TinyOL: TinyML with Online-Learning on Microcontrollers. In *International Joint Conference on Neural Networks (IJCNN)*, 2021.

[39] Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998.

[40] Ameya Prabhu, Hasan Abed Al Kader Hammoud, Puneet K. Dokania, Philip H.S. Torr, Ser-Nam Lim, Bernard Ghanem, and Adel Bibi. Computationally budgeted continual learning: What does matter? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3698–3707, June 2023.

[41] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost, 2016.

[42] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging. In *International Conference on Machine Learning (ICML)*, 2022.

[43] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. Melon: Breaking the Memory Wall for Resource-Efficient On-Device Machine Learning. In *Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2022.

[44] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-Device Training Under 256KB Memory. In *Advances on Neural Information Processing Systems (NeurIPS)*, 2022.

[45] Aaqib Saeed, Tanir Ozcelebi, and Johan Lukkien. Multi-task Self-Supervised Learning for Human Activity Detection. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(2):61:1–61:30, June 2019.

[46] Andreas Bulling, Ulf Blanke, and Bernt Schiele. A Tutorial on Human Activity Recognition Using Body-worn Inertial Sensors. *ACM Comput. Surv.*, 46(3):33:1–33:33, January 2014.

[47] Xiaolong Zhai, Beth Jelfs, Rosa H. M. Chan, and Chung Tin. Self-Recalibrating Surface EMG Pattern Recognition for Neuroprosthesis Control Based on Convolutional Neural Network. *Frontiers in Neuroscience*, 11, 2017.

[48] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of

Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[49] Xiaoxi He, Zimu Zhou, and Lothar Thiele. Multi-Task Zipping via Layer-wise Neuron Sharing. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[50] H. Jégou, M. Douze, and C. Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, January 2011.

[51] Young D. Kwon, Jagmohan Chauhan, and Cecilia Mascolo. FastICARL: Fast Incremental Classifier and Representation Learning with Efficient Budget Allocation in Audio Sensing Applications. In *Proc. Interspeech 2021*, pages 356–360, 2021.

[52] Young D. Kwon, Jagmohan Chauhan, and Cecilia Mascolo. YONO: Modeling Multiple Heterogeneous Neural Networks on Microcontrollers. In *2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 285–297, 2022.

[53] Young D. Kwon, Jagmohan Chauhan, Hong Jia, Stylianos I. Venieris, and Cecilia Mascolo. LifeLearner: Hardware-Aware Meta Continual Learning System for Embedded Computing Platforms. In *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*, SenSys '23, page 138–151, New York, NY, USA, 2024. Association for Computing Machinery.

[54] Hong Jia, Young Kwon, Alessio Orsino, Ting Dang, DOMENICO TALIA, and Cecilia Mascolo. TinyTTA: Efficient Test-time Adaptation via Early-exit Ensembles on Edge Devices. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 43274–43299. Curran Associates, Inc., 2024.

[55] Hong Jia, Young D. Kwon, Dong Mat, Nhat Pham, Lorena Qendro, Tam Vu, and Cecilia Mascolo. UR2M: Uncertainty and Resource-Aware Event Detection on Microcontrollers. In *2024 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–10, 2024.

[56] Reza Hadi Mogavi, Chao Deng, Justin Juho Kim, Pengyuan Zhou, Young D. Kwon, Ahmed Hosny Saleh Metwally, Ahmed Tlili, Simone Bassanelli, Antonio Bucchiarone, Sujit Gujar, Lennart E. Nacke, and Pan Hui. ChatGPT in education: A blessing or a curse? A qualitative study exploring early adopters' utilization and perceptions. *Computers in Human Behavior: Artificial Humans*, 2(1):100027, 2024.

[57] Anish Das, Young D. Kwon, Jagmohan Chauhan, and Cecilia Mascolo. Enabling

On-Device Smartphone GPU based Training: Lessons Learned. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 533–538, 2022.

[58] Jagmohan Chauhan, Young D. Kwon, and Cecilia Mascolo. Exploring On-Device Learning Using Few Shots for Audio Classification. In *2022 30th European Signal Processing Conference (EUSIPCO)*, pages 424–428, 2022.

[59] Nhat Pham, Hong Jia, Minh Tran, Tuan Dinh, Nam Bui, Young Kwon, Dong Ma, Phuc Nguyen, Cecilia Mascolo, and Tam Vu. PROS: An Efficient Pattern-Driven Compressive Sensing Framework for Low-Power Biopotential-Based Wearables with on-Chip Intelligence. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, MobiCom '22, page 661–675, New York, NY, USA, 2022. Association for Computing Machinery.

[60] K. Shatilov, Y. D. Kwon, L. Lee, D. Chatzopoulos, and P. Hui. MyoKey: Inertial Motion Sensing and Gesture-based QWERTY Keyboard for Extended Realities. *IEEE Transactions on Mobile Computing*, (01):1–1, mar 5555.

[61] A. K. Vallapuram, Y. D. Kwon, L. Lee, F. Xu, and P. Hui. Causal Analysis on the Anchor Store Effect in a Location-based Social Network. In *2022 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 202–209, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[62] Anish K. Vallapuram, Pengyuan Zhou, Young D. Kwon, Lik Hang Lee, Hengwei Xu, and Pan Hui. HideNseek: Federated Lottery Ticket via Server-side Pruning and Sign Supermask, 2022.

[63] Abhishek Kumar, Tristan Braud, Young D. Kwon, and Pan Hui. Aquilis: Using Contextual Integrity for Privacy Protection on Mobile Devices. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(4), December 2020.

[64] Anish K. Vallapuram, Nikhil Nanda, Young D. Kwon, and Pan Hui. Interpretable Business Survival Prediction. In *Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '21, pages 99–106, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Liang-yu Chen, Yutong Chen, Young D. Kwon, Youwen Kang, and Pan Hui. IAN: Interpretable Attention Network for Churn Prediction in LBSNs. In *Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '21, pages 23–30, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Sandra Servia-Rodriguez, Cecilia Mascolo, and Young D. Kwon. Knowing when

we do not know: Bayesian continual learning for sensing-based analysis tasks. *arXiv:2106.05872 [cs]*, June 2021.

[67] Biyi Fang, Xiao Zeng, and Mi Zhang. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, pages 115–127, 2018.

[68] Angkoon Phinyomark and Erik Scheme. EMG Pattern Recognition in the Era of Big Data and Deep Learning. *Big Data and Cognitive Computing*, 2(3):21, September 2018.

[69] Nicholas D. Lane, Petko Georgiev, and Lorena Qendro. DeepEar: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, page 283–294, New York, NY, USA, 2015. Association for Computing Machinery.

[70] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

[71] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-Time Video Analytics: The Killer App for Edge Computing. *Computer*, 50(10):58–67, 2017.

[72] Fangxin Wang, Miao Zhang, Xiangxiang Wang, Xiaoqiang Ma, and Jiangchuan Liu. Deep Learning for Edge Computing Applications: A State-of-the-Art Survey. *IEEE Access*, 8:58322–58336, 2020.

[73] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[74] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The German Traffic Sign Recognition Benchmark: A Multi-Class Classification Competition. In *International Joint Conference on Neural Networks (IJCNN)*, 2011.

[75] Ju Ren, Yundi Guo, Deyu Zhang, Qingqing Liu, and Yaoxue Zhang. Distributed and Efficient Object Detection in Edge Computing: Challenges and Solutions. *IEEE Network*, 32(6):137–143, 2018.

[76] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[77] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. LAVEA: latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[78] Bichen Wu, Alvin Wan, Forrest Iandola, Peter H. Jin, and Kurt Keutzer. SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 446–454, 2017.

[79] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3D Object Detection Network for Autonomous Driving. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6526–6534, 2017.

[80] Ming Zeng, Haoxiang Gao, Tong Yu, Ole J. Mengshoel, Helge Langseth, Ian Lane, and Xiaobing Liu. Understanding and Improving Recurrent Networks for Human Activity Recognition by Continuous Attention. In *Proceedings of the 2018 ACM International Symposium on Wearable Computers*, ISWC '18, pages 56–63, 2018.

[81] Vishvak S. Murahari and Thomas Plötz. On attention models for human activity recognition. In *Proceedings of the 2018 ACM International Symposium on Wearable Computers*, ISWC '18, page 100–103, New York, NY, USA, 2018. Association for Computing Machinery.

[82] Xin Qin, Jindong Wang, Yiqiang Chen, Wang Lu, and Xinlong Jiang. Domain Generalization for Activity Recognition via Adaptive Feature Fusion. *ACM Trans. Intell. Syst. Technol.*, 14(1), November 2022.

[83] Junru Zhang, Lang Feng, Zhidan Liu, Yuhan Wu, Yang He, Yabo Dong, and Duanqing Xu. Diverse Intra- and Inter-Domain Activity Style Fusion for Cross-Person Generalization in Activity Recognition. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 4213–4222, New York, NY, USA, 2024. Association for Computing Machinery.

[84] Francisco Javier Ordóñez and Daniel Roggen. Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition. *Sensors*, 16(1):115, January 2016.

[85] Christoph Amma, Thomas Krings, Jonas Böer, and Tanja Schultz. Advancing Muscle-Computer Interfaces with High-Density Electromyography. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 929–938, 2015.

[86] Faizan Haque, Mathieu Nancel, and Daniel Vogel. Myopoint: Pointing and Clicking Using Forearm Mounted Electromyography and Inertial Motion Sensors. In *Proceed-*

ings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI '15, pages 3653–3656, 2015.

[87] Vincent Becker, Pietro Oldrati, Liliana Barrios, and Gábor Sörös. Touchsense: Classifying Finger Touches and Measuring Their Force with an Electromyography Armband. In *Proceedings of the 2018 ACM International Symposium on Wearable Computers*, ISWC '18, pages 1–8, 2018.

[88] Jamileh Yousefi and Andrew Hamilton-Wright. Characterizing EMG data using machine-learning tools. *Computers in Biology and Medicine*, 51:1–13, August 2014.

[89] Brent D. Winslow, Mitchell Ruble, and Zachary Huber. Mobile, Game-Based Training for Myoelectric Prosthesis Control. *Frontiers in Bioengineering and Biotechnology*, 6, 2018.

[90] T. Scott Saponas, Desney S. Tan, Dan Morris, Ravin Balakrishnan, Jim Turner, and James A. Landay. Enabling Always-available Input with Muscle-computer Interfaces. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 167–176, 2009.

[91] Erik Scheme and Kevin Englehart. Electromyogram pattern recognition for control of powered upper-limb prostheses: state of the art and challenges for clinical use. *Journal of Rehabilitation Research and Development*, 48(6):643–659, 2011.

[92] Kirill A. Shatilov, Dimitris Chatzopoulos, Alex Wong Tat Hang, and Pan Hui. Using Deep Learning and Mobile Offloading to Control a 3D-printed Prosthetic Hand. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 3(3):102:1–102:19, September 2019.

[93] Beomjune Shin, Sung Hoon Lee, Kangkyu Kwon, Yoon Jae Lee, Nikita Crispe, So-Young Ahn, Sandeep Shelly, Nathaniel Sundholm, Andrew Tkaczuk, Min-Kyung Yeo, Hyojung J. Choo, and Woon-Hong Yeo. Automatic Clinical Assessment of Swallowing Behavior and Diagnosis of Silent Aspiration Using Wireless Multimodal Wearable Electronics. *Advanced Science*, 11(34):2404211, 2024. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/advs.202404211.

[94] Hong Lu, A. J. Bernheim Brush, Bodhi Priyantha, Amy K. Karlson, and Jie Liu. SpeakerSense: energy efficient unobtrusive speaker identification on mobile phones. In *Proceedings of the 9th international conference on Pervasive computing*, Pervasive'11, pages 188–205, San Francisco, USA, June 2011.

[95] Chenren Xu, Sugang Li, Gang Liu, Yanyong Zhang, Emiliano Miluzzo, Yih-Farn Chen, Jun Li, and Bernhard Firner. Crowd++: unsupervised speaker count with smartphones. In *Proceedings of the 2013 ACM international joint conference on*

Pervasive and ubiquitous computing, UbiComp '13, pages 43–52, Zurich, Switzerland, September 2013.

[96] Yu Su, Ke Zhang, Jingyu Wang, and Kurosh Madani. Environment Sound Classification Using a Two-Stream CNN Based on Decision-Level Fusion. *Sensors*, 19(7):1733, January 2019.

[97] Youngki Lee, Chulhong Min, Chanyou Hwang, Jaeung Lee, Inseok Hwang, Younghyun Ju, Chungkuk Yoo, Miri Moon, Uichin Lee, and Junehwa Song. SocioPhone: everyday face-to-face interaction monitoring platform using multi-phone sensor fusion. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, MobiSys '13, pages 375–388, Taipei, Taiwan, June 2013.

[98] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello Edge: Keyword Spotting on Microcontrollers. *arXiv:1711.07128 [cs, eess]*, November 2017.

[99] Guoguo Chen, Carolina Parada, and Georg Heigold. Small-footprint keyword spotting using deep neural networks. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4087–4091. IEEE, 2014.

[100] Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. SoundSense: scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, MobiSys '09, pages 165–178, Kraków, Poland, June 2009.

[101] Petko Georgiev, Sourav Bhattacharya, Nicholas D. Lane, and Cecilia Mascolo. Low-resource Multi-task Audio Sensing for Mobile and Embedded Devices via Shared Deep Neural Network Representations. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(3), September 2017.

[102] Sung Hoon Lee, Yoon Jae Lee, Kangkyu Kwon, Daniel Lewis, Lissette Romero, Jimin Lee, Nathan Zavanelli, Emily Yan, Ki Jun Yu, and Woon-Hong Yeo. Soft Smart Biopatch for Continuous Authentication-Enabled Cardiac Biometric Systems. *Advanced Sensor Research*, 2(12):2300074, 2023. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/adsr.202300074.

[103] Saurav Jha, Martin Schiemer, Franco Zambonelli, and Juan Ye. Continual learning in sensor-based human activity recognition: An empirical benchmark analysis. *Information Sciences*, 575:1–21, October 2021.

[104] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Alex; Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. A Continual Learning Survey: Defying Forgetting in Classification Tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 44(7):3366–3385, 2022.

[105] Da-Wei Zhou, Qi-Wei Wang, Zhi-Hong Qi, Han-Jia Ye, De-Chuan Zhan, and Ziwei Liu. Class-Incremental Learning: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(12):9851–9873, 2024.

[106] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A Comprehensive Survey of Continual Learning: Theory, Method and Application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(8):5362–5383, 2024.

[107] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual Learning Through Synaptic Intelligence. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3987–3995. PMLR, 06–11 Aug 2017.

[108] Jonathan Schwarz, Wojciech Czarnecki, Jelena Luketina, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & Compress: A scalable framework for continual learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4528–4537. PMLR, 10–15 Jul 2018.

[109] Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory Aware Synapses: Learning what (not) to forget. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[110] Arslan Chaudhry, Puneet K. Dokania, Thalaiyasingam Ajanthan, and Philip H. S. Torr. Riemannian Walk for Incremental Learning: Understanding Forgetting and Intransigence. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[111] Cuong V. Nguyen, Yingzhen Li, Thang D. Bui, and Richard E. Turner. Variational Continual Learning. In *International Conference on Learning Representations*, 2018.

[112] David Lopez-Paz and Marc' Aurelio Ranzato. Gradient Episodic Memory for Continual Learning. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[113] Lorenzo Pellegrini, Gabriele Graffieti, Vincenzo Lomonaco, and Davide Maltoni. Latent Replay for Real-Time Continual Learning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10203–10209, 2020.

[114] Jagmohan Chauhan, Young D. Kwon, Pan Hui, and Cecilia Mascolo. ContAuth: Continual Learning Framework for Behavioral-based User Authentication. *Proc. IMWUT*, 4(4):122:1–122:23, December 2020.

[115] Francisco M. Castro, Manuel J. Marin-Jimenez, Nicolas Guil, Cordelia Schmid, and Karteek Alahari. End-to-End Incremental Learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[116] Yue Wu, Yinpeng Chen, Lijuan Wang, Yuancheng Ye, Zicheng Liu, Yandong Guo, and Yun Fu. Large Scale Incremental Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[117] Sudhanshu Mittal, Silvio Galesso, and Thomas Brox. Essentials for class incremental learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2021.

[118] Z. Li and D. Hoiem. Learning without Forgetting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(12):2935–2947, December 2018.

[119] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual Learning with Deep Generative Replay. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 2990–2999. 2017.

[120] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks, 2014.

[121] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[122] Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong Learning with Dynamically Expandable Networks. In *International Conference on Learning Representations*, 2018.

[123] Ching-Yi Hung, Cheng-Hao Tu, Cheng-En Wu, Chien-Hung Chen, Yi-Ming Chan, and Chu-Song Chen. Compacting, Picking and Growing for Unforgetting Continual Learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[124] Shipeng Yan, Jiangwei Xie, and Xuming He. DER: Dynamically Expandable Representation for Class Incremental Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3014–3023, June 2021.

[125] Da-Wei Zhou, Qi-Wei Wang, Han-Jia Ye, and De-Chuan Zhan. A Model or 603 Exemplars: Towards Memory-Efficient Class-Incremental Learning. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023.

[126] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng

Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lyu, and Zhihua Wu. MNN: A Universal and Efficient Inference Engine. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 1–13, 2020.

[127] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 800–811, 2021.

[128] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L Hayes, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[129] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-Learning in Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 44(9):5149–5169, 2022.

[130] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *International Conference on Machine Learning (ICML)*, 2017.

[131] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your MAML. In *International Conference on Learning Representations*, 2019.

[132] Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. Meta-SGD: Learning to Learn Quickly for Few-Shot Learning. *arXiv:1707.09835 [cs]*, 2017.

[133] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical Networks for Few-shot Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.

[134] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip H. S. Torr, and Timothy M. Hospedales. Learning to Compare: Relation Network for Few-Shot Learning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[135] Victor Garcia Satorras and Joan Bruna Estrach. Few-Shot Learning with Graph Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[136] Xueting Zhang, Debin Meng, Henry Gouk, and Timothy M. Hospedales. Shallow Bayesian Meta Learning for Real-World Few-Shot Recognition. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[137] Yunhui Guo, Noel C. Codella, Leonid Karlinsky, James V. Codella, John R. Smith,

Kate Saenko, Tajana Rosing, and Rogerio Feris. A Broader Study of Cross-Domain Few-Shot Learning. In *European Conference on Computer Vision (ECCV)*, 2020.

[138] Eleni Triantafillou, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Utku Evci, Kelvin Xu, Ross Goroshin, Carles Gelada, Kevin Swersky, Pierre-Antoine Manzagol, and Hugo Larochelle. Meta-Dataset: A Dataset of Datasets for Learning to Learn from Few Examples. In *International Conference on Learning Representations (ICLR)*, 2020.

[139] Shawn Beaulieu, Lapo Frati, Thomas Miconi, Joel Lehman, Kenneth O Stanley, Jeff Clune, and Nick Cheney. Learning to Continually Learn. In *ECAI 2020*, pages 992–1001. IOS Press, 2020.

[140] Khurram Javed and Martha White. Meta-learning representations for continual learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[141] Chi Zhang, Nan Song, Guosheng Lin, Yun Zheng, Pan Pan, and Yinghui Xu. Few-Shot Incremental Learning With Continually Evolved Classifiers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12455–12464, June 2021.

[142] Ali Cheraghian, Shafin Rahman, Pengfei Fang, Soumava Kumar Roy, Lars Petersson, and Mehrtash Harandi. Semantic-Aware Knowledge Distillation for Few-Shot Class-Incremental Learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2534–2543, June 2021.

[143] Xiaoyu Tao, Xiaopeng Hong, Xinyuan Chang, Songlin Dong, Xing Wei, and Yihong Gong. Few-Shot Class-Incremental Learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12180–12189, Seattle, WA, USA, June 2020. IEEE.

[144] Mengmi Zhang, Tao Wang, Joo Hwee Lim, Gabriel Kreiman, and Jiashi Feng. Variational Prototype Replays for Continual Learning. *arXiv:1905.09447 [cs]*, February 2020.

[145] Bernd Fritzke. A Growing Neural Gas Network Learns Topologies. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7. MIT Press, 1994.

[146] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level Concept Learning through Probabilistic Program Induction. *Science*, 350(6266):1332–1338, 2015.

[147] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized Convolutional Neural Networks for Mobile Devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[148] Mingxing Tan and Quoc Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, May 2019.

[149] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:1602.07360 [cs]*, November 2016.

[150] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, and Kurt Keutzer. SqueezeNext: Hardware-Aware Neural Network Design. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[151] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, June 2018.

[152] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *European Conference on Computer Vision (ECCV)*, 2018.

[153] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv:1704.04861 [cs]*, April 2017.

[154] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[155] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations*, 2019.

[156] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[157] Mingxing Tan and Quoc Le. EfficientNetV2: Smaller Models and Faster Training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International*

Conference on Machine Learning, volume 139 of *Proceedings of Machine Learning Research*, pages 10096–10106. PMLR, 18–24 Jul 2021.

[158] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations (ICLR)*, 2020.

[159] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations (ICLR)*, 2019.

[160] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[161] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[162] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas Donald Lane. Zero-Cost Proxies for Lightweight {NAS}. In *International Conference on Learning Representations (ICLR)*, 2021.

[163] Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[164] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *International Conference on Learning Representations (ICLR)*, 2016.

[165] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[166] Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7765–7773, 2018.

[167] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye.

AutoCompress: An Automatic DNN Structured Pruning Framework for Ultra-High Compression Rates. *AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[168] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning Filters for Efficient ConvNets. November 2016.

[169] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. A Systematic DNN Weight Pruning Framework using Alternating Direction Method of Multipliers. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[170] Hongjia Li, Ning Liu, Xiaolong Ma, Sheng Lin, Shaokai Ye, Tianyun Zhang, Xue Lin, Wenyao Xu, and Yanzhi Wang. ADMM-based Weight Pruning for Real-Time Deep Learning Acceleration on Mobile Devices. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, pages 501–506, New York, NY, USA, May 2019. Association for Computing Machinery.

[171] Tianyun Zhang, Shaokai Ye, Xiaoyu Feng, Xiaolong Ma, Kaiqi Zhang, Zhengang Li, Jian Tang, Sijia Liu, Xue Lin, Yongpan Liu, Makan Fardad, and Yanzhi Wang. StructADMM: Achieving Ultrahigh Efficiency in Structured Pruning for DNNs. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–15, 2021.

[172] Ao Ren, Tianyun Zhang, Shaokai Ye, Jiayu Li, Wenyao Xu, Xuehai Qian, Xue Lin, and Yanzhi Wang. ADMM-NN: An Algorithm-Hardware Co-Design Framework of DNNs Using Alternating Direction Methods of Multipliers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 925–938, New York, NY, USA, April 2019. Association for Computing Machinery.

[173] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained Ternary Quantization. In *International Conference on Learning Representations (ICLR)*, 2017.

[174] Fengfu Li, Bo Zhang, and Bin Liu. Ternary Weight Networks. *arXiv:1605.04711 [cs]*, November 2016.

[175] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *Advances in Neural Information Processing Systems*, 28, 2015.

[176] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv:1602.02830 [cs]*, March 2016.

[177] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-

Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *European Conference on Computer Vision (ECCV)*, 2016.

[178] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D. Lane, and Yarin Gal. A Systematic Study of Binary Neural Networks' Optimisation. In *International Conference on Learning Representations*, 2019.

[179] Diwen Wan, Fumin Shen, Li Liu, Fan Zhu, Jie Qin, Ling Shao, and Heng Tao Shen. TBN: Convolutional Neural Network with Ternary Inputs and Binary Weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[180] Shihui Yin, Zhewei Jiang, Jae-Sun Seo, and Mingoo Seok. XNOR-SRAM: In-Memory Computing SRAM Macro for Binary/Ternary Deep Neural Networks. *IEEE Journal of Solid-State Circuits*, 55(6):1733–1743, June 2020.

[181] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing Neural Networks with the Hashing Trick. In *International Conference on Machine Learning*, pages 2285–2294. PMLR, June 2015.

[182] Jingyuan Zhao, Zhang Sihao, and Zeng Jing. Review of the sparse coding and the applications on image retrieval. In *2016 International Conference on Communication and Electronics Systems (ICCES)*, pages 1–5, October 2016.

[183] Tiezheng Ge, Kaiming He, and Jian Sun. Product Sparse Coding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[184] Hessam Bagherinezhad, Mohammad Rastegari, and Ali Farhadi. LCNN: Lookup-Based Convolutional Neural Network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[185] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing Deep Convolutional Networks using Vector Quantization. *arXiv:1412.6115 [cs]*, December 2014.

[186] Ting Chen, Lala Li, and Yizhou Sun. Differentiable Product Quantization for End-to-End Embedding Compression. In *International Conference on Machine Learning*, pages 1617–1626. PMLR, November 2020.

[187] Y. Kalantidis and Y. Avrithis. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2329–2336, June 2014.

[188] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized Product Quantization.

IEEE Transactions on Pattern Analysis and Machine Intelligence, 36(4):744–755, April 2014.

[189] Seulki Lee and Shahriar Nirjon. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, MobiSys '20, pages 175–190, New York, NY, USA, June 2020. Association for Computing Machinery.

[190] Yousun Ko, Alex Chadwick, Daniel Bates, and Robert Mullins. Lane Compression: A Lightweight Lossless Compression Method for Machine Learning on Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 20(2):16:1–16:26, March 2021.

[191] Rich Caruana. Multitask Learning. *Machine Learning*, 28(1):41–75, July 1997.

[192] Lei Han and Yu Zhang. Multi-stage multi-task learning with reduced rank. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[193] Andrew M McDonald, Massimiliano Pontil, and Dimitris Stamos. Spectral k-Support Norm Regularization. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[194] Wu Liu, Tao Mei, Yongdong Zhang, Cherry Che, and Jiebo Luo. Multi-task deep visual-semantic embedding for video thumbnail selection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3707–3715, 2015.

[195] Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3994–4003, 2016.

[196] Lei Han and Yu Zhang. Learning multi-level task groups in multi-task learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[197] Zhuoliang Kang, Kristen Grauman, and Fei Sha. Learning with Whom to Share in Multi-task Feature Learning. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 521–528, New York, NY, USA, June 2011. ACM.

[198] Giwoong Lee, Eunho Yang, and Sung Hwang. Asymmetric Multi-task Learning Based on Task Relatedness and Loss. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 230–238, New York, New York, USA, 20–22 Jun 2016. PMLR.

[199] Mingsheng Long, ZHANGJIE CAO, Jianmin Wang, and Philip S Yu. Learning

Multiple Tasks with Multilinear Relationship Networks. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[200] Raghuraman Krishnamoorthi. Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper. *arXiv:1806.08342 [cs, stat]*, 2018.

[201] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. Band: Coordinated Multi-DNN Inference on Heterogeneous Mobile Processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 235–247, New York, NY, USA, 2022. Association for Computing Machinery.

[202] Neiwen Ling, Kai Wang, Yuze He, Guoliang Xing, and Daqi Xie. RT-MDL: Supporting Real-Time Mixed Deep Learning Tasks on Edge Platforms. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, page 1–14, New York, NY, USA, 2021. Association for Computing Machinery.

[203] Neiwen Ling, Kai Wang, Yuze He, Guoliang Xing, and Daqi Xie. RT-MDL: Supporting Real-Time Mixed Deep Learning Tasks on Edge Platforms. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, page 1–14, New York, NY, USA, 2021. Association for Computing Machinery.

[204] Shuochao Yao, Jinyang Li, Dongxin Liu, Tianshi Wang, Shengzhong Liu, Huajie Shao, and Tarek Abdelzaher. Deep Compressive Offloading: Speeding up Neural Network Inference by Trading Edge Computation for Network Latency. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems (SenSys)*, 2020.

[205] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-Memory Neural Network Training: A Technical Report. *arXiv*, 2019.

[206] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael W Mahoney, and Joseph E Gonzalez. ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training. In *International Conference on Machine Learning (ICML)*, 2021.

[207] Zizheng Pan, Peng Chen, Haoyu He, Jing Liu, Jianfei Cai, and Bohan Zhuang. Mesa: A Memory-saving Training Framework for Transformers. *arXiv preprint arXiv:2111.11124*, 2021.

[208] R David Evans and Tor Aamodt. AC-GC: Lossy Activation Compression with Guaranteed Convergence. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[209] Xiaoxuan Liu, Lianmin Zheng, Dequan Wang, Yukuo Cen, Weize Chen, Xu Han, Jianfei Chen, Zhiyuan Liu, Jie Tang, Joey Gonzalez, Michael Mahoney, and Alvin Cheung. GACT: Activation Compressed Training for Generic Network Architectures. In *International Conference on Machine Learning (ICML)*, 2022.

[210] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Conference on Machine Learning and Systems (MLSys)*, 2020.

[211] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic Tensor Rematerialization. In *International Conference on Learning Representations (ICLR)*, 2021.

[212] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *International Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[213] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[214] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks. *ACM SIGPLAN Notices*, 53(1), 2018.

[215] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. TinyTL: Reduce Memory, Not Parameters for Efficient On-Device Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[216] Christos Profentzas, Magnus Almgren, and Olaf Landsiedel. MiniLearn: On-Device Learning for Low-Power IoT Devices. In *International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2022.

[217] Zhongnan Qu, Zimu Zhou, Yongxin Tong, and Lothar Thiele. P-Meta: Towards On-Device Deep Model Adaptation. In *28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2022.

[218] Yue Wang, Ziyu Jiang, Xiaohan Chen, Pengfei Xu, Yang Zhao, Yingyan Lin, and Zhangyang Wang. E2-Train: Training State-of-the-art CNNs with Over 80% Energy Savings. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and

R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[219] Monika Schak and Alexander Gepperth. A Study on Catastrophic Forgetting in Deep LSTM Networks. In Igor V. Tetko, Věra Kůrková, Pavel Karpov, and Fabian Theis, editors, *Artificial Neural Networks and Machine Learning – ICANN 2019: Deep Learning*, pages 714–728, Cham, 2019. Springer International Publishing.

[220] Sang-Woo Lee, Jin-Hwa Kim, Jaehyun Jun, Jung-Woo Ha, and Byoung-Tak Zhang. Overcoming Catastrophic Forgetting by Incremental Moment Matching. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 4652–4662. 2017.

[221] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[222] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12, 2016.

[223] Ruoming Pang, Tara Sainath, Rohit Prabhavalkar, Suyog Gupta, Yonghui Wu, Shuyuan Zhang, and Chung-Cheng Chiu. Compression of End-to-End Models. In *Proc. INTERSPEECH*, pages 27–31, 2018.

[224] Gido M. van de Ven and Andreas S. Tolias. Three scenarios for continual learning. *arXiv:1904.07734 [cs, stat]*, April 2019.

[225] Max Welling. Herding dynamical weights to learn. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1121–1128, Montreal, Quebec, Canada, June 2009.

[226] Jonathan Schwarz, Wojciech Czarnecki, Jelena Luketina, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & Compress: A scalable framework for continual learning. In *Proc. ICML*, pages 4528–4537, July 2018.

[227] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580 [cs]*, July 2012.

[228] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kjærgaard, Anind Dey, Tobias Sonne, and Mads Møller Jensen. Smart Devices are Different: Assessing and MitigatingMobile Sensing Heterogeneities for Activity Recognition. In *Proceedings of the 13th ACM Conference on Embedded Networked*

Sensor Systems, SenSys '15, page 127–140, New York, NY, USA, 2015. Association for Computing Machinery.

[229] Gary King and Langche Zeng. Logistic Regression in Rare Events Data. *Political Analysis*, 9(2):137–163, 2001.

[230] A. Reiss and D. Stricker. Introducing a New Benchmarked Dataset for Activity Monitoring. In *2012 16th International Symposium on Wearable Computers*, pages 108–109, June 2012.

[231] Thomas Stiefmeier, Daniel Roggen, Georg Ogris, Paul Lukowicz, and Gerhard Tröster. Wearable Activity Tracking in Car Manufacturing. *IEEE Pervasive Computing*, 7(2):42–50, April 2008.

[232] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. DeepSense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 351–360, Republic and Canton of Geneva, Switzerland, 2017.

[233] Harish Haresamudram, David V. Anderson, and Thomas Plötz. On the Role of Features in Human Activity Recognition. In *Proceedings of the 23rd International Symposium on Wearable Computers*, ISWC '19, pages 78–88, 2019.

[234] Manfredo Atzori, Arjan Gijsberts, Claudio Castellini, Barbara Caputo, Anne-Gabrielle Mittaz Hager, Simone Elsig, Giorgio Giatsidis, Franco Bassetto, and Henning Müller. Electromyography data for non-invasive naturally-controlled robotic hand prostheses. *Scientific Data*, 1:140053, December 2014.

[235] G. Li, A. E. Schultz, and T. A. Kuiken. Quantifying Pattern Recognition—Based Myoelectric Control of Multifunctional Transradial Prostheses. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 18(2):185–192, April 2010.

[236] Petko Georgiev, Nicholas D. Lane, Kiran K. Rachuri, and Cecilia Mascolo. DSP.Ear: leveraging co-processor support for continuous audio sensing on smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, page 295–309, New York, NY, USA, 2014. Association for Computing Machinery.

[237] M. Smith and T. Barnwell. A new filter bank theory for time-frequency representation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):314–327, March 1987.

[238] Rahaf Aljundi, Punarjay Chakravarty, and Tinne Tuytelaars. Expert Gate: Lifelong

Learning With a Network of Experts. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[239] Lisa Feldman Barrett and James A Russell. Independence and bipolarity in the structure of current affect. *Journal of personality and social psychology*, 74(4):967, 1998.

[240] Justin Salamon, Christopher Jacoby, and Juan Pablo Bello. A Dataset and Taxonomy for Urban Sound Research. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, page 1041–1044, New York, NY, USA, 2014. Association for Computing Machinery.

[241] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2017.

[242] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. *Proceedings of Machine Learning and Systems*, 3, March 2021.

[243] Wearable Device for Blind People Could be a Life Changer | NVIDIA Blog, October 2016.

[244] Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 9017–9028. 2018.

[245] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. And the Bit Goes Down: Revisiting the Quantization of Neural Networks. In *International Conference on Learning Representations*, 2020.

[246] Pierre Stock, Angela Fan, Benjamin Graham, Edouard Grave, Rémi Gribonval, Herve Jegou, and Armand Joulin. Training with Quantization Noise for Extreme Model Compression. In *International Conference on Learning Representations*, 2021.

[247] Julieta Martinez, Jashan Shewakramani, Ting Wei Liu, Ioan Andrei Barsan, Wenyuan Zeng, and Raquel Urtasun. Permute, Quantize, and Fine-Tune: Efficient Compression of Neural Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15699–15708, June 2021.

[248] R. Gray. Vector quantization. *IEEE ASSP Magazine*, 1(2):4–29, April 1984.

[249] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning

for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[250] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[251] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. MCUNet: Tiny Deep Learning on IoT Devices. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

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

[253] Pete Warden. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv:1804.03209 [cs]*, April 2018.

[254] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, 2017.

[255] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 215–223, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[256] Francesca Palermo, Matteo Cognolato, Arjan Gijsberts, Henning Müller, Barbara Caputo, and Manfredo Atzori. Repeatability of grasp recognition for robotic hand prosthesis control based on sEMG data. In *2017 International Conference on Rehabilitation Robotics (ICORR)*, pages 1154–1159, July 2017.

[257] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[258] Sebastian Houben, Johannes Stallkamp, Jan Salmen, Marc Schlipsing, and Christian Igel. Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark. In *International Joint Conference on Neural Networks (IJCNN)*, 2013.

[259] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster Gaze Prediction with Dense Networks and Fisher Pruning. *arXiv*, 2018.

[260] Minyoung Kim, Da Li, Shell X Hu, and Timothy Hospedales. Fisher SAM: Information Geometry and Sharpness Aware Minimisation. In *International Conference on Machine Learning (ICML)*, 2022.

[261] Jack Turner, Elliot J Crowley, Michael O'Boyle, Amos Storkey, and Gavin Gray. BlockSwap: Fisher-guided Block Substitution for Network Compression on a Budget. In *International Conference on Learning Representations (ICLR)*, 2020.

[262] Wei-Hong Li, Xialei Liu, and Hakan Bilen. Cross-domain few-shot learning with task-specific adapters. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7161–7170, June 2022.

[263] Janarthanan Rajendran, Alexander Irpan, and Eric Jang. Meta-Learning Requires Meta-Augmentation. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 5705–5715. Curran Associates, Inc., 2020.

[264] Abdelrahman Hosny, Marina Neseem, and Sherief Reda. Sparse Bitmap Compression for Memory-Efficient Training on the Edge. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, 2021.

[265] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, pages 1–1, 2019.

[266] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[267] Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. uNAS: 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, April 2021. Association for Computing Machinery.

[268] Filip Svoboda, Javier Fernandez-Marques, Edgar Liberis, and Nicholas D. Lane. Deep Learning on Microcontrollers: A Study on Deployment Costs and Challenges. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, EuroMLSys '22, page 54–63, New York, NY, USA, 2022. Association for Computing Machinery.

[269] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[270] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, December 2017. Conference Name: Proceedings of the IEEE.

[271] Marat Dukhan, Yiming Wu, Hao Lu, and Bert Maher. QNNPACK: Open source library for optimized mobile deep learning. https://engineering.fb.com/2018/10/29/ml-applications/qnnpack/, 2018.

[272] Shohreh Deldari, Hao Xue, Aaqib Saeed, Daniel V. Smith, and Flora D. Salim. COCOA: Cross Modality Contrastive Learning for Sensor Data. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 6(3), sep 2022.

[273] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[274] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*, 2021.

[275] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesus Corbal, Nikita Shustrov, Roma Dubtsov, Evarist Fomenko, and Vadim Pirogov. Mixed Precision Training of Convolutional Neural Networks using Integer Operations. In *International Conference on Learning Representations*, 2018.