Second Year Report

Young D. Kwon
Churchill College

UNIVERSITY OF CAMBRIDGE

Efficient Continual and On-device Learning in Mobile Computing

University of Cambridge
Department of Computer Science and Technology
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: ydk21@cam.ac.uk

June 28, 2022

Contents

List of Figures

List of Tables

Chapter 1

Progress Updates

Research Statement. With the rise of mobile devices, and the Internet of Things (IoT), the proliferation of sensory-type data has fostered the adoption of deep neural networks (DNN) in the modeling of a variety of mobile sensing applications [1]. A crucial characteristic common to these applications, often sitting on edge devices, is the need for a trained model to accommodate new classes and adapt to a dynamically changing environment. In such settings, the ability to perform continual and on-device Learning [2, 3] (to learn new knowledge, i.e., new classes in this thesis) without forgetting how to perform previously learned knowledge, becomes essential yet challenging in scenarios where a learned model is deployed on a resource-constrained device in the real world. Motivated by this, the main objective and scope of my research are to develop an *efficient continual and on-device learning system in mobile computing.* During the first two years of my Ph.D., I conducted a comprehensive literature search for CL and on-device learning, analyzed various system bottleneck that causes inefficiency in applying CL to resource-constrained systems, and developed new algorithms and systems that drastically improve the system efficiency without losing any accuracy.

First-year Contributions. During my first year, I first investigated the advantages and disadvantages of current CL methods in mobile sensing tasks. I then overcame the strict resource constraints of mobile and embedded systems by developing a new CL method optimizing the computational and storage requirements of one of the representative CL methods.

I began by analyzing the feasibility and applicability of CL methods in various mobile sensing applications while considering the limits poised by mobile and edge platforms, namely, low computational power, smaller memory and storage. Specifically, I adopted six CL methods from three different CL categories to evaluate their effectiveness and efficiency. Also, I employed six datasets from three modalities of mobile sensing tasks such as (1)

Human Activity Recognition (HAR) [4] based on accelerometer, gyroscope, and magnetometer, (2) Gesture Recognition (GR) [5] based on surface electromyography (sEMG), and (3) Emotion Recognition (ER) [6] based on speech. Furthermore, I implemented an end-to-end CL framework on two devices with different specifications: Jetson Nano and an off-the-shelf smartphone. Through the extensive evaluation, I found that the rehearsal-based CL approach (saving a small set of samples for rehearsal not to forget existing knowledge while learning new classes) often outperforms other CL approaches. This work became a good starting point helping me understand the challenges and limitations of current CL methods when applied to mobile sensing applications on resource-constrained devices. The result of the first work is presented in Appendix A. Currently, this work [7] has been published at SEC '21 (The Sixth ACM/IEEE Symposium on Edge Computing) and wins the **best paper award**.

Based on the findings of the first work, I realized that one of the major bottlenecks of enabling end-to-end CL on-device is an expensive computational requirement to learn new user inputs/classes (e.g., activities in HAR, gestures in GR). Furthermore, iCaRL (rehearsal-based CL method that I found outperforms other methods in [7]) requires a large storage budget to store representative samples of learned classes. Motivated by these limitations of prior works, I proposed a novel CL method, FastICARL, that improves upon iCaRL by reducing the CL time and alleviating the storage requirements to store rehearsal samples. To develop FastICARL, I first optimized the construction process of an exemplar set (which takes most of the CL time) to shorten the CL time to tackle the limitation of computational overhead. Specifically, to find the informative exemplars that can best approximate feature vectors over all training examples, iCaRL relies on herding which contains inefficient double for loops. Instead, FastICARL utilizes a k-nearest-neighbor and a max heap data structure to search exemplars more efficiently. In addition, to address the limitation on storage burden in resource-constrained devices, I further optimized FastICARL by applying quantization on rehearsal samples to reduce the storage requirement. Furthermore, I converted the 32-bit float data type into 16-bit float and 8-bit integer data types. Furthermore, I implemented the end-to-end CL framework on mobile and embedded devices of two different specifications: Jetson Nano and a smartphone (Google Pixel 4). To demonstrate its effectiveness and efficiency, I experimented with it in two audio sensing applications: an Emotion Recognition (ER) task and an Environmental Sound Classification (ESC) as a case study. The result of this work is presented in Appendix B. Also, this work [8] is published at INTERSPEECH '21 (Conference of the International Speech Communication Association).

Second-year Contributions. In my second year, in addition to enabling efficient CL, I started to expand the scope of my research to further optimize the on-device learning as mobile and embedded systems deployed in the real world often need to operate multiple applications. Also, since such systems' memory is limited (even more so on extremely resource-constrained platforms like microcontrollers (MCUs) with 100 KB of SRAM), maintaining a pre-trained model for each application is not scalable nor practical (applying scalar quantization [9] on each model can mitigate memory/storage issues, however, the compression rate is low). Then, sharing network structure [10] can be a solution for *correlated and similarly structured models*, however, it is not practical when systems want to operate *multiple heterogeneous models*. Therefore, I have asked the following research question: *is it feasible to compress multiple heterogeneous models without sacrificing accuracy on severely resource-constrained devices like MCUs?* To answer this research question, I propose YONO, a product quantization (PQ) [11] based approach that compresses multiple heterogeneous models and enables in-memory model execution and model switching for dissimilar multi-task learning on MCUs. I first adopt PQ to learn codebooks that store weights of different models. Also, I propose a novel network optimization and heuristics to maximize the compression rate and minimize the accuracy loss. Then, I develop an online component of YONO for efficient model execution and switching between multiple tasks on an MCU at run time without relying on an external storage device. Through extensive experiments, YONO shows remarkable performance as it can compress multiple heterogeneous models with negligible or no loss of accuracy up to 12.37×. Furthermore, YONO's online component enables an 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 generalizability of my method. To summarize, YONO shows great potential and opens further doors to enable multi-task learning systems on extremely resource-constrained devices. The result of this work is presented in Appendix C. Also, this work [12] is published at IPSN '22 (The 21st International Conference on Information Processing in Sensor Networks).

In addition, enabling efficient CL is also limited due to the difficulty of collecting labeled data since many prior works in CL require a relatively large amount of labeled data to learn new classes [13, 14, 3]. For mobile sensing tasks, it is more complicated to collect labeled data different from image data. This point leads me to ask the following questions: *how can I reduce the amount of labeled data for learning new incoming classes to enable CL while ensuring high performance? To what extent can I decrease the labeled data size, and what are the trade-offs between the amount of labeled data and the CL model's performance?* To answer these questions, I propose MetaCLNet, a novel rehearsal-based Meta CL method, that achieves the best of both worlds: enhanced CL performance and improved system efficiency. MetaCLNet combines rehearsal techniques and meta-learning for the first time to ensure high CL performance (less forgetting, fast learning, and high accuracy based on only a few samples per class). Also, to minimize resource overheads, MetaCLNet employs various optimization techniques such as compression of rehearsal samples and quantization of neural weights and activations. Surprisingly, MetaCLNet achieves near optimal CL performance, falling short by only 2.8% on accuracy compared to the oracle, outperforming existing Meta CL methods with substantial accuracy gains of 4.1-16.1%. Furthermore, compared to the state-of-the-art (SOTA) Meta CL method, MetaCLNet drastically reduces the memory footprint by 178.7×, end-to-end training latency by 80.8-94.2%, and energy consumption by 80.9-94.2%. I successfully deployed MetaCLNet on two edge devices, thereby enabling efficient CL on resource-constrained platforms where it is impractical to run SOTA methods. This work is described in Appendix D and is currently under review at SenSys '22 (The 20th ACM Conference on Embedded Networked Sensor Systems).

Ongoing Research. In recent years, with the increasing need to make tiny MCUs intelligent to facilitate various use cases such as smart homes, smart buildings and factories, TinyML designed to enable machine learning or deep learning on MCUs has attracted much attention from academia and industry. Likewise, performing CL on extremely resource-constrained devices like MCUs could have a massive impact on many mobile applications since many deployed MCUs can be updated continually with a new stream of new user data. However, to deploy tiny CL systems on MCUs, numerous challenges still need to be addressed. First of all, there is no deep learning framework that supports training (backpropagation) on MCUs. Second, MCUs have minimal on-chip memory resources. For example, a "high-end" MCU such as STM32F769 has only 512 KB of SRAM and 2 MB of embedded Flash (eFlash). Third, MCUs have limited computational power. For example, STM32F769's Arm 32-bit Cortex-M7 CPU is 216MHz, much slower than Cortex-A cores (e.g., Cortex-A78 used in Exynos 2100 has 2.81 GHz). Last but not least, MCUs are battery-powered, and thus the energy is very limited.

As the first step to developing tiny CL systems on MCUs, I have started a project to bring the DNN training that is only available on a server or powerful edge devices with at least a few hundred MB of RAM to a tiny MCUs with only a few hundred KB of RAM (e.g., at least 1,000× smaller in terms of available system resources). To overcome the challenges described above, I plan to propose an efficient DNN training methodology that aims to minimize the scarce on-chip memory of MCUs and computational costs. Also, I will take into account hardware characteristics of MCUs (e.g., embedded Flash is read-only during run time) while developing DNN training.

After establishing DNN training on MCUs, I will incorporate the developed DNN training framework to deploy tiny CL systems on MCUs. For this project, I want to build upon my previous work, MetaCLNet. Further, I want to optimize it to be deployed on MCUs. Some ideas for improvement are as follows: (1) the data augmentation to improve the CL performance and (2) searching for efficient network architectures designed for MCUs.

Chapter 2

Thesis Outline

The following is the proposed thesis outline:

1. **Introduction.** This chapter introduces the background and motivations to perform continual and on-device learning with real-world application scenarios in mobile computing.

2. **Related work.** This chapter describes the relevant research in the areas of on-device ML and CL to discuss the necessity, novelty, and contributions of this thesis.

3. **Exploring the performance and resource trade-offs of CL in mobile computing.** This chapter is based on the work published at SEC '21 that investigated the performance and resource trade-offs of various CL methods in many mobile sensing datasets of different data modalities. In addition, this chapter discusses the advantages and disadvantages of various CL approaches to provide insights for the subsequent chapters in my thesis.

4. **Efficient continual learning.** Based on the challenges and limitations discussed and identified in the previous chapter, this chapter explains two works that propose efficient CL methods. One work is published at INTERSPEECH '21, and the other work is under review at SenSys '22.

5. **Bringing on-device learning from edge devices to microcontrollers.** This chapter describes the frameworks that explore the interesting area of TinyML designed for extremely resource-constrained platforms, i.e., microcontrollers (MCUs). This chapter explains two works: the first work published at IPSN '22 introduces the compression techniques that can support multiple heterogeneous DNNs on MCUs. The second work (my current project) will describe how I develop and overcome the various challenges to enable DNN training on MCUs.

6. **Efficient continual learning on microcontrollers.** This chapter explains the final work that orchestrates all the small pieces developed during my Ph.D. to build the tiny and efficient CL systems on MCUs.
7. **Conclusion.** This chapter will first summarize the overall findings, contributions, and impacts of my research. On top of that, I will discuss the limitations and corresponding future works of this thesis. After that, I will conclude the thesis.

Chapter 3

Timeline

My research focuses on building an efficient on-device system that is exceptionally lightweight and capable of updating itself to changing environments and user inputs continually with minimal human intervention. The following is a proposed timeline outlining the targets and milestones for my research.

1. Literature review: Months 1-4 (Done)
    - Review the literature regarding CL and improve my knowledge about deep neural networks.
    - Understand the knowledge about mobile sensing, embedded systems.
2. Initial exploration of CL in mobile sensing. Months 5-9 (Done)
    - Conduct the foundational research to identify the feasibility and applicability of applying various CL methods in mobile sensing applications. (published at SEC '21)
    - Learn practical guidelines for applying CL in mobile sensing tasks on resource-constrained devices.
    - Propose a novel exemplar-based CL method called FastICARL. FastICARL reduces the latency for the CL time and requires less storage than the original iCaRL (published at INTERSPEECH '21).
3. Implementation of existing works and further literature review: Months 10-12 (Done)
    - Extensively review model compression literature, e.g., NAS, pruning, quantization, weight sharing.

- Implement some of the prior works to fully understand them and later use them as baselines of my future work.
- Review meta-learning and meta CL literature.
- Implement representative meta CL works to use them as baselines.

4. Exploration of better compression techniques for multiple heterogeneous networks on MCUs: Months 13-16 (Done)
    - Start with applying the basic PQ to a single model to see if it retains the accuracy and to what extent it can reduce a storage and memory footprint.
    - Extend PQ to be applied to multiple models and evaluate the performance of the PQ with respect to the accuracy, memory, storage footprint, and energy consumption.
    - Come up with optimization techniques based on PQ or on other compression techniques to further shrink the memory/storage footprint of multiple heterogeneous models.
    - Evaluate the optimized compression technique to what extent it can improve upon the baseline methods in terms of performance and system aspects.
    - Evaluate my method in various application scenarios. For example, when a user wants to include new models in my weight sharing method, it can represent the unseen models without learning from scratch.
    - This work is published at IPSN '22.
5. Adaptive CL systems with minimal human intervention: Months 17-21 (Done)
    - Analyze the advantages and disadvantages of the current state-of-the-art meta CL method (ANML).
    - Extend the regularization-based meta CL framework to rehearsal-based meta CL to maximize the performance with some additional overheads on memory.
    - Incorporate the possible optimizations based on quantization and latent replay to reduce resource usage of the rehearsal-based meta CL.
    - This work is submitted to SenSys '22.
6. Bringing DNN training from the server/edge to MCUs: Months 22-27 (In progress)
    - Propose an efficient and effective DNN training technique that can be done on extremely resource-constrained devices. Some possible directions are incorporating the few-shot learning for rapid adaptation with fewer samples and

    transfer learning with additional residual modules to recover the full accuracy.

   - Develop the proposed DNN training technique on MCUs.

7. Enabling adaptive and efficient CL systems on MCUs: Months 28-33
   - Combine all the building blocks developed during my Ph.D. to build the final CL systems that are adaptive to changing user inputs and environments and deployable on any device ranging from tiny devices like MCUs to edge devices like Jetson Nano or Raspberry Pi 3B+.

8. Thesis writing: Months 34-36

Chapter 4

Contributions

Papers Published

[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.
The Sixth ACM/IEEE Symposium on Edge Computing, 2021. (SEC '21). **Best Paper Award**

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

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

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

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

Papers Under Shepherding

[17] *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.
The 28th Annual International Conference on Mobile Computing and Networking, 2022. (MobiCom '22)

Papers Submitted

MetaCLNet: Rehearsal-based Meta Continual Learning with Compressed Latent Relays and Neural Weights
Young D. Kwon, Hong Jia, Jagmohan Chauhan, and Cecilia Mascolo.
Submitted to the 20th ACM Conference on Embedded Networked Sensor Systems, 2022. (SenSys '22)
Note: This work builds upon [18] and was later published as [19] at SenSys'23.

Work In Progress

UR2M: Uncertainty and Resource-aware Wearable Event Detection on Microcontrollers
Hong Jia, **Young D. Kwon**, Dong Ma, Nhat Pham, Lorena Qendro, Tam Vu, and Cecilia Mascolo.
Note: This work was initially targeted for either MobiCom'23 or UbiComp'23 and was ultimately published at PerCom'24 [20].

Building an on-device fully decentralized learning framework
Issam Nedjai, Abhirup Ghosh, **Young D. Kwon**, and Cecilia Mascolo.
Note: Potential target workshop is ISWC at UbiComp '22.

MCUTrain: Deep Neural Network Training on Microcontrollers
Young D. Kwon, in collaboration with Samsung AI researchers and Prof. Cecilia Mascolo.
Note: Preliminary results presented at SenSys'23 PhD forum [21], with full paper published at ICML'24 [22].

Appendix A

Exploring System Performance of Continual Learning for Mobile and Embedded Sensing Applications

Abstract

Continual learning approaches help deep neural network models adapt and learn incrementally by trying to solve catastrophic forgetting. However, whether these existing approaches, applied traditionally to image-based tasks, work with the same efficacy to the sequential time series data generated by mobile or embedded sensing systems remains an unanswered question.

To address this void, we conduct the first comprehensive empirical study that quantifies the performance of three predominant continual learning schemes (i.e., regularization, replay, and replay with examples) on six datasets from three mobile and embedded sensing applications in a range of scenarios having different learning complexities. More specifically, we implement an end-to-end continual learning framework on edge devices. Then we investigate the generalizability, trade-offs between performance, storage, computational costs, and memory footprint of different continual learning methods.

Our findings suggest that replay with exemplars-based schemes such as iCaRL has the best performance trade-offs, even in complex scenarios, at the expense of some storage space (few MBs) for training examples (1% to 5%). We also demonstrate for the first time that it is feasible and practical to run continual learning on-device with a limited memory budget. In particular, the latency on two types of mobile and embedded devices suggests that both incremental learning time (few seconds - 4 minutes) and training time (1 - 75 minutes) across datasets are acceptable, as training could happen on the device when the embedded device is charging thereby ensuring complete data privacy. Finally, we present some guidelines for practitioners who want to apply a continual learning paradigm for mobile sensing tasks.

A.1 Introduction

Deep learning has revolutionized the performance of various disciplines, including mobile and embedded systems applications. This is particularly true for applications relying on continuous streams of sensor data such as activity recognition [23], mental health, and wellbeing [24], gesture recognition [25], tracking and localization [26]. However, a crucial characteristic common to the above applications is the need for a trained model to adapt to accommodate new classes and to a dynamically changing environment. In these settings, the ability to *continually* learn [2, 3], that is, to learn consecutive tasks without forgetting how to perform previously learned tasks, becomes essential. Let us consider an example. Alice has a deep learning model deployed on her smartphone for human activity recognition (HAR) to recognize simple activities such as sitting and standing. As time passes, the model might want to learn new activities such as walking to be more beneficial to a very active Alice. A static model will learn new activities but will fail to predict older activities correctly due to *catastrophic forgetting* (CF) [27]. CF means the abrupt and near-complete loss of knowledge obtained from previous tasks when the model learns new tasks. Specifically, weights in a model important to previous task A (i.e., previous task) are changed to optimize towards task B (i.e., new task), which often leads to the degradation of task A's performance. With addressing CF issues, continual learning [3] allows deep learning models to learn incrementally (adapt or accommodate new classes/behaviors) and obviates the need to be trained every time from scratch, which might waste valuable resources on Alice's device.

In practice, enabling deep learning models to continually learning is very challenging due to the CF problem. Since CF was first identified in Multi-Layer Perceptrons (MLPs), many researchers have proposed methods to mitigate it [28, 29, 13, 30, 31] and evaluate it using small and large datasets [2, 32, 33]. However, the proposed methods are mainly evaluated in the field of computer vision with MLPs or Convolutional Neural Networks (CNN) based deep learning models. *It is unclear whether these methods are viable in sensor-based applications, where the modality of the data is significantly different from images, and sequence information needs to be captured [34].* Moreover, *most of the existing Incremental Learning (IL)*[1] *techniques [35] do not take into account the resource requirements of these devices*, which may make them inapplicable to embedded and mobile systems deployments. There is a clear need to understand the resource consumption limitations of existing continual learning methods to see if they are applicable to resource-constrained edge platforms.

To address the aforementioned limitations of prior work, we conduct 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

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

of Human Activity Recognition (HAR) [4] based on accelerometer, gyroscope, and magnetometer data. Next, we include two datasets from Gesture Recognition (GR) [5] based on surface electromyography (sEMG). We further incorporate an Emotion Recognition (ER) dataset [6] based on speech among audio sensing tasks to make our results generalizable to different modalities across diverse applications. **Second,** we examine trade-offs of studied IL methods in terms of their performance, storage footprint, computational costs, and the peak memory limit to consider the feasibility and applicability of the IL methods on mobile and embedded devices. To investigate the system limitations imposed by different configurations of IL, we implemented the IL framework on two types of devices with different specifications – an Nvidia Jetson Nano GPU (used in mobile robotics and tablets) and a smartphone (One Plus 7 Pro) CPU – with respect to computational costs, storage, and memory footprint.

Overall, the major contributions and findings of this paper are:

First, we conduct a systematic investigation of the CF problem on mobile and embedded applications using six state-of-the-art IL methods falling under three paradigms: **regularization** ((1) Elastic Weight Consolidation: EWC [29], (2) Synaptic Intelligence: SI [36], and (3) Online EWC [37]), **replay** ((4) Learning without Forgetting: LwF [28]), and **replay with exemplars** ((5) Incremental Classifier and Representation Learning: iCaRL [13] and (6) Gradient Episodic Memory: GEM [31]). In addition, to make our study generalizable across different modalities of data, we perform analysis on six datasets of three different sensing applications (HAR, GR, and ER).

Second, 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 [32]. 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 where each task contains distinct classes, which fits well with practical application scenarios (see §A.3.1 for detail). Specifically, we try three scenarios: adding only one class to a base classifier (simple), adding half of the classes, $N/2$, to a base classifier at once (mildly complex), and a very practical (complex) scenario where half of the classes, $N/2$, are added incrementally to a base classifier one by one, where N is the total number of classes. Through extensive experiments, we find that all IL methods perform well when presented with simple scenarios but fail in the complex scenario, except for iCaRL. The main reason for iCaRL's strong performance is its use of exemplar samples. To the best of 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 continual learning 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.

Third, we find that iCaRL and GEM require a modest amount of storage, which seemingly is not an issue on many modern devices as they support a large amount of storage (in order of a few GBs). Even at a maximum number of stored exemplars (i.e., 20% - 40% of training samples), iCaRL and GEM require only 2 MB–115 MB. However, GEM and EWC-based algorithms are computationally expensive in that the average IL time varies 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. iCaRL, in particular, needs less than a minute on a smartphone to do IL on a per-task basis and operates within a reasonable peak memory overhead (196–2,127 MB). In sum, our study shows that simple deep learning architectures such as one and two-layer long short-term memory (LSTM) [38] can be trained entirely on the smartphone, thereby ensuring complete user privacy.

Finally, based on our findings, we present a series of lessons and guidelines to help practitioners and researchers in their use of continual deep learning for mobile sensing applications.

In addition to the above contributions, we adapt the experimental protocol proposed in [32] which considers learning only two tasks. We extend this protocol so that it can incorporate any number of tasks $(D_1, ..., D_k)$ in an incremental manner and identify the best performing IL model by permutating a set of hyper-parameters and IL-method-specific parameters (see §A.3.3 for detail). Finally, we believe that our work and findings open the door to the use of continual learning in edge devices and applications.

A.2 Related Work

We begin by reviewing continual learning approaches and empirical studies to evaluate them, followed by applications of deep learning in the mobile and edge sensing domain.

A.2.1 Continual Learning

Continual learning studies the ability to learn over time from a coming stream of data by incorporating new knowledge while retaining previously learned experiences [3]. Continual learning is also called incremental learning (IL) [13], lifelong learning [3], and sequential learning [27]. In a continual learning setup, learning methods typically suffer from CF [27, 39], that is, 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 is because the learned parameters of the network that are optimized to perform well in task A (i.e., important weights to task A) are changed to maximize/minimize the objective/loss of task B. In recent years, many researchers have focused on solving the CF issue by proposing a range of IL approaches. The first group of approaches is a *regularization-based* method [29, 36, 37, 40] where regularization terms are added to the loss function to minimize changes to important weights of a model for previous tasks to prevent forgetting. Another group of approaches is a *replay-based* method [28] where model parameters are updated for learning a representation by using training data of the currently available classes, which is different from *replay with exemplars-based* method [13, 31, 8] where updating the model requires training data from the new class and also few training samples from earlier classes.

The proposed IL methods to solve CF are empirically evaluated using small and large datasets [2, 32]. However, these empirical studies either adopt only a few methods [2, 32] or neglect resource constraints of mobile and embedded devices with respect to storage and latency [32, 41]. To fill this gap, we perform a systematic study on six most cited (or state-of-the-art) IL methods from three representative categories of IL approaches with three continual learning scenarios with different difficulties. Also, we conduct the first comprehensive study of generalizability and trade-offs between performance, storage, and computational costs among the studied IL methods on mobile and embedded devices.

A.2.2 Deep Learning for Mobile Sensing Systems

Deep learning is increasingly being applied in mobile and embedded systems as it achieves state-of-the-art performances on many sensing applications such as activity recognition [42, 43], gesture recognition [44], and audio sensing [45]. [42] experimented with three variants of deep learning approaches such as feed-forward, convolutional, and recurrent neural networks on HAR datasets, and present guidelines for training neural networks. [46] proposed the DeepConvLSTM model in which convolutional layers extract the features from raw IMU data, and Long-Short Term Memory (LSTM) recurrent layers capture temporal dynamics of feature activations to improve the performance of HAR.

Deep neural networks have also helped applications that need to recognize hand gestures using surface electromyographic (sEMG) signals generated during muscle contractions [47, 48, 49]. [5] proposed a self-re-calibrating framework which can be updated to maintain the model's performance so that it does not need users' additional labels for re-training. [47] used sEMG of the forearm to classify finger touches with their proposed neural architecture combining convolutional, feed-forward, and LSTM layers.

Many works have investigated using deep learning for audio sensing tasks including Emotion Recognition, Speaker Identification [50], and Keyword Spotting. [51] proposed a deep learning modeling and optimization framework that specifically targets various audio sensing tasks in resource-constrained embedded systems. Keyword recognition [52] achieved 45% relative improvement with a deep learning model compared to a competitive Hidden Markov Model-based system.

In contrast to these works, we investigate whether current IL methods can enable a practical continual learning system for mobile and embedded sensing applications on-device and

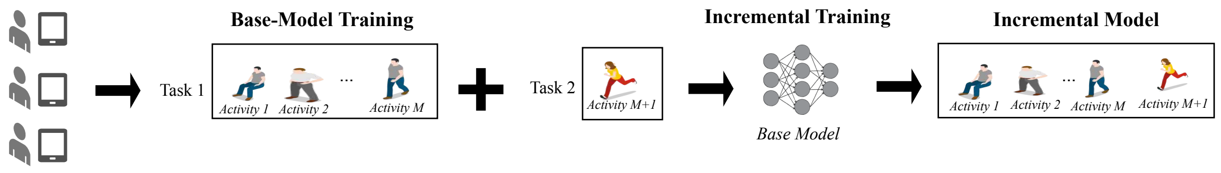


Figure A.1: Overview of our continual learning system.

what the performance implications of such systems are. In addition, to fully understand the issue of CF in mobile sensing where the modality of the data is significantly different from image datasets [34] with which the IL methods are typically evaluated, we implement an end-to-end continual learning framework that evaluates various IL methods in three embedded sensing applications (e.g., HAR, GR, and ER) with different data modalities (e.g., accelerometer, sEMG, and speech).

A.3 Continual Learning for Mobile and Embedded Sensing Framework

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 (§A.3.1). Then, we present six IL methods evaluated in this work (§A.3.2). We then describe the hyper-parameters of the LSTM based deep learning model and the different IL methods (§A.3.3). After that, we propose our novel IL model training process in §A.3.4. Next, we describe the datasets used in this study (§A.4.1). Finally, we provide brief details about our implementation (§A.3.5)

A.3.1 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 [32]. Figure A.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.

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. [32], 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 [2]. 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 [53].

A.3.2 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 state of the art methods (most cited) for IL and are most often used in machine learning papers for comparison. We now describe the employed methods.

LSTMs [38]: 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 [29]: 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.,

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

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

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

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

$$L(\theta) = L_i(\theta) + \lambda \sum_k \Omega^i_k (\theta^*_k - \theta_k)^2 \tag{A.3}$$

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^i_k represents the per-parameter regularization strength taking into account all previous tasks, calculated as:

$$\sum_{j=0}^{i-1} \frac{w^j_k}{(\triangle\theta^j_k)^2 + \varepsilon} \tag{A.4}$$

parameter distance $\triangle\theta^j_k$ 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 [28]: 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 [13]: 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 [54]). 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 [31]: 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.

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 A.5, 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.

A.3.3 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 [29, 55, 36, 28], 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 [13, 31], 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).

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 $\epsilon$
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 $q_k$
```

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 [56] 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 [53]. Note that budget size, $\mathcal{B} \subset \{1\%, 5\%, 10\%, 20\%\}$, is given as an input and fixed for replay with exemplars-based methods while other hyper-parameters are permuted. Since the total number of samples for each dataset is different, we use a ratio from the total training samples rather than a fixed number of samples for the budget size.

A.3.4 Model Training Process

We extend protocol [32] 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). 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 A.5), 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 are reported in Table A.1.

A.3.5 Implementation

We implemented our continual learning framework on Nvidia Jetson Nano and One Plus Pro smartphone 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 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 §A.4.1 for details) are not balanced [**?**, 42]. As in [57], we

Table A.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

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.

A.4 Experimental Setup

Before we present the findings of this work in Section A.5, we describe 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 (§A.4.1) and evaluation metrics adopted for systematic comparison of the IL techniques and their trade-offs between system aspects (e.g., storage and computational costs) (§A.4.2).

A.4.1 Datasets

We focus on three sensing applications (e.g., HAR, GR, and ER) as they are some of the most popular applications in the mobile sensing. Table A.1 shows the overview of the employed datasets.

Human Activity Recognition (HAR)

For the HAR application, we used three datasets: (1) HHAR [58], (2) PAMAP2 [59], and (3) Skoda [60]. 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.

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 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. [61]. 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 [61]. One user (i.e., the first participant) is used for testing, and the remaining users are left for training.

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. [42]. 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 [42, 23, 62], we use runs 1 and 2 from the sixth participant for testing and remaining data for training.

Skoda: The Skoda dataset contains activities of assembly-line workers in a manufacturing scenario. One subject wore 20 3D accelerometers on both arms. Following the preprocessing steps [46, 23], 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 [63] 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.

Ninapro (Per Subject): The Ninapro database is widely used in research on the hand movement recognition application. 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. [64], 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 [5, 65]. 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.

Ninapro (LOUO): To have consistent evaluation with the HAR application we adopt 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 remainders for training. The preprocessing steps are the same as in Ninapro (Per Subject).

Audio Sensing Task

We pick Emotion Recognition (ER) since it is one of the most widely adopted audio sensing tasks. We employ the EmotionSense dataset [6] 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 behavior of individuals in various settings [66][45][24].

EmotionSense: The EmotionSense 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. [51], we extract 24 log filter banks [67] 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.

A.4.2 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 [68] 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:

$$A_k = \frac{1}{k} \sum_{j=1}^{k} a_{k,j} \tag{A.5}$$

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:

$$f_j^k = \max_{l \in 1,\ldots,k-1} a_{l,j} - a_{k,j}, \quad \forall j < k \tag{A.6}$$

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:

$$I_k = a_k^* - a_{k,k} \tag{A.7}$$

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 metrics mentioned above, we also report **storage** and **latency** required to execute each IL method.

A.5 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 §A.5.1. Then, we study the performance of IL methods for Scenario 3 in §A.5.2. We examine the generalizability of IL methods across different datasets (§A.5.3). Then, we discuss the trade-offs of IL methods with respect to the storage, computational costs, and memory footprint. (§A.5.4). Finally, in §A.5.5, we investigate the effect of iCaRL specific parameters on the performance.

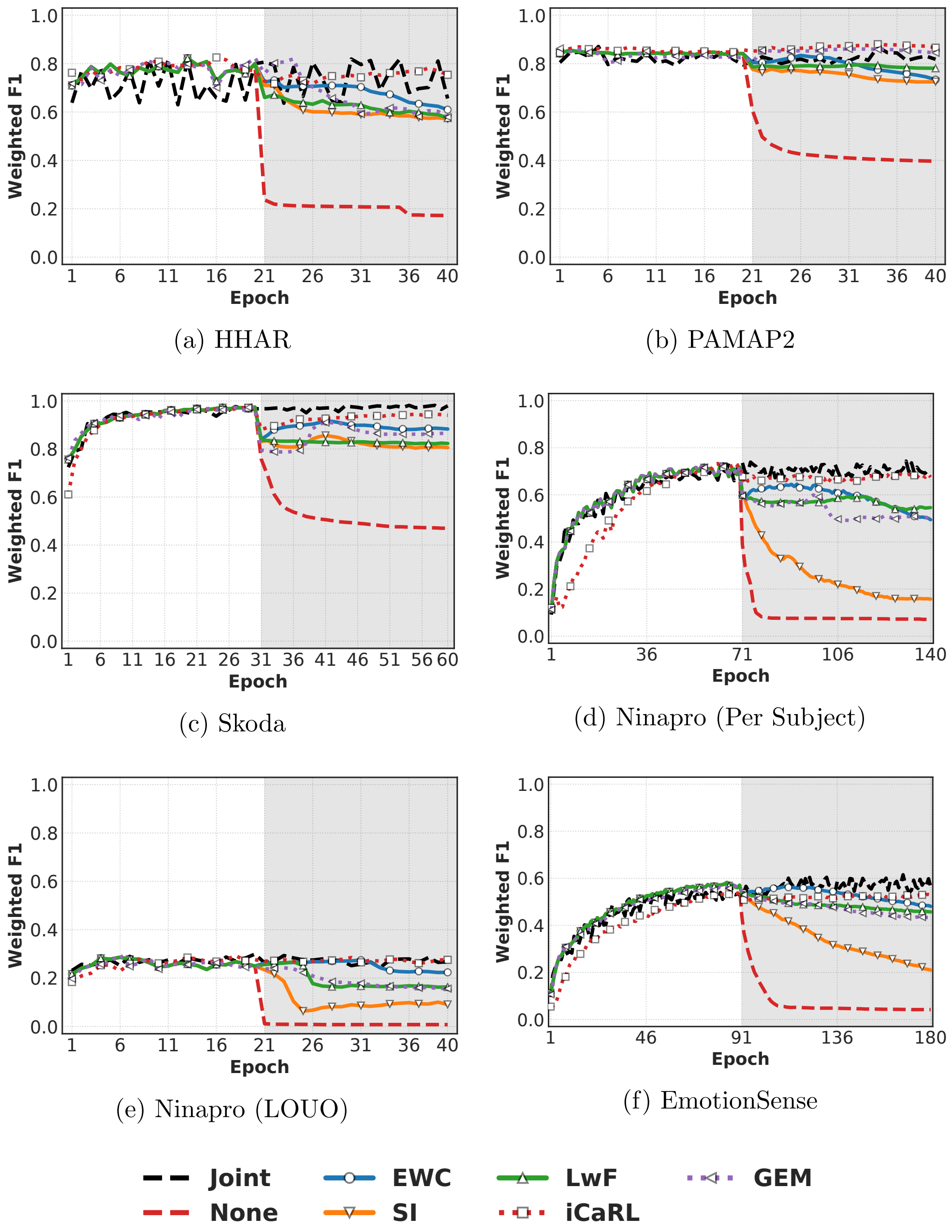


Figure A.2: The performance comparison of the five IL methods including two baselines in Scenario 1 on each dataset.

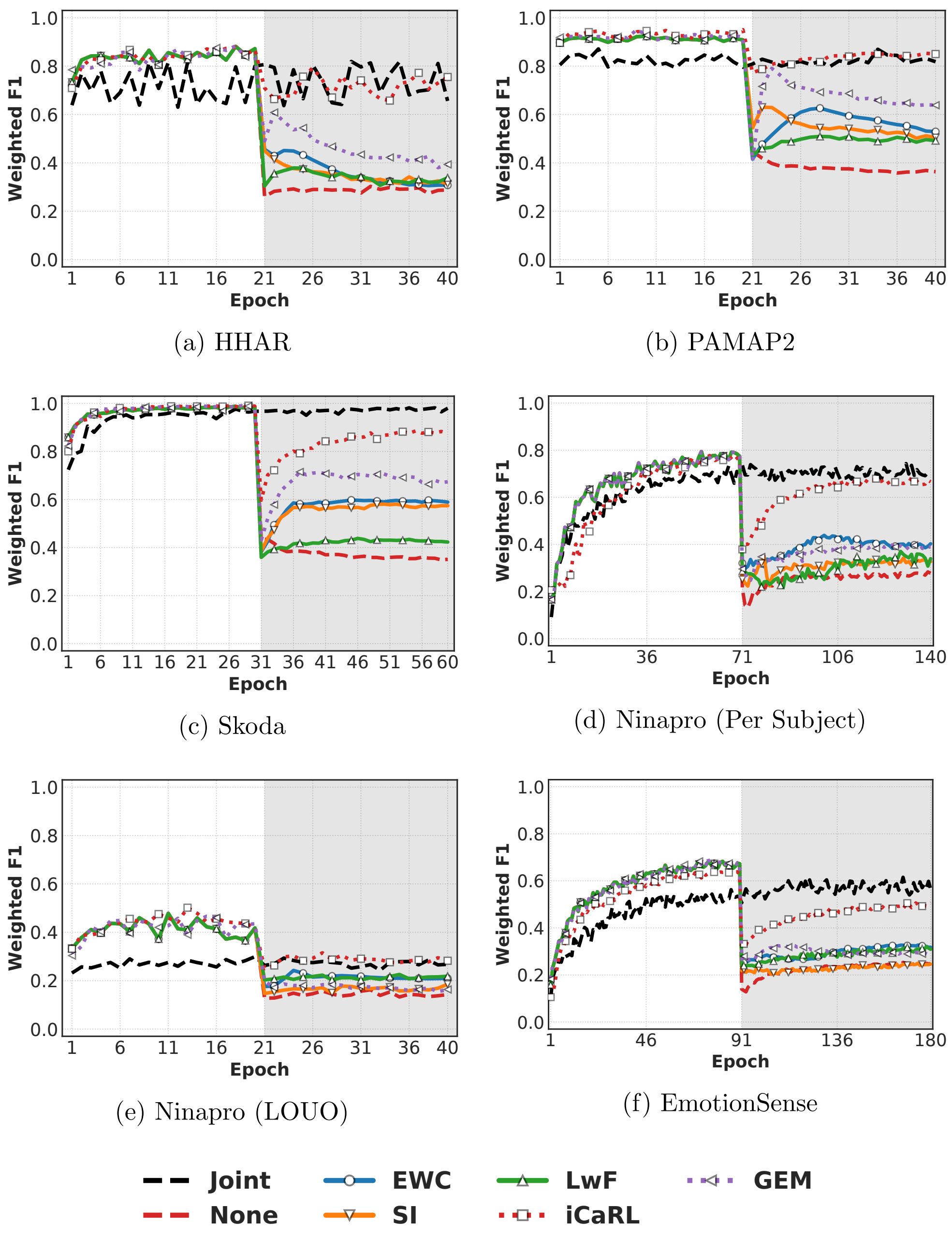


Figure A.3: Performance comparison in Scenario 2.

A.5.1 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 A.2 and Figure A.3, respectively. For

HAR and GR applications, the results of iCaRL/GEM with the budget size of 20% are shown in Figure A.2 and Figure A.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. **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. The white part in the figure shows performance on Task 1, and the grey part shows performance for Task 2.

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 [13]. 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 a 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 the 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 [69], the performance of LwF significantly decreases when the 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 in a batch gradient descent which can overestimate the importance of the weights and thereby lead 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 a smaller number of classes than the joint model, where all classes need to be classified from the first epoch.

A.5.2 Performance on Many Sequential Tasks

Figure A.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.*

A.5.3 Generalization

Table A.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

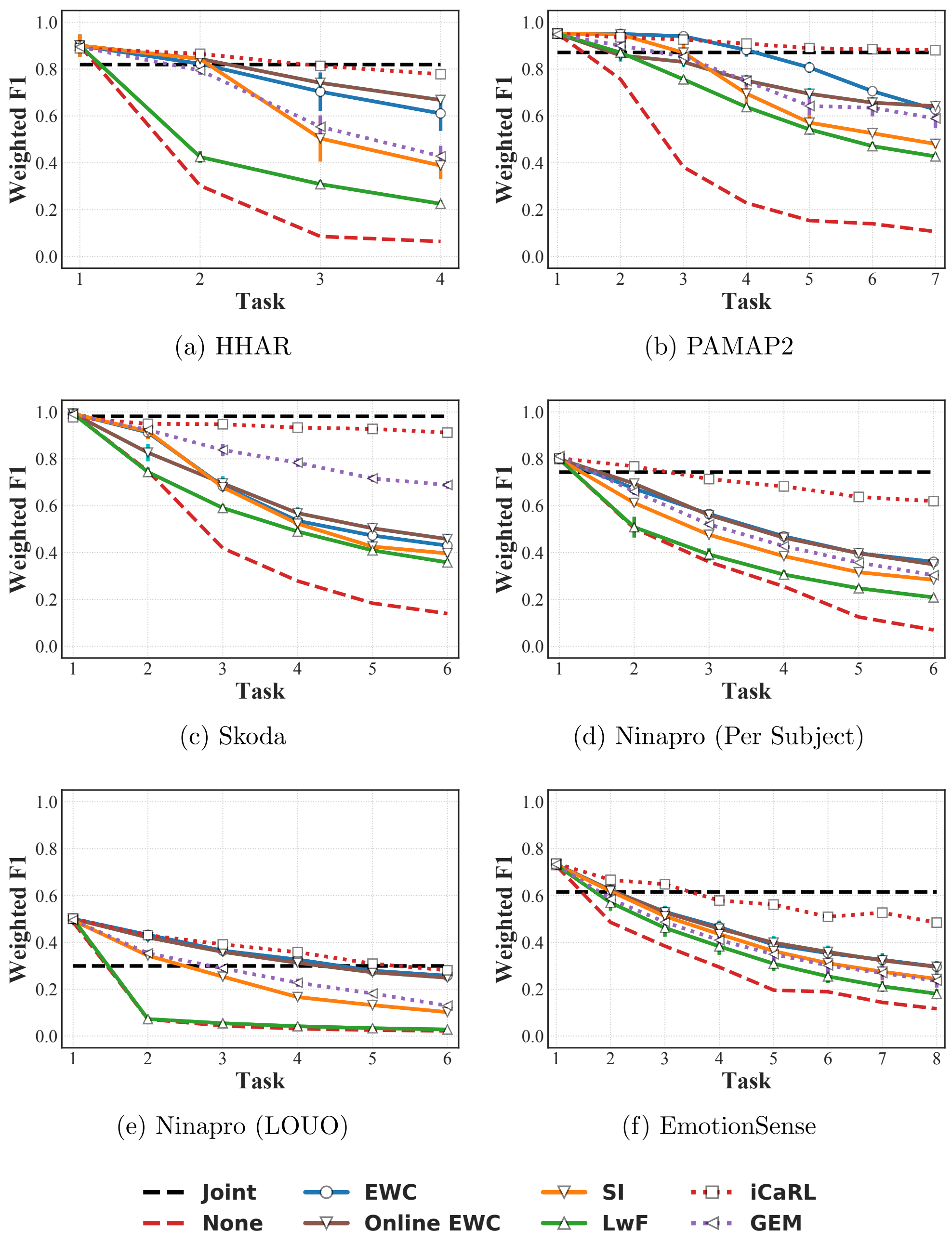


Figure A.4: The performance comparison in Scenario 3. All reported results are averaged over 10 trials, and standard-error intervals are depicted.

new tasks easily and thus has low performance in general. SI is neither good at learning new tasks (high I) nor at remembering old knowledge (high F). EWC and online EWC

Table A.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	-

offer a decent alternative to iCaRL without needing extra storage on-device but at the 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).*

A.5.4 Storage, Latency, and Memory Footprint

Storage: We report the storage overhead of each IL method, as shown in Table A.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^i_k), and reference weights (θ^*_k) of importance weights of the current task. EWC stores

Table A.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	$\mathcal{T} \times \mathcal{M} + \mathcal{B}$

fisher matrices and means for each task. Unlike EWC, Online EWC is only required to 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 A.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 A.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

Table A.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)**	**EmotionSense**
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

not excessive, as evident from Table A.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 §A.5.5).

Latency: The average training and incremental learning time to execute different IL methods are illustrated in Table A.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

[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 A.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)**	**EmotionSense**
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

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 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).

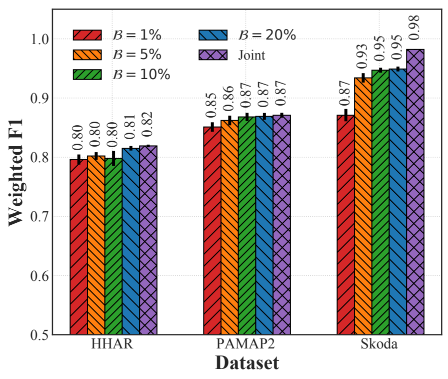
Having realized that iCaRL is the most promising method in terms of accuracy and latency, we wanted to check if iCaRL can also effectively work on modern smartphone CPUs. For this, we have implemented iCaRL on OnePlus 7 Pro for three datasets: Skoda, Ninapro (Per Subject), and EmotionSense as they represent 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

Table A.6: Average Latency (Training Time/IL Time) in seconds for iCaRL on three datasets - Scenario 3 on Smartphone.

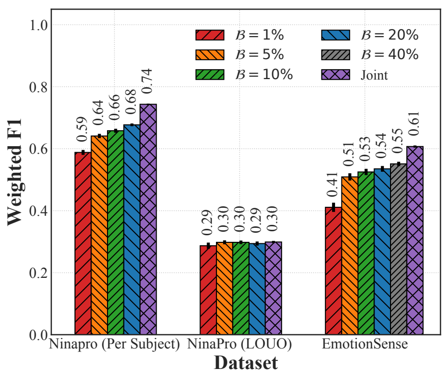
IL Method	**Skoda**	**Ninapro (Per Subject)**	**EmotionSense**
iCaRL (1%)	4400/9	1956/1.28	1568/0.5
iCaRL (5%)	3894/29	1974/3	1388/1.91
iCaRL (10%)	3869/72	2312/4.5	1535/2.6
iCaRL (20%)	3902/212	2008/5.1	1517/4.7
iCaRL (40%)	-	-	1506/8.1

results are shown in Table A.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 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.*

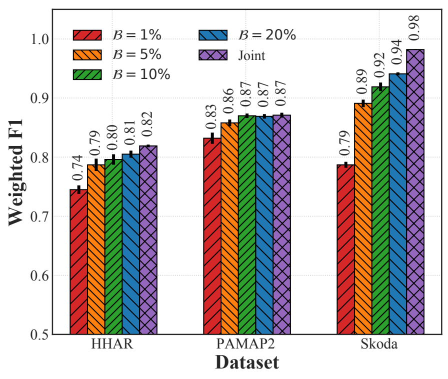
Memory footprint: We further examine the peak memory usage of iCaRL with its largest budget size of 20-40% on all the datasets to evaluate whether or not it can fit the tight memory budget of Jetson Nano. The peak memory overheads of running the end-to-end IL range from 196 MB for our smallest dataset 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*



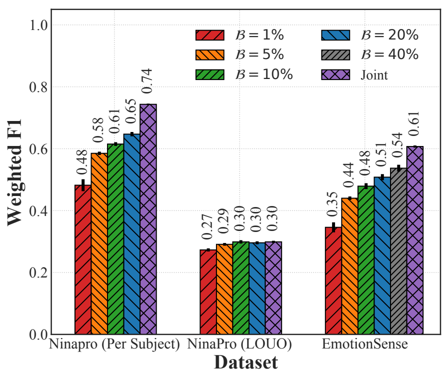
(a) Scenario 1



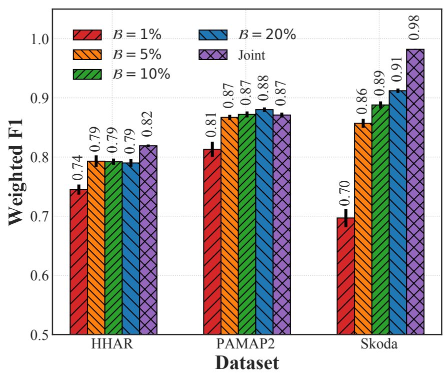
(b) Scenario 1 (GR & ER)



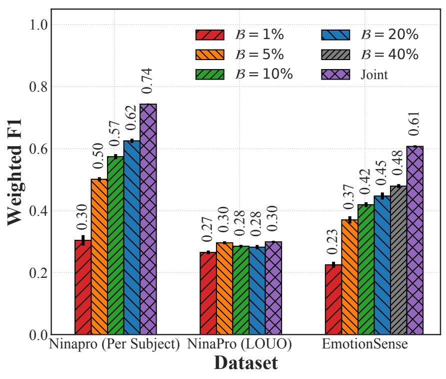
(c) Scenario 2



(d) Scenario 2 (GR & ER)



(e) Scenario 3



(f) Scenario 3 (GR & ER)

Figure A.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.

embedded sensing domain is applicable on resource-constrained devices within a reasonable memory overhead.

A.5.5 Performance with IL parameters

We study the importance of the storage budget parameter for iCaRL as it is the best performing IL method. Figure A.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.*

A.6 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 A.3) 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.

A.7 Conclusions and Future Work

In this paper, we studied the CF problem using six prominent IL methods based on three representative sensing applications (i.e., HAR, GR, and ER) in three continual learning scenarios with varying complexities. With our end-to-end IL framework implemented on Nvidia Jetson Nano and a smartphone (OnePlus 7 Pro), we conducted extensive experiments to investigate IL methods' performance, generalizability, and trade-offs of storage, computational costs, and memory footprints. We first identified that CF occurs in mobile and embedded sensing applications when IL methods are not used. We also found that while most IL methods solve the CF in simple scenarios, only iCaRL among the compared methods can successfully alleviate CF issues in more challenging scenarios across the employed datasets. Furthermore, we demonstrated that the IL approaches incur minor to modest storage, peak memory usage, and latency overheads (a minute per task in general), 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 IL to edge platforms.

As future work, we believe that it would be worthwhile to further investigate continual learning on more severely resource-constrained devices such as microcontrollers as they have smaller storage, limited memory, and low computational power to apply IL methods. Moreover, we want to study how model compression techniques such as quantization affect IL methods' performance. Similarly, combining binary neural networks with IL methods can be interesting future work. The other key point our study highlighted is that the major bottleneck comes from the training during the IL process. In this context, techniques such as Mixed Precision Training (MPT) [70] and quantization using only 16 or 8-bit floating-point representation [71] for weights might help improve the training efficiency in terms of its computational costs, memory footprints and latency.

Acknowledgments

This work is supported by a Google Faculty Award 2019 and by Nokia Bell Labs through their donation for the Centre of Mobile, Wearable Systems and Augmented Intelligence to the University of Cambridge. The authors declare that they have no conflict of interest with respect to the publication of this work.

Appendix B

FastICARL: Fast Incremental Classifier and Representation Learning with Efficient Budget Allocation in Audio Sensing Applications

Abstract

Various incremental learning (IL) approaches have been proposed to help deep learning models learn new tasks/classes continuously without forgetting what was learned previously (i.e., avoid catastrophic forgetting). With the growing number of deployed audio sensing applications that need to dynamically incorporate new tasks and changing input distribution from users, the ability of IL on-device becomes essential for both efficiency and user privacy.

However, prior works suffer from high computational costs and storage demands which hinders the deployment of IL on-device. In this work, to overcome these limitations, we develop an end-to-end and on-device IL framework, FastICARL, that incorporates an exemplar-based IL and quantization in the context of audio-based applications. We first employ k-nearest-neighbor to reduce the latency of IL. Then, we jointly utilize a quantization technique to decrease the storage requirements of IL. We implement FastICARL on two types of mobile devices and demonstrate that FastICARL remarkably decreases the IL time up to 78-92% and the storage requirements by 2-4 times without sacrificing its performance. FastICARL enables complete on-device IL, ensuring user privacy as the user data does not need to leave the device.

B.1 Introduction

A recent development of deep learning has revolutionized various audio-based applications such as emotion recognition (ER) [6], environmental sound classification (ESC) [72], and keyword spotting [73, 74]. However, in a real-world setting where a deployed audio classification models may need to dynamically incorporate new tasks (i.e., new classes or inputs) from users [65] and changing input distribution [75], current supervised learning approaches are severely limited due to the constrained nature of available resources on the edge devices and the catastrophic forgetting (CF) issue [27]. That is, a deep learning model becomes able to recognize a new task but forgets previously learned knowledge.

Many researchers proposed a range of Incremental Learning (IL) methods [3] to solve the CF problem. The first group of the IL approaches is a *regularization-based method* [29, 36, 37] where regularization terms are added to the loss function to minimize changes to important weights of a model for previous tasks to prevent forgetting. Kirkpatrick et al. [29] proposed a regularization-based method, Elastic Weight Consolidation (EWC), which uses the Fisher information matrix to identify important weights to the previous tasks and update less on those weights while learning a new task. Another group of the IL approaches is *exemplars-based method* [13, 31] where the method requires to store important samples from previous tasks to prevent from forgetting learned tasks. Rebuffi et al. [13] proposed a representative exemplar-based method, ICARL, that first utilizes herding [54] to search for exemplars (informative samples) and then uses knowledge distillation loss on the previously learned classes and classification losses on a new class to prevent forgetting and learn the new class. However, prior works are limited in two ways. First, It is challenging to enable IL on-device since IL 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 machine learning systems on resource-constrained devices. Quantization and low-bit precision of model parameters are utilized to reduce the size of the model [9, 76]. Low-rank factorization [77, 78] and pruning [79] have been proven effective in reducing model size, while retaining accuracy. IL with optimizations that allow its use on-device, however, has never been explored in the context of audio-based applications.

In this work, an end-to-end framework, FastICARL, is developed to enable efficient and accurate on-device IL in two audio sensing applications, an ER task and an ESC task. Also, FastICARL is a new IL method devised to improve upon the representative exemplar-based IL method, ICARL, as we observed that ICARL consistently outperforms EWC and other regularization-based IL methods [36, 37]. However, it has computational and storage issues. Thus, FastICARL solves these limitations while maintaining accuracy. First, we optimize the construction process of an exemplar set (which takes most of the IL time) to shorten the IL time to tackle the first limitation. Specifically, to find the informative exemplars that can best approximate feature vectors over all training examples, ICARL relies on herding which contains inefficient double for loops. Instead, FastICARL utilizes a k-nearest-neighbor and a max heap data structure to search exemplars more efficiently. In addition, to address the second limitation, we further optimize FastICARL by applying quantization on exemplars to reduce the storage requirement. We convert the 32-bit float data type into 16-bit float and 8-bit integer data types. Furthermore, we implement our end-to-end IL framework on mobile and embedded devices of two different specifications: Jetson Nano and a smartphone (Google Pixel 4). For a smartphone implementation, we employ MNN [80] and our implementation enables complete on-device training of new tasks/classes unlike TensorFlow Lite [81] or PyTorch Mobile [82] where only on-device inference is enabled.

Overall, the major contributions and findings of this paper are as follows. We design, implement, and evaluate FastICARL, which overcomes the limitations of the prior work. First of all, FastICARL shows that it can effectively solve the CF issues happening in audio-based datasets by achieving 69% and 71% weighted F1-scores for ER and ESC, respectively. FastICARL reduces the latency of exemplar set selection up to 78% on Jetson Nano and 92% on Google Pixel 4. Moreover, FastICARL decreases the storage requirement by 2-4 times without sacrificing its performance. In addition, we demonstrate that FastICARL can enable on-device IL without the support of the cloud. Hence, FastICARL ensures complete data privacy as user data does not need to leave the device. Finally, to the best of our knowledge, FastICARL is the first end-to-end and on-device framework that incorporates exemplar-based IL and quantization techniques in the context of audio sensing applications.

B.2 Methodology

In this section, we formulate our problem (§B.2.1) and describe the important prior work (§B.2.2). After that, we propose our IL method, FastICARL (§B.2.3).

B.2.1 Problem Formulation

We focus on Sequential Learning Tasks (SLTs) [32] from the audio sensing tasks, where new classes (e.g., different sounds in ESC) 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 (i.e., multi-task learning). Note that each task consists of disjoint groups of classes as we adopt class-incremental learning [53]. Formally, we are given training samples, $X^1, X^2, ...,$, where X^y is a set of samples of class y. Inspired by prior works [2, 65], 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).

B.2.2 ICARL

ICARL is the representative exemplar-based IL method in the literature that attempts to solve the CF problem of class-incremental setting. At the high level, ICARL maintains a set of exemplar samples for each observed class (see Algorithm 2). An exemplar set is a subset of all samples of the class to carry the most representative information of the class. When new tasks (classes) become available, ICARL first creates a new training set by joining all exemplar sets and the data of the new class. Then, it updates its weight parameters by minimizing a classification loss of the new task (class) as well as the distillation loss of the previous tasks (classes). Then, ICARL builds an exemplar set for the new class and trims the existing exemplars for previous classes. Finally, the classification is performed by finding the nearest-class-mean of exemplars to a given test sample in a feature space extracted from the learned representation.

B.2.3 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 comes 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 back-propagation, 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-neighbor 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 B.3. 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 §B.3.4). To alleviate this storage demand, we apply quantization on exemplar sets on the fly. Note that since budget sizes take up 72-99% of the storage requirements of FastICARL, we apply quantization only on

Algorithm 2: Construction and quantization of exemplar sets for ICARL/FastICARL

```
Input: Feature Extractor $\mathcal{F}()$, The number of exemplars to be stored $m$,
       Quantization bit $b$, IL method
Output: Quantized Exemplar set $Q$
Data: $X = \{x_1, ..., x_n\}$ of class $y$
1  $\mu \leftarrow \frac{1}{n}\sum_{i=1}^{n}\mathcal{F}(x_i)$                    // calculate class mean
   /* find m exemplars out of n samples                                  */
2  if IL method is ICARL then
3      for $k = 1, ..., m$ do
4          $p_k \leftarrow \operatorname*{argmin}_{x \in X} \left\| \mu - \frac{1}{k}(\mathcal{F}(x) + \sum_{i=1}^{k-1}\mathcal{F}(p_i)) \right\|$
5  if IL method is FastICARL then
       /* calculate feature distance between each sample and class mean   */
6      for $i = 1, ..., n$ do
7          $d_i = \mathcal{F}(x_i) - \mu$
       /* build max heap with size k                                      */
8      create max heap $H$ of pair {d, index}
9      for $k = 1, ..., m$ do
10         H.insert( $d_k, k$ )
       /* loop over the remaining samples while updating the max heap     */
11     for $k = m + 1, ..., n$ do
12         if $d_k < H$.extractMaxDist() then
13             $H$.pop()                                  // delete one item from H
14             $H$.insert( $d_k, k$ )
       /* build a sorted exemplar set P                                   */
15     for $k = m, ..., 1$ do
16         $i \leftarrow H$.extractMaxDistIndex(), $H$.pop()
17         $p_k \leftarrow x_i$
18 for $k = 1, ..., m$ do
19     $q_k \leftarrow$ Quantize($p_k, b$)
20 $Q \leftarrow (q_1, ..., q_m)$                                // Quantized exemplar set
```

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 quantization scheme used in [9] to minimize the information loss in quantization. The scheme utilizes an affine mapping of integers q to real numbers r, i.e.,

$$r = S(q - Z) \tag{B.1}$$

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

B.3 Evaluation

B.3.1 Datasets

We experiment with our method on two audio applications.

EmotionSense: For emotion recognition (ER) application, we employ the EmotionSense dataset [6] as it is used in multiple studies in audio sensing [66, 24, 45]. It contains audio signals which are clustered into five standard broader emotion groups, generally used by social psychologists [83] such as (1) Happy, (2) Sad, (3) Fear, (4) Anger, and (5) Neutral. This dataset has 2,235 samples, and each measurement corresponds to a particular emotion based on a 5-second context window. Following [51], we extract 24 log filter banks [67] from each audio frame over a time window of 30 ms with 10 ms stride. 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. As a result, we created an input of size 20×24.

UrbanSound8K: For environment sound classification (ESC) application, we adopt the UrbanSound8K dataset [72] 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 labeled urban sounds collected in real-world settings. This dataset consists of 10 audio event classes such as car horn, drilling, street music, etc. Following [84], 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 3-seconds of sound, we created an input of size 128×85, where 128 represents the number of frames and 85 represents aggregated feature size of the four audio features.

B.3.2 Experimental Setup

Task: As described in §B.2.1, 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 §B.3.4 are averaged over five times of experiments.

Model Architecture: We adopt a convolutional neural networks (CNN) architecture from prior work [84] to construct the ER and ESC models. To identify a high-performing

Table B.1: 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	

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 normalization, 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 B.1) while the majority of the model parameters are consumed in the FC layer as shown in [84]. 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 layers and a 0.5 dropout probability are adopted for the second and fourth Conv layers. ADAM optimizer [85] and learning rate of 0.001 are used.

Evaluation Protocol: Following prior works [6, 84], the 10% of each class is used as the test set and the remaining as the training data. In addition, 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 quantization levels (32, 16, and 8 bits). Finally, FastICARL (32, 16, and 8 bits) is compared.

Table B.2: 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

B.3.3 Implementation

To evaluate our framework on resource-constrained devices, we implemented it on an embedded (Jetson Nano) and a mobile device (Google Pixel 4). The Jetson Nano is an embedded mobile platform with four cores and 4 GB RAM. It is often utilized 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 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.

B.3.4 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 B.1. For both datasets, we present the performance according to the size of the budgets storing exemplars (5%, 10%, and 20%) to analyze 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 quantization of the saved exemplars for both ICARL and FastICARL is minimal. As shown in Table B.1, 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 5% budget size to achieves a weighted F1-score of 60-64% and successfully retain its weighted F1-score even after losing some information by applying quantization up to 8 bits on its exemplars.

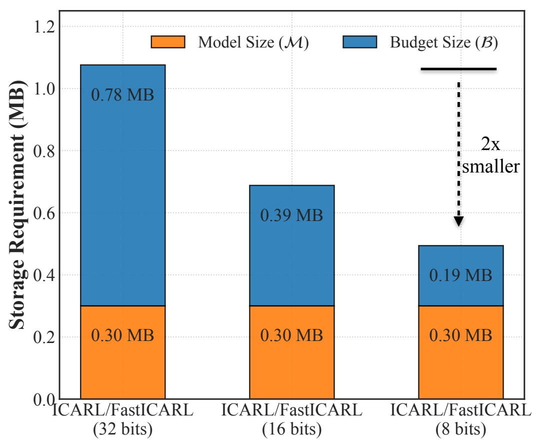
Latency: We measure the computational costs of sequentially learning additional classes based on a pre-trained model. The average IL time to run different IL methods is presented in Table B.2. The IL time of FastICARL (32, 16, and 8 bits) ranges 4.96-67.1 seconds on Jetson Nano and 0.83-11.2 seconds 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 B.1, FastICARL requires at most 0.49 MB for the EmotionSense dataset and 18 MB for the UrbanSound8K dataset, decreasing the storage requirement 2 to 4 folds over ICARL. Model sizes for EmotionSense and UrbanSound8K datasets are fixed as 0.3 MB and 1 MB, respectively.

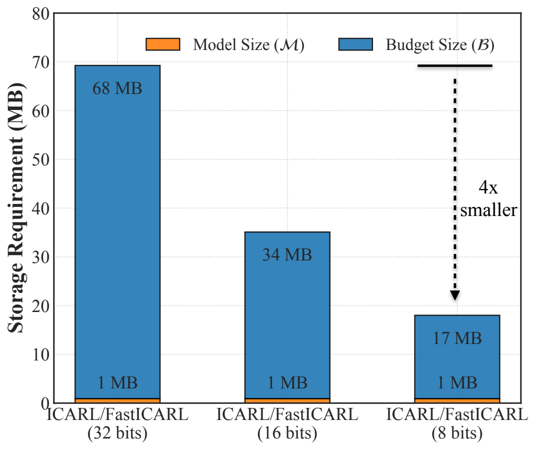
Based on the results in this section, we have demonstrated that FastICARL enables faster IL by reducing the IL time and storage requirements by applying quantization.

B.4 Conclusions

In this paper, we developed an end-to-end and on-device IL framework, FastICARL, that enables efficient and accurate IL in mobile sensing applications. We implemented



(a) EmotionSense



(b) UrbanSound8K

Figure B.1: 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.

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 quantizing its exemplars without sacrificing the performance.

There are many interesting directions that deserve further research. First of all, we want to extend our work to enable a higher degree of quantization (such as using 2 or 3 bits) and apply pruning on a model to reduce the model parameters and speed up the training process, which is another bottleneck of the IL. Furthermore, it is worth investigating IL methods on more severely resource-constrained devices such as micro-controller units having meager system resources.

Acknowledgments

This work is supported by ERC through Project 833296 (EAR) and by a Google Faculty Award. We thank Pete Warden for his valuable comments and suggestions.

Appendix C

YONO: Modeling Multiple Heterogeneous Neural Networks on Microcontrollers

Abstract

Internet of Things (IoT) systems provide large amounts of data on all aspects of human behavior. Machine learning techniques, especially deep neural networks (DNN), have shown promise in making sense of this data at a large scale. Also, the research community has worked to reduce the computational and resource demands of DNN to compute on low-resourced microcontrollers (MCUs). However, most of the current work in embedded deep learning focuses on solving a single task efficiently, while the multi-tasking nature and applications of IoT devices demand systems that can handle a diverse range of tasks (such as activity, gesture, voice, and context recognition) with input from a variety of sensors, simultaneously.

In this paper, we propose YONO, a product quantization (PQ) based approach that compresses multiple heterogeneous models and enables in-memory model execution and model switching for dissimilar multi-task learning on MCUs. We first adopt PQ to learn codebooks that store weights of different models. Also, we propose a novel network optimization and heuristics to maximize the compression rate and minimize the accuracy loss. Then, we develop an online component of YONO for efficient model execution and switching between multiple tasks on an MCU at run time without relying on an external storage device.

YONO shows remarkable performance as it can compress multiple heterogeneous models with negligible or no loss of accuracy up to 12.37×. Furthermore, YONO's online component enables an 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 generalizability of our method. To summarize, YONO shows great potential and opens further doors to enable multi-task learning systems on extremely resource-constrained devices.

C.1 Introduction

With the rise of mobile, wearable devices, and the Internet of Things (IoT), the proliferation of sensory type data has fostered the adoption of deep neural networks (DNN) in the modeling of a variety of mobile sensing applications [1]; researchers use DNN trained on sensory data in mobile sensing tasks such as human activity recognition [23, 86], gesture recognition [25], tracking and localization [26], mental health and wellbeing [24], and audio sensing applications [34]. While machine learning (ML) models are becoming more efficient on resource-constrained IoT devices [87], most existing on-device systems, designed for microcontroller units (MCUs), are targeted at one specific application [88, 89, 90]. Conversely, multi-application systems capable of directly supporting a wide range of applications on-device could be more versatile and useful in practice. Specifically, we envisage a system powered by MCUs that can recognize users' voice commands, activities and gestures, identify everyday objects and people, and understand the surrounding environments: this has the potential to boost the utilization of IoT devices in practice (e.g., help visually impaired individuals understand their environments [91]).

However, realizing such multi-tasking system faces three major challenges. **First**, multiple dissimilar tasks based on different modalities of incoming data (e.g., voice recognition (audio), activity recognition (accelerometer signals), object classification (image)) need to co-exist in the same framework. As discussed in [92], conventional multi-task learning (MTL) approaches cannot address *multiple heterogeneous networks* effectively. **Second**, IoT devices based on MCUs are extremely resource-constrained [93, 94]. 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. **Finally**, in real-world deployment scenarios, context switching of different ML tasks at run-time could incur overheads on memory-constrained MTL systems as demonstrated in [92], 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 the usability and responsiveness of the system.

To solve these challenges, one of the common techniques employed is to compress individual models separately using pruning [79, 95] and quantization [9]. However, model compression techniques are limited since extensive and iterative finetuning is required to ensure high performance after compressing a model. Also, 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 generalization of multiple tasks, while increasing the compression rate of the model by sharing network structures. However, *sharing/compressing multiple heterogeneous networks* has not been fully examined. Furthermore, prior work [92] attempts to solve the MTL of multiple heterogeneous networks by sharing weights of multiple models via virtualization. However, this method is complex, and the compression ratio is constrained to 8.08× (see §C.4.2 for detail), thereby limiting the type of IoT devices on which it can operate. Further, 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 [96]).

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 Quantization (PQ) [11] to maximize compression rate and on-chip memory operations to minimize external disk accesses for heterogeneous multi-task learning. 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 quantized. 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 neighbor search out of billions of vectors in the database community [11, 97, 98] and single layer compression in an individual model [99, 100, 101, 102, 103], we jointly apply PQ on multiple models instead of on a layer of a model. We find just one pair of codebooks that are generalizable and thus can be shared across many dissimilar tasks. We then propose a novel optimization 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 maximize 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. The first component is an offline phase in which a shared PQ codebook is learned and multiple models are incorporated. We implement the offline phase of our system on a server. The second component is 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 state-of-the-art prior work on heterogeneous MTL [92] 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 [92] due to our usage of the optimized network architecture (see §C.4.2 for detail) and is very close to the accuracy of the uncompressed models (0.4% loss in accuracy). Further, 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). Next, to evaluate the generalizability 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 [92]). 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 C.3 for details), and demonstrate that YONO enables an 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).

C.2 YONO

In this section, we first present the overview of our multitasking system, YONO (§C.2.1). Then, we introduce the background on PQ and its applications on single model compression (§C.2.2). We then explain how we utilize 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 optimization process to ensure the performance of the compressed networks remain close to original models (§C.2.4). On top of that, based on an observation (detailed in §C.2.5), we further propose optimization heuristics to maximize 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 (§C.2.6).

C.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 §C.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 C.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.

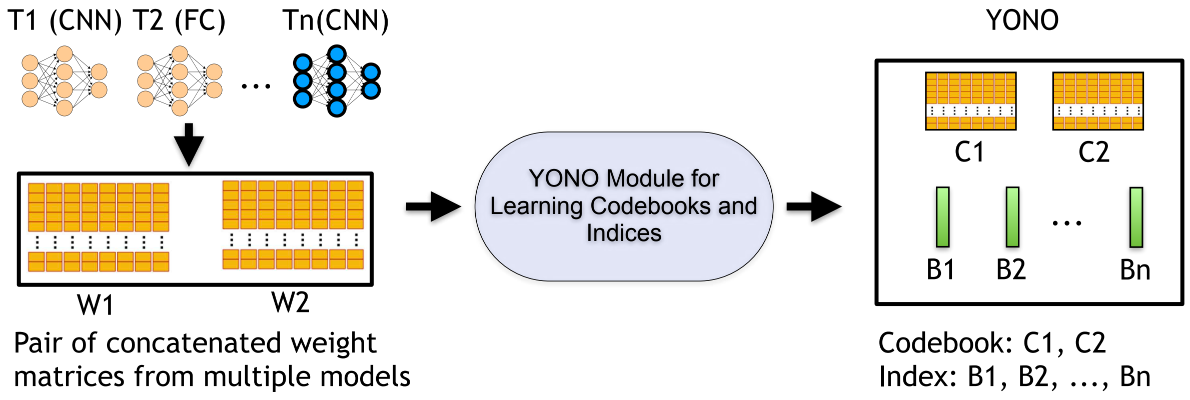

T1 (CNN) T2 (FC)
Tn(CNN)
W1
W2
Pair of concatenated weight matrices from multiple models
YONO Module for Learning Codebooks and Indices
YONO
C1
C2
B1
B2
Bn
Codebook: C1, C2
Index: B1, B2, ..., Bn

Figure C.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 optimization process and heuristics to minimize the accuracy loss compared to the original models.

C.2.2 Product Quantization 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 quantization (VQ) [104], in which it attempts to find the nearest codeword, **c**, to encode a given vector, **w**. Suppose we are given a codebook, **C**, that contains a set of representative codewords, we can reconstruct/approximate the given vector **w** by using **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.

$$\underset{b}{\operatorname{argmin}} \|\mathbf{w} - \mathbf{C}b\|^2 \tag{C.1}$$

where **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, **c**, nearest to the given vector, **w**). $\|\cdot\|$ is a l_2 norm. Solving Equation C.1 is equivalent to searching the nearest codeword. Besides, the codebook, **C**, is learned by running the standard k-means clustering over all the given vectors [11].

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.

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

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, $\ldots$, 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 $\mathbf{W}$, the reconstructed layer of a neural network will perform normally as demonstrated in prior works using PQ to compress a single neural network [99, 102].

C.2.3 Compressing Multiple Heterogeneous Networks

As described in §C.2.2, PQ is typically used to compress a single model in machine learning literature [101, 103]. 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 [105]). 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 maximize the compression ratio. We will explain how we ensure high performance of the compressed models in the next subsections (§C.2.4 and §C.2.5).

As in Figure C.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 [102]. 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 optimized network architectures [106, 107, 108]. 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 generalizable 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.

C.2.4 Network Optimization

After learning a pair of codebooks for multiple models as in §C.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 [95], 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 C.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 optimization process to improve the performance of the decompressed models. As discussed in prior works [99, 101], in general, higher weight differences (i.e., errors) result in increased loss of accuracy. Thus, to minimize the impact of the weight differences, we adopt to use the iterative optimization procedure, inspired by the Expectation-Maximization (EM) algorithm [109] and prior work [101]. 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 optimization procedure is novel in that (i) we perform network optimization across multiple heterogeneous networks and (ii) we do not update codewords in our learned codebooks since we want our codebooks to be generalizable to compress unseen models and datasets during the codebook learning procedure, different from single model compression methods [99, 101, 102, 103]. In Section C.4, we demonstrate the generalizability of our learned codebooks and our system on new models that are trained on new datasets that YONO did not see in its codebook learning.

C.2.5 Optimization Heuristics

In addition, we further propose an optimization heuristic that can maximize 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

Algorithm 3: YONO Network optimization 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 $\epsilon$, 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 */
1  $\widehat{\mathbf{W}} \leftarrow \mathbf{C}(\mathbf{b})$   // reconstruct a model via codebooks and indices
2  for $\ell = 2, ..., L-1$ do
3      FreezeWeights($\widehat{\mathbf{W}}^\ell$)
   // run network training (e.g., BackProp) with loss function
4  Finetune($\widehat{\mathbf{W}}, \mathcal{D}^{TRAIN}$)
5  $acc\_orig \leftarrow$ Evaluate($\mathbf{W}, \mathcal{D}^{TEST}$))
6  $acc\_recon \leftarrow$ Evaluate($\widehat{\mathbf{W}}, \mathcal{D}^{TEST}$))
7  if $acc\_orig - \epsilon \leq acc\_recon$ then
8      return $\widehat{\mathbf{W}}, \mathbf{b}$
/* Perform a further network optimization step */
9  $S \leftarrow (1, L)$ // finetuning layer set
10 $\hat{\mathbf{b}} \leftarrow \mathbf{b}$
11 for $i = 1, ..., L-2$ do
       // E-step: code re-assignment
12     for $\ell \notin S$ do
13         $\hat{\mathbf{b}}^\ell \leftarrow \underset{b \in \hat{\mathbf{b}}^\ell}{\text{argmin}} \left\|\hat{\mathbf{w}}^\ell - \mathbf{C}b\right\|^2$
       // M-step: model update
14     $\widehat{\mathbf{W}} \leftarrow \mathbf{C}(\hat{\mathbf{b}})$
15     for $\ell \notin S$ do
16         FreezeWeights($\widehat{\mathbf{W}}^\ell$)
17     Finetune($\widehat{\mathbf{W}}, \mathcal{D}^{TRAIN}$)
18     $acc\_orig \leftarrow$ Evaluate($\mathbf{W}, \mathcal{D}^{TEST}$))
19     $acc\_recon \leftarrow$ Evaluate($\widehat{\mathbf{W}}, \mathcal{D}^{TEST}$))
20     if $acc\_orig - \epsilon \leq acc\_recon$ then
21         return $\widehat{\mathbf{W}}, \hat{\mathbf{b}}$
22     if heuristics is $OURS$ then
           // choose a layer to finetune based on our heuristics
23         $\ell \leftarrow \underset{\ell}{\text{argmax}} \left\|\mathbf{W}^\ell - \widehat{\mathbf{W}}^\ell\right\|^2 / N^\ell$
24         $S \leftarrow (S, \ell)$
```

in each layer is considerably different. For example, MicroNet-KWS-M [88] (we adopt this network architecture in our evaluation. Refer to Section C.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 optimization 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.

$$\operatorname*{argmax}_{\ell} \left\| \mathbf{W}^{\ell} - \widehat{\mathbf{W}}^{\ell} \right\|^{2} / N^{\ell} \tag{C.3}$$

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.

In summary, through the optimization heuristics, YONO identifies a layer with the highest weight difference per parameter. After that, YONO finetunes the identified layer using our network optimization process introduced in §C.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 maximize 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 quantized into 8-bit integers in the online component of YONO as described in the next subsection.

C.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 C.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.

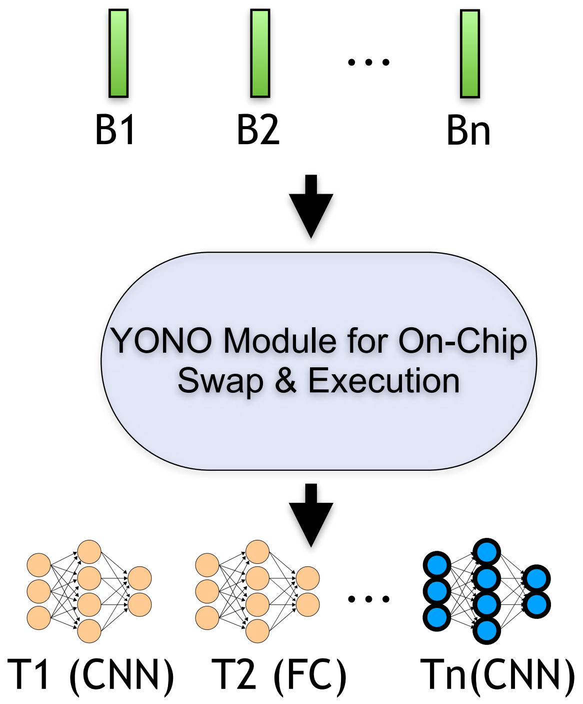


Figure C.2: Overview of the online component of YONO. The online module enables fast and efficient model loading/swap and in-memory execution.

Learned Codebooks: As described in subsections §C.2.2-C.2.5, YONO learns a pair of codebooks by applying PQ on multiple heterogeneous neural networks with our novel optimization 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, quantization information, and so on.

Task Executor: We now present the explanation of our task executor. As we adopt TensorFlow Lite for Microcontrollers (TFLM) [87] to run the deployed model on MCUs, YONO also follows its model representation and interpreter-based task execution. As 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 [110] 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 utilizes the PQ codebooks, indices, and relevant elements, such as batch normalization layer's mean and variance, quantization 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 quantized to 8-bit integers. Thus, while loading each layer of the model, YONO converts 16-bit floats to 8-bit integers using the saved quantization information. Specifically, we use the quantization scheme used in [9] to minimize the information loss in quantization. We utilize an affine mapping of integer q to real number r for constant quantization 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 quantized 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.

C.3 System Implementation

We introduce the hardware and software implementation of YONO.

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 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 §C.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.

C.4 Evaluation

We now present the results of the evaluation on our system. §C.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 [92] that tackles MTL of different neural networks, we begin with evaluating our system with the same datasets used in [92] consisting of five datasets for two modalities (i.e., image and audio) (§C.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 §C.4.3. Further, to demonstrate the generalizabilty 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 (§C.4.4). Finally, we present the results of our online in-memory model execution and swap operations in §C.4.5.

C.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 [6, 84], 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 Virtualization (NWV) [92] is the state-of-the-art heterogeneous MTL system that treats weights of neural networks as consecutive memory locations which can be virtualized and shared by multiple models. Note that we use reported results of [92] on an MCU, which relies on simplified LeNet architecture.

Scalar Quantization (Int8): This baseline compresses a single model by quantizing 32-bit floats into low-precision fixed-point representation (e.g., 8-bit) [9, 111]. As in [111], we employ both post-training quantization and quantization-aware training schemes. We then report the results of the best-performing scheme in our evaluation. Besides, we only include 8-bit quantization as sub-byte datatypes (e.g., 4-bit or 2-bit) are not natively supported by MCUs [88]. We leave sub-byte quantization 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.

Table C.1: Summary of datasets, model architectures, mobile applications used in §C.4.2 and §C.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

PQ-M: This baseline uses PQ to compress multiple heterogeneous models to a pair of the shared codebooks but does not apply our optimization process and heuristics as described in Section C.2. We include this to conduct an ablation study to evaluate the impact of the proposed optimization 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 optimization process without the heuristics described in Section C.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.

C.4.2 Performance

Following [92], 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 [92] to make a fair comparison. First, four image datasets are employed, namely MNIST [112], CIFAR-10 [96], SVHN [113], and GTSRB [114] 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) [115] for keyword spotting is used.

Model Architecture. We adopt optimized neural network architectures, designed to be used in the resource-constrained setting, such as variants of MicroNet [88], simplified LeNet used in [92]. For MNIST, we use the simplified LeNet as it is used in [92] 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 [8], the number of convolutional filters. A basic convolutional layer consists of 3×3 convolution, batch normalization, 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 optimizer [85] and learning rate of 0.001 are used. The datasets, architectures, and applications are summarized in Table C.1.

Accuracy. We show the accuracy results here. Figure C.3 shows the accuracy of each baseline so that we can analyze the impact of our proposed techniques in our system. To begin with, the uncompressed (original) model serves as a performance upper bound. 8-bit quantization 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 specialized 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 optimization 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 multiple MTL systems [92] and other techniques focusing on a single model compression [9, 79, 101]. 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 [92]. Also, note that differently from [92] which used LeNet, we used optimized network architectures such as MicroNet and lightweight CNN that can execute on resource-constrained MCUs (refer to §C.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 [92], respectively.

Compression Efficiency. Table C.2 shows the overall efficiency in compressing heterogeneous networks trained with five datasets of two modalities. First, the combined

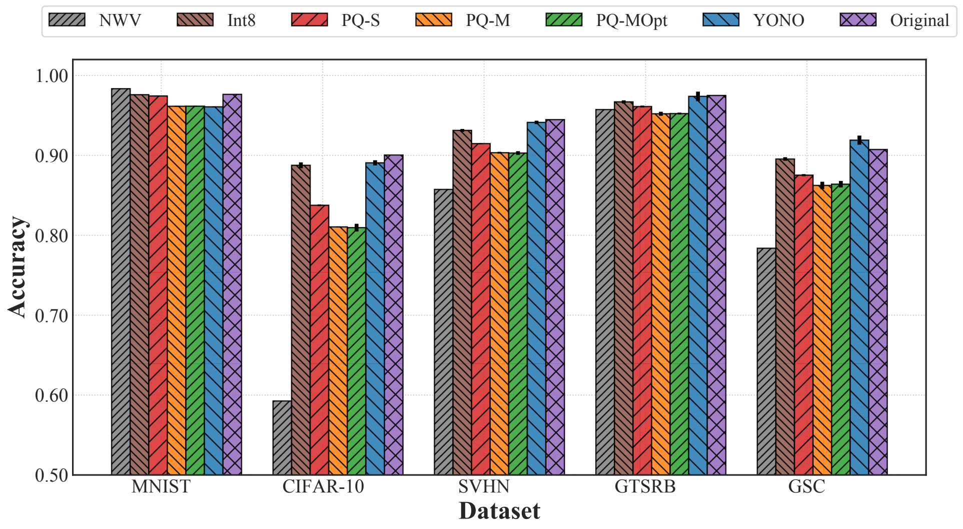


Figure C.3: The inference accuracy of the heterogeneous MTL systems trained with five datasets of two modalities. Reported results are averaged over five trials, and standard-deviation intervals are depicted.

Table C.2: The compression efficiency of the heterogeneous MTL systems trained with five datasets of two modalities.

	NWV [92]	**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

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 quantization 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 [92].

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

C.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 [58, 25].

Datasets. On top of the five datasets used in the previous subsection, we add two datasets of two additional modalities: HHAR [58] and Ninapro DB2 [63], 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 optimized neural architectures and also conduct a hyper-parameter search as described in §C.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 C.1 for detail).

Accuracy. Figure C.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 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 §C.4.2. 8-bit quantization 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 optimization 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 C.3 shows the overall efficiency in compressing heteroge-

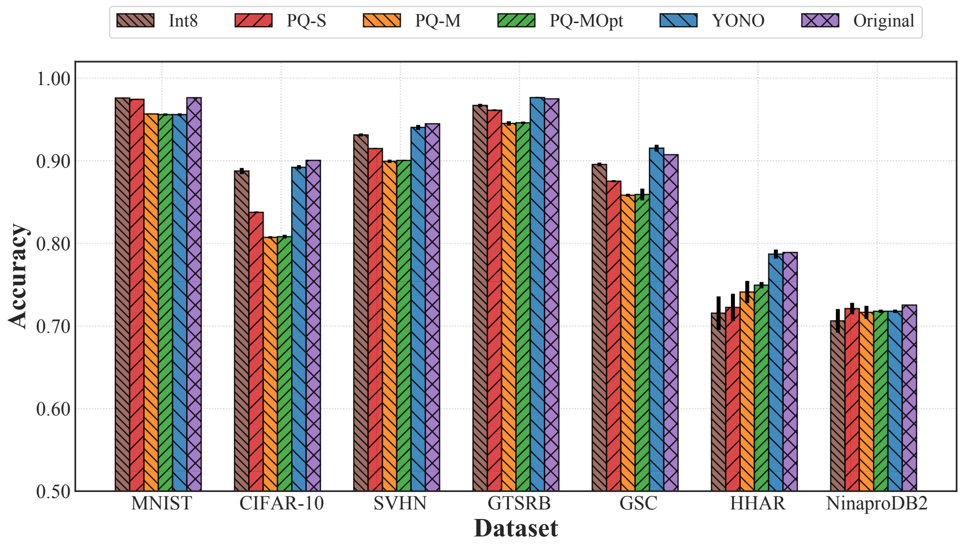


Figure C.4: The inference accuracy of the heterogeneous MTL systems trained with seven datasets of four modalities. Reported results are averaged over five trials, and standard-deviation intervals are depicted.

Table C.3: The compression efficiency 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

neous models trained with seven datasets of four modalities. Similar to the compression results in §C.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 quantization, 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 quantization 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 summarize, 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.*

C.4.4 Generalizability

We now investigate the generalizability 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 the unseen models from new datasets using the same codebooks that are learned previously (§C.4.3). This can be particularly useful since the learned codebooks of YONO can still be utilized 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 §C.4.3 are not affected, ensuring high accuracy on previous datasets. Then, in §C.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 [116], (2) STL-10 [117], and two audio datasets (3) EmotionSense [6], (4) UrbanSound [72], and two HAR datasets (5) PAMAP2 [59], (6) Skoda [60], and lastly two sEMG datasets (7) Ninapro DB3 [63] and (8) Ninapro DB6 [118]. 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 [90], in our work. Then, we follow the same hyper-parameter search process as described in §C.4.2. Table C.4 summarizes 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 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 C.5 shows the accuracy results on the eight unseen datasets with diverse network architectures. 8-bit quantization presents a moderate error rate of 2.5% similar to the results in §C.4.2 and §C.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

Table C.4: Summary of datasets, model architectures, mobile applications used in §C.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

Table C.5: The compression efficiency 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

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.

Compression Efficiency. The compression results for heterogeneous models with eight unseen datasets are shown in Table C.5. The size of the uncompressed models is the largest, 4.11 MB, in this setup compared to §C.4.2 and §C.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 §C.4.2 and 11.77× in §C.4.3) without loss of accuracy on all the unseen datasets.

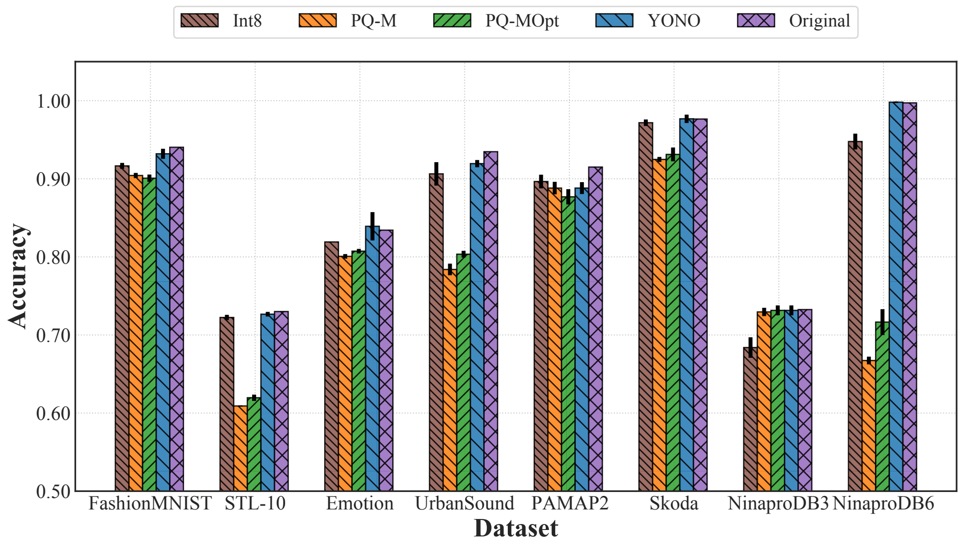
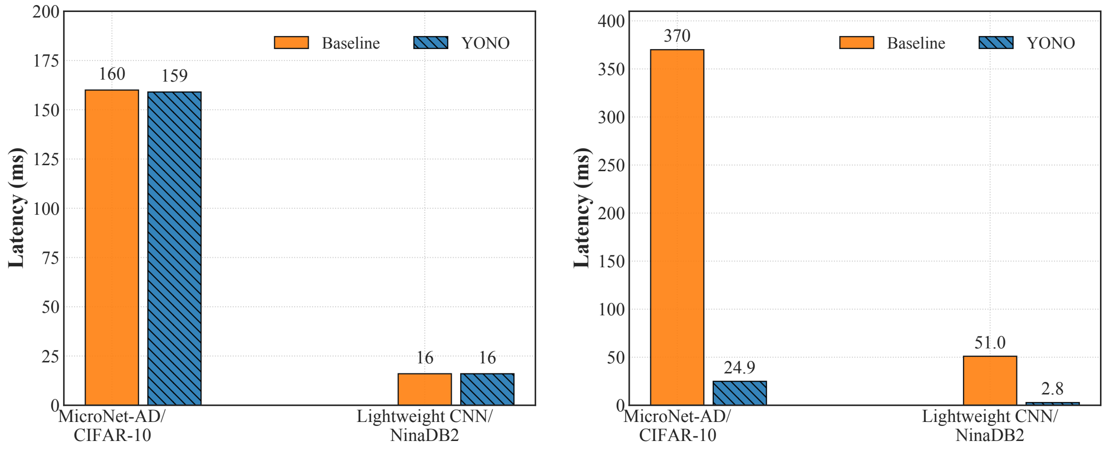


Figure C.5: The inference accuracy of the heterogeneous MTL systems applied to unseen datasets of four modalities. Reported results are averaged over five trials, and standard-deviation intervals are depicted.

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

C.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 §C.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 C.6 and C.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 §C.4.2-§C.4.4 (see Tables C.1 and C.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



(a) Execution (b) Loading/Switching

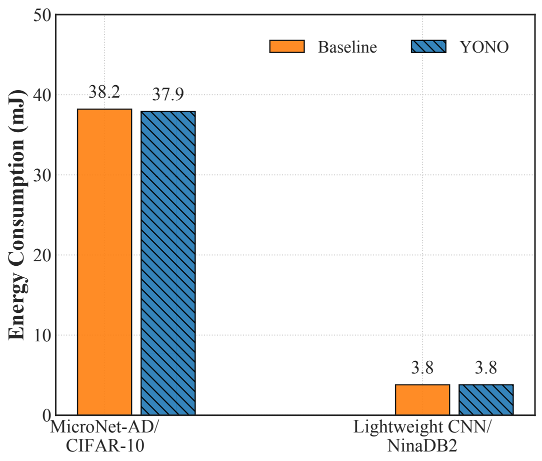
Figure C.6: The model execution and loading/switching time of YONO and the baseline.

latency and energy consumption as in Figures C.6 and C.7.

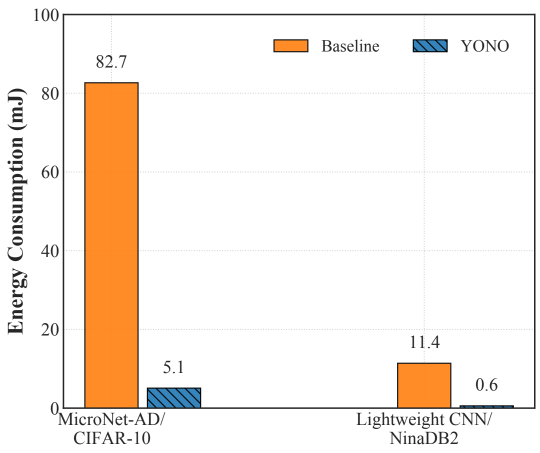
Latency. We measure the latency of the model execution and model loading/swap by using MBed Timer API, as shown in Figure C.6. In terms of execution time, both YONO and the baseline show a swift execution time (16-160 ms per inference) that can be useful in practice, and there is no meaningful latency 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 [92] 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 C.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 summarize, 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 C.7: The energy consumption of model execution and loading/switching of YONO and the baseline.

C.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 optimization 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.

Generalizability of YONO. In Section C.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 §C.4.3 and the other eight unseen datasets in §C.4.4), which shows that YONO is a generalizable framework. Other datasets and network architectures (e.g., LSTMs [23] 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 serialization format to hold model weights can be interesting future work since the weights of a model takes the majority of the space.

C.6 Related Work

Multitask Learning. Multi-task learning allows learning correlated tasks such that accuracy of both or one of the tasks is improved by exploiting the similarities and differences across tasks [119]. Common approaches include common feature learning [120, 121], low-rank parameter search [122, 123], task clustering [124, 125], and task relation learning [126, 127]. These works achieve limited compression by sharing the first few network layers. However, their main goal is to increase the robustness and generalization of multiple task learners. Thus, keeping multiple heterogeneous DNN models into 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. Comparing this, YONO allows to run multiple DNN models efficiently while remaining within the limited resource constraints on embedded devices.

Besides, NWV [92] was introduced to compress multiple heterogeneous models of different network architectures and tasks. NWV also minimizes the context switching overhead by retaining all shared weights on the 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, the work only employs a simplified LeNet architecture in the experiments of IoT use cases, and thus the accuracy of the system is limited. Conversely, YONO not only increases compression rates but compresses even the highly optimized models (e.g., MicroNet, DS-CNN), while achieving high accuracy that is useful in practice.

Mobile and Embedded Sensing Applications. Deep learning is increasingly being applied in mobile and embedded systems as it achieves state-of-the-art performances on many sensing applications such as computer vision applications [93], audio sensing [45], activity recognition [58], gesture recognition [63]. First of all, there exist many vision applications, to name a few, tiny image classification [96, 113], traffic sign recognition [114]. Besides, audio sensing application is also one of the foundational mobile sensing applications [7, 40] that much research has focused on to deliver behavioral insights to users. The audio sensing tasks include Emotion Recognition (ER) [6], Speaker Identification [128], Environmental Sound Classification (ESC) [84], and Conversation Analysis [129], and Keyword Spotting (KWS) [90]. Next, one of the most widely studied mobile sensing application is HAR [23, 86], where the aim is to determine various human activities automatically using body-worn IMU (Inertial Movement Units) sensors. In application frequently used in mobile sensing is to recognize hand gestures (e.g., fist and open palm) using sEMG (surface Electromyography) signals generated during muscle contractions [25, 47]. sEMG signal is used for medical [130], rehabilitation [131], human-computer interactions [48, 49], upper-limb prostheses control [132], and authentication [133].

Model Compression. Many researchers focus on developing a method to improve efficiency without sacrificing the model's accuracy due to a large burden of training deep network architecture and its data [100]. First of all, many researchers have focused on designing and hand-drafting more efficient network architectures, namely, SqueezeNets [134], ShuffleNets [108], and MobileNets [106, 107], and MicroNet [88]. In particular, we employ MicroNet as one of our backbone network architectures since it shows impressive performance and efficiency on tiny IoT systems such as MCUs.

In addition, another thread of research is weight pruning methods that leverage the inherent redundancy in the weights of neural networks [79, 135, 136, 137, 138, 95]. Furthermore, quantization of model weights and activiations has been an active area of research. Many prior works quantize the weights and activations from 32-bit float to 8-bit integer [9], ternary values (2-bit) [139, 140], binary values (1-bit) [141, 76, 142, 143], and mixed precision [144, 145]. Also, weight clustering methods are proposed to group weights into several clusters to compress a model.

Moreover, researchers studied techniques that quantize an array of scalars of the weights to compress a model or a particular layer. Some works extended a sparse coding [146] to learn a compact representation that covers the feature space of weights of a model [147, 148]. Also, many researchers examined vector quantization-based methods [100]. For example, Gong et al. [99] conducted an empirical study to compare binarized networks, scalar quantization using k-means (i.e., weight clustering), Product Quantization (PQ) [11]. Several recent works apply PQ to compress a deep neural network with more than 11 million parameters [101, 102, 103]. Albeit its impressive results, all the prior works only focused on utilizing PQ to a single and bulky model at a scale of millions of parameters, lacking the understanding of how the method can be used to deal with heterogeneous MTL applications and compress tiny models that should fit into the extremely limited memory budget of MCUs (less than 512 KB). Thus, for the first time in this work, we develop YONO, a PQ-based model compression framework that operates heterogeneous models on tiny IoT devices. We propose a novel network optimization procedure and heuristics to achieve high accuracy close to the uncompressed models. Also, YONO enables fast and efficient model execution and swapping on an MCU.

C.7 Conclusions

We have presented an efficient MTL system, YONO, that compresses multiple heterogeneous models through PQ codebooks, our novel network optimization 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$\times$ 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.

Acknowledgments

This work is supported by a Google Faculty Award, ERC through Project 833296 (EAR), and Nokia Bell Labs through their donation for the Centre of Mobile, Wearable Systems and Augmented Intelligence to the University of Cambridge.

Appendix D

MetaCLNet: Rehearsal-based Meta Continual Learning with Compressed Latent Replay and Neural Weights

Abstract

Continual Learning (CL) methods are designed to help deep neural networks (DNNs) adapt and learn new knowledge without forgetting previously learned information. However, current CL methods suffer from two major issues: (a) they typically require a moderate to a large number of training samples to learn new classes as in the case of traditional CL methods, and (b) Meta CL methods require a few samples of labeled training data but are still limited in performance including lower accuracy, large memory footprints and high computational costs. This limits their applicability to real-world scenarios where labeled user data is not abundant or CL needs to run on resource-constrained edge devices.

In this work, we propose MetaCLNet, a novel rehearsal-based Meta CL method, that achieves the best of both worlds: enhanced CL performance and improved system efficiency. MetaCLNet combines rehearsal techniques and meta-learning for the first time to ensure high CL performance (less forgetting, fast learning, and high accuracy). Also, to minimize resource overheads, MetaCLNet employs various optimization techniques such as compression of rehearsal samples and quantization of neural weights and activations.

MetaCLNet achieves near optimal CL performance, falling short by only 2.8% on accuracy compared to the oracle, outperforming existing Meta CL methods with substantial accuracy gains of 4.1-16.1%. Furthermore, compared to the state-of-the-art (SOTA) Meta CL method, MetaCLNet drastically reduces the memory footprint by 178.7×, end-to-end training latency by 80.8-94.2%, and energy consumption by 80.9-94.2%. We successfully deployed MetaCLNet on two edge devices, thereby enabling efficient CL on resource-constrained platforms where it is impractical to run SOTA methods.

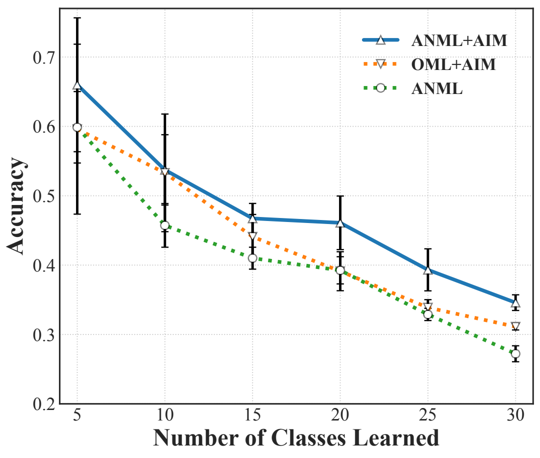
D.1 Introduction

With the rise of mobile devices, and the Internet of Things (IoT), the proliferation of sensory-type data has fostered the adoption of deep neural networks (DNN) in the modeling of a variety of mobile sensing applications [1]. A crucial characteristic common to these applications, often sitting on edge devices, is the need for a trained model to accommodate new classes and adapt to a dynamically changing environment. In such settings, the ability to *continually* learn [2, 3, 7], that is, to learn new knowledge (i.e., new classes in this work) without forgetting how to perform previously learned knowledge, becomes essential yet challenging. For example, let us consider a real-world application scenario of Continual Learning (CL) in the wild. A user has a DNN model deployed on a smartphone or an embedded device that can perform voice recognition of simple keywords such as 'yes' and 'no' or object recognition to identify simple objects from images such as 'cars' and 'buses'. Then, as time passes, the user wants the deployed model to recognize new keywords and image types to simulate the real-world scenario akin to a human who can learn new concepts continuously. Also, note that the user is reluctant to label many ground-truth samples for new keywords and images due to a high manual effort.

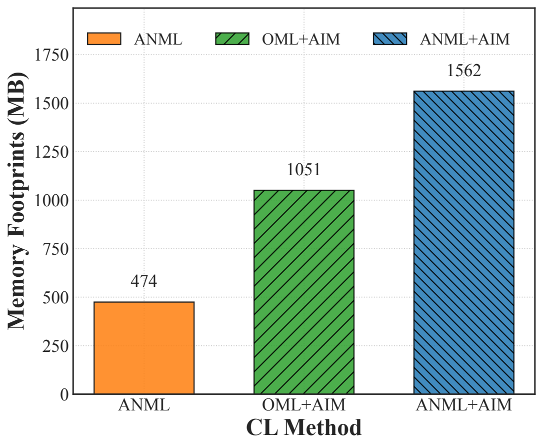
To enable Continual Learning (CL), many approaches have been proposed in the literature. This includes *regularization-based methods* [29, 36], *dynamic architecture-based methods* [149, 30, 150], and *rehearsal-based methods* [13, 14, 65]. Although these CL methods largely alleviate the forgetting issue of a learned model, they are data hungry, since a large number of labeled training samples are required to learn new information continuously. This also incurs high resource overheads: computational and memory. Hence, the applicability of the aforementioned CL methods to real-world mobile applications, where labeled user data is scarce and the computing resources are constrained, is limited.

Meta CL [151, 152, 153] has been proposed to resolve the challenges of traditional CL methods, alleviating the issues mentioned above by relying only on a few samples of new classes to adapt and learn. However, as shown in Figure D.1a, Meta CL's performance degrades when many classes are added. Additionally, state-of-the-art (SOTA) Meta CL methods such as OML+AIM and ANML+AIM [153] require a large memory footprint to perform CL, as shown in Figure D.1b. This large required memory easily exceeds the RAM size of embedded devices such as Raspberry Pi devices (e.g., 512 MB or 1 GB). Also, we observed that the end-to-end training time of SOTA Meta CL methods to conduct CL for multiple classes is very computationally expensive. These aspects make prior Meta CL methods very challenging to be deployed on resource-constrained devices.

This Work. To address these limitations and challenges, we propose and develop a CL system, **MetaCLNet**, that achieves: (1) superior accuracy in continually learning new classes without forgetting existing classes with only a few training samples (10-30) close to the upper bound performance based on i.i.d. training, (2) drastically lower system



(a) Performance



(b) Memory Overhead

Figure D.1: Preliminary analysis of the Meta CL methods.

overheads (e.g., memory footprint) compared to SOTA and (3) fast learning for rapid online deployment on the edge devices.

Rehearsal-based Meta CL. To solve the accuracy degradation problem of the Meta CL methods, MetaCLNet combines Meta CL with rehearsal-based replay. In prior Meta CL methods, the given samples for learning new classes are discarded once they are used for training. Conversely, recognizing the importance of such samples, MetaCLNet stores them (exemplars) to prevent forgetting when it encounters and learns new classes by replaying the saved samples (raw data or latent representations) of learned classes (see Sections D.2.3 and D.2.4).

Compressed Latent Replay. We adopt various optimization techniques to minimize the resource overheads of our system. We use a latent representation of the rehearsal samples instead of the raw data themselves. Also, as 90% of the values of the latent representations are zeros due to ReLU non-linearity, we utilize the sparse bitmap compression [154] to store the indices of the non-zero values and discard the zero values (Section D.2.5). To maximize the compression rate even further, MetaCLNet applies Product Quantization (PQ) [11] on the non-zero values of latent representations after the sparse bitmap compression (Section D.2.6).

Quantized Neural Weights/Activations. On top of the sparse bitmap compression and PQ, we further optimize our system by quantizing the weights and activations of the feature extraction network frozen during the deployment phase. Neural weight quantization decreases the model size and latency to perform inference. Freezing the network enables rapid on-device learning of new classes on the fly as MetaCLNet can bypass the execution of the feature extraction part during training and update the classifier part with the compressed samples for rehearsal, which is computationally lightweight than updating the whole network (Section D.2.7).

To evaluate MetaCLNet, we employ two image datasets, CIFAR-100 [96] and MiniImageNet [155], used in prior SOTA Meta CL work [153] (as this allows us to compare performance fairly) and an audio dataset, Google Speech Command V2 (GSCv2) [115] to show generalization to another type of data. We show that MetaCLNet achieves near optimal CL performance, falling short by only 2.8% accuracy compared to the upper bound performance (oracle) in our evaluation. MetaCLNet also outperforms current Meta CL methods with substantial accuracy gains of 4.1-16.1% on average for the three datasets, demonstrating the effectiveness of co-utilization of Meta CL and rehearsal-based learning. MetaCLNet adopts various compression techniques (i.e., sparse bitmap compression and PQ) that can effectively reduce the burden of storing replay samples (either raw data or latent representations). Moreover, quantizing the feature extractor of the network into 8-bit integers further decreases the resource requirements (i.e., memory footprint) of MetaCLNet. As a result, to perform CL on all three datasets, MetaCLNet requires only 3.40-15.45 MB of memory and thereby obtains a compression rate of 11.4-178.7$\times$ compared to ANML+AIM (SOTA MetaCL method).

We successfully deployed MetaCLNet on two edge devices (Jetson Nano and Pi 3B+), enabling more efficient CL on resource-constrained platforms on which SOTA does not always fit due to their large memory requirements. Further, MetaCLNet decreases the end-to-end training time by 80.8-94.2% and the overall energy consumption on the edge devices by 80.9-94.2% compared to ANML+AIM. Furthermore, we perform an ablation study to identify the impact of each proposed component in our system and a parameter analysis to examine the effect of the various hyper-parameters that could affect the CL performance of our system (Section D.4).

The contributions of MetaCLNet are summarized as follows.

- To the best of our knowledge, MetaCLNet is the first CL system that combines Meta CL and rehearsal-based CL strategy to develop a rehearsal-based Meta CL system that achieves high accuracy.
- We employ several optimization techniques including sparse bitmap compression and PQ on the latent representations of rehearsal samples. On top of such optimization, further quantizing neural weights and activations of the feature extractor of the deployed DNN models allows MetaCLNet to run on the edge devices directly, which is infeasible for SOTA to run due to its excessive memory requirements.
- Through extensive experiments on Jetson Nano and PI 3B+, we demonstrate that MetaCLNet outperforms existing Meta CL baselines in terms of latency and energy

consumption.

- Finally, the ablation study reveals that each proposed component of this work is effective in making our CL system more efficient. Also, the parameter analysis demonstrates that MetaCLNet can achieve similar CL performance to the upper bound with only 10-30 samples per class, show rapid convergence with one or two epochs, and accomplish a high compression rate for latent representations.

D.2 Design

In this section, we begin by introducing the problem formulation of CL in more detail and the overview of MetaCLNet (§D.2.1). We then describe Meta CL and relevant prior works (§D.2.2). After that, we propose a novel rehearsal-based Meta CL method that can effectively address the performance degradation issues of existing Meta CL methods (§D.2.3). Also, we describe two rehearsal techniques that are often utilized in the CL literature (§D.2.4). In addition, we present two techniques for optimizing the resource overheads incurred by the saved rehearsal samples (§D.2.5 and §D.2.6). On top of that, we further quantize the model itself to minimize the overall memory footprint required for performing CL during deployment (§D.2.7).

D.2.1 Problem Formulation & System Overview

Problem Formulation. In this work, we employ the Sequential Learning Tasks (SLTs) [32] where new classes can emerge over time. Hence, the CL needs to continually learn new classes without forgetting the previously learned classes, similar to real-world application scenarios. In other words, SLTs indicate that a CL model continuously learns a stream of classes, $T_1, T_2, ..., T_t, ..., T_N$, one after another instead of learning all the classes at once, which is similar to the class-incremental learning setting [53]. N is the number of classes, and each T_t consists of (X_t, Y_t) for inputs X_t and target labels Y_t from sets $\mathcal{X}$ and $\mathcal{Y}$, respectively. A CL model g parameterized by θ learns a mapping, $g_\theta : \mathcal{X} \to \mathcal{Y}$.

For a given CL problem of SLTs, the goal of CL methods is to minimize the following objective for some loss function $\ell(\cdot, \cdot)$:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \mathbb{E}_{\mathbf{x},\mathbf{y} \sim T_i} \ell(g_\theta(\mathbf{x}), \mathbf{y}) \tag{D.1}$$

where θ represents a set of weights that are updated to minimize the objective. In this work, we focus on the classification problem. The frequently used symbols are listed in Table D.1.

Table D.1: Frequently-used symbols and definitions.

Symbol	Definition
$T = (X, Y)$	class with inputs X and targets Y
T_t	(t)-th class of a sequence of classes
X_t, Y_t	inputs and targets of (t)-th class
$\mathbf{x}_i^t, \mathbf{y}_i^t$	(i)-th entry of inputs and targets of (t)-th class
N	total number of classes in a given sequence
$\lVert\cdot\rVert, \odot$	l_2 norm, Hadamard product
f_θ	feature extractor parameterized by θ
ϕ_θ	classifier parameterized by θ
θ^{NM}	weights of neuromodulation network
θ^{P}	weights of prediction network
θ^{CLF}	weights of classifier network
θ^{W}	weights of AIM
θ^{RLN}	weights of representation learning network
θ^{PLN}	weights of prediction learning network

System Overview. we now describe the overview of MetaCLNet that combines the idea of Meta CL and rehearsal-based learning and optimizes the system efficiency in terms of the memory footprint, computational costs, and energy consumption on the embedded devices. MetaCLNet is comprised of two phases. The first phase, called meta-training, is performed on a server to find the good weight initialization by utilizing meta-learning in the CL setup with a few samples. The second phase is meta-testing where a meta-learned model is deployed on embedded devices and learns new classes continually. In addition, as shown in Figure D.2, there exist three components in MetaCLNet to ensure excellent performance and efficiency when it is deployed on resource-constrained devices: (1) co-utilization of Meta CL and rehearsal strategy to resolve the accuracy degradation issue, (2) adoption of compression techniques such as the sparse bitmap compression and PQ to reduce the memory footprint of rehearsal samples, and (3) quantization of weights and activations of the feature extractor to reduce the memory footprint, computational costs, and energy consumption.

D.2.2 Meta Continual Learning

We now provide an introduction to Meta CL. Then, we explain how the learning procedure of meta-learning is utilized in the context of the CL problem and helpful for mitigating the forgetting issue by reviewing three of the state-of-the-art Meta CL methods [151, 152, 153].

Given a single trajectory of samples from a stream of classes $\mathcal{T}$, minimizing the CL loss of

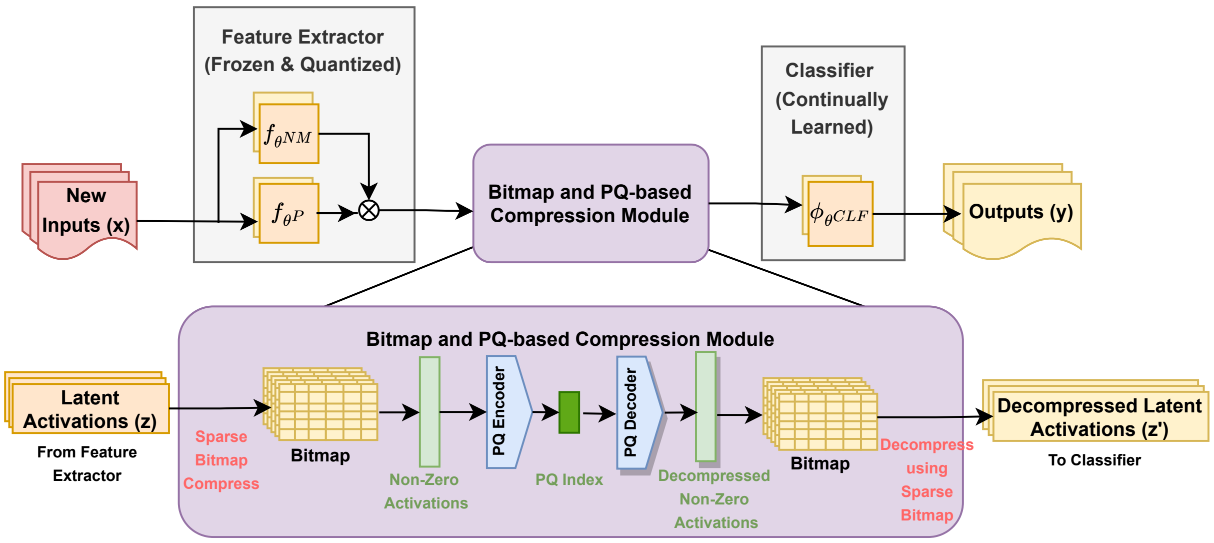

Feature Extractor (Frozen & Quantized)
$f_{\theta NM}$
$f_{\theta P}$
New Inputs (x)
Bitmap and PQ-based Compression Module
Classifier (Continually Learned)
$\phi_{\theta CLF}$
Outputs (y)
Bitmap and PQ-based Compression Module
Latent Activations (z)
From Feature Extractor
Sparse Bitmap Compress
Bitmap
Non-Zero Activations
PQ Encoder
PQ Index
PQ Decoder
Decompressed Non-Zero Activations
Bitmap
Decompress using Sparse Bitmap
Decompressed Latent Activations (z')
To Classifier

Figure D.2: The overview of the proposed CL system, MetaCLNet.

DNN that is trained end-to-end is very challenging [151]. First of all, learning a stream of different classes incurs catastrophic forgetting (CF) [27] (i.e., DNN forgets previously learned classes when it starts to learn a new class). Moreover, training DNNs is extremely sample-inefficient: the minimization problem requires multiple training epochs to converge to a reasonable solution. On one hand, many CL methods [150, 14, 29] are proposed to alleviate the forgetting problem. However, they require a moderate or large amount of labeled data and many training epochs. On the other hand, another learning approach, called meta-learning, is proposed to make DNN more sample-efficient [156], requiring only a few samples to adapt or learn new data distributions from a correlated stream of data [157, 158]. However, these meta-learning methods neglect the forgetting problem of the already learned classes.

To overcome the challenges mentioned thus far, several Meta CL methods [151, 152, 153] are proposed. First, to enable fast adaptation with only a few samples, Meta CL methods are based on the training procedure of meta-learning. The meta-learning uses an outer loop and an inner loop where the outer loop takes steps to improve the learning ability of the inner loop that optimizes the DNN model with a few samples (see Algorithm 4 for detail). This phase is called *meta-training* and aims to find a better weight initialization of DNNs for fast adaptation with a few samples. After the meta-training is finished, the quality of the learned DNNs is tested using only inner loop updates without outer loop updates. This phase is referred to as *meta-testing* (see Algorithm 5 for detail).

Second, to prevent the forgetting problem, Meta CL methods separate the network architecture into the feature extractor and the classifier. This separation of the feature extraction and the classifier parts is the common network architecture of the Meta CL methods such as OML, ANML, and Attentive Independent Mechanisms (AIM) methods (OML+AIM and ANML+AIM). Then, Meta CL adopts the concept of fast and slow learning on an architecture level: the feature extractor is updated in the outer loop (slow weights) to prevent forgetting, and the classifier is updated in the inner loop (fast weights) to learn new classes swiftly. This approach has proven to be useful in preventing CF over many classes. In addition, the network architectures of prior Meta CL methods are represented as follows. For OML, the feature extractor is denoted as a representation learning network (RLN), $f_{\theta^{RLN}}$. Then, the classifier is referred to as a prediction learning network (PLN), $\phi_{\theta^{PLN}}$. Note that AIM layers are parameterized by θ^W and are inserted between the feature extractor and the classifier. The output of OML+AIM, $\hat{\mathbf{y}}$, is computed as follows.

$$\hat{\mathbf{y}} = \phi_{\theta^{PLN}}(f_{\theta^W}(f_{\theta^{RLN}}(\mathbf{x}))) \tag{D.2}$$

Next, ANML is composed of three networks: a neuromodulatory network, $f_{\theta^{NM}}$, and a prediction network, f_{θ^P} that constructs the feature extractor, followed by the classifier part, $f_{\theta^{CLF}}$. Within the feature extractor, the neuromodulatory network is to modulate the outputs of the prediction networks by performing the Hadamard product between the outputs of those two networks before passing it to the classifier. Like OML+AIM, ANML+AIM has AIM layers inserted between the feature extractor and the classifier. The output of ANML+AIM, $\hat{\mathbf{y}}$, is then calculated as follows.

$$\hat{\mathbf{y}} = \phi_{\theta^{CLF}}(f_{\theta^W}(f_{\theta^{NM}}(\mathbf{x}) \odot f_{\theta^P}(\mathbf{x}))) \tag{D.3}$$

D.2.3 MetaCLNet

As described in §D.2.2, current Meta CL methods are useful in mitigating the forgetting problem, however, they all come with some limitations. For example, while OML and ANML help alleviate the CF issue compared to the case without any CL applied, these methods often fail to maintain high CL performance. However, OML+AIM and ANML+AIM require remarkably more model parameters (e.g., 3-15×) than OML and ANML, increasing the memory footprint for training. Therefore, we propose our novel rehearsal-based Meta CL system, MetaCLNet, to address these limitations. This subsection describes how we combine the Meta CL and rehearsal strategy to solve the forgetting problem and meanwhile ensure high performance.

Given a stream of classes, $\mathcal{T}$, the aim of Meta CL is to learn new classes one after another with only a few samples (e.g., 30) instead of training all the classes at once in i.i.d. fashion [7]. In prior Meta CL methods, the given samples are discarded once they are used for training. However, according to the CL literature [159], rehearsal-based CL methods often outperform other types of CL methods such as regularization-based and architectural-based CL methods by identifying and saving representative samples (i.e., exemplars) and replaying these stored samples while learning new classes. Inspired by rehearsal-based CL, we propose a novel Meta CL method called rehearsal-based Meta CL by incorporating rehearsal strategy into the Meta CL. In detail, MetaCLNet stores the given samples and then replays these saved samples for rehearsal to solve forgetting issues (for previously learned classes) when learning new classes. Besides, MetaCLNet employs the network architecture of the prior Meta CL work, ANML, consisting of the feature extractor and the final classifier. Also, MetaCLNet chooses the last layer of the feature extraction part as the latent replay layer and stores the activations of the layer as rehearsal samples as MetaCLNet eventually quantizes and freezes the feature extraction part during deployment as explained in Section D.2.7. We now explain the meta-training and meta-testing procedures of MetaCLNet in more detail.

Meta-Training and Meta-Testing Procedures. Algorithm 4 shows the procedure of meta-training of Rehearsal-based Meta CL, MetaCLNet. First of all, the meta-training process of rehearsal-based Meta CL is the same as that of Meta CL [152]. In detail, it is comprised of an inner loop inside an outer loop of optimization. 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 given samples k (e.g., 10-30) of S_{traj}. Following the k sequential updates on a single meta-training class, the meta-loss in the outer loop (Line 6) is computed by making predictions using the weights after the last inner-loop weight update iteration using all the given samples on the single class (S_{traj}) plus randomly sampling additional samples from the set of all the meta-training classes (S_{rand}). All the weights of DNN are updated through outer-loop gradient updates based on ADAM [85]. The learning rates, α for the inner loop and β for the outer loop, are used as hyper-parameters.

After executing the meta-training phase on a server, in the meta-test phase, our system is deployed on resource-constrained devices and evaluated on its ability to learn unseen classes. Algorithm 5 shows the meta-testing phase of the rehearsal-based Meta CL. In conventional Meta CL, the meta-test procedure contains only inner-loop optimization without outer-loop optimization, i.e., only fast weights except for slow weights are fine-tuned. It is because prior Meta CL methods do not adopt rehearsal techniques and often the feature extractor is frozen during the meta-test phase. However, in our rehearsal-based Meta CL, we replicate the meta-training procedure (i.e., inner-loop and outer-loop optimization) in the meta-test procedure to utilize the inner-loop update for the fast adaptation of new classes and the outer-loop update to prevent forgetting of previously learned classes. Thus, our proposed meta-test process starts with the inner-loop weight updates to learn new classes swiftly using a few samples (Lines 5-6), then followed by the outer-loop weight updates to retain the knowledge on the previously learned classes using the replayed exemplars plus the new samples (Line 8). Note that although the outer-loop iteration could run multiple epochs,

Algorithm 4: Meta-Training Procedure of MetaCLNet

```
Require: $N$ sequential classes $\mathcal{T}$; learning rates (LR) $\alpha$, $\beta$; inner-loop iterations $k$;
         modules $f_\theta, \phi_\theta$
/* Perform an outer-loop using meta-learning */
1 for $t = 1, ..., N$ do
2     $S_{traj} \sim \mathcal{T}_t$
3     $S_{rand} \sim \mathcal{T}$
      /* Perform an inner-loop */
4     for $i = 1, ..., k$ do
5         Update fast weights using $S_{traj}$        ▷ LR: $\alpha$
          /* OML(+AIM):$\phi_{\theta^{PLN}}(f_{\theta^W})$, ANML(+AIM):$f_{\theta^P}, \phi_{\theta^{CLF}}(f_{\theta^W})$, MetaCLNet(Ours): $\phi_{\theta^{CLF}}$ */
6     Update slow weights using $\{S_{traj}, S_{rand}\}$        ▷ LR: $\beta$
      /* OML(+AIM):$f_{\theta^{RLN}}$, ANML(+AIM): $f_{\theta^{NM}}, f_{\theta^P}, \phi_{\theta^{CLF}}$, MetaCLNet(Ours): $f_{\theta^{NM}}, f_{\theta^P}, \phi_{\theta^{CLF}}$ */
```

the performance converges after one or two epochs (see §D.4.6 for details). In addition, to reduce the memory overheads, our proposed compression technique for rehearsal samples (BitPQ: the combination of the sparse bitmap compression and PQ) is performed on the fly before and after the slow weight update (Lines 7, 9-10). Further details of rehearsal techniques (§D.2.4) and the compression methods for rehearsal samples (§D.2.5 and §D.2.6) are presented in the following subsections.

D.2.4 Rehearsal Techniques

We now describe two widely used rehearsal techniques: native rehearsal and latent replay.

Native Rehearsal. This is a rehearsal technique in which a random subset of the given classes is stored as rehearsal samples to be replayed later to mitigate the forgetting issue. The raw input data for the model are stored for rehearsal. For example, images are stored for image datasets, and MFCC features are stored for an audio dataset. Storing and replaying the raw input data is often adopted in many rehearsal-based CL methods [13, 31, 160].

Latent Replay. This approach is to store latent representations of a selected layer in DNNs, instead of storing copies of raw input data. Note that typically in DNNs, layers close to the input represent low-level feature extraction, and layers close to the classifier represent class-specific discriminant features. In our network architecture, the activations of the layer after the feature extractor and before the classifier are stored as rehearsal samples. When the feature extractor is frozen, latent replay is functionally equivalent to raw input replays. However, latent replay achieves computational saving since the forward pass of the feature extractor can be omitted when replaying latent representations.

Algorithm 5: Meta-Testing Procedure of MetaCLNet

```
Require: $N$ sequential unseen classes $\mathcal{T}$; learning rates (LR) $\alpha$, $\beta$; inner-loop
         iterations $k$; modules $f_\theta, \phi_\theta$, $BitPQ_{compress}, BitPQ_{decompress}$
1  $S_{train} = \{\}$, $S_{rehearsal} = \{\}$
   /* Perform an outer-loop using meta-learning                                      */
2  for $t = 1, ..., N$ do
3      $S_{traj} \sim \mathcal{T}_t$
4      $S_{train} = \{S_{train}, S_{traj}\}$
       /* Perform an inner-loop                                                       */
5      for $i = 1, ..., k$ do
6          Update fast weights using $S_{traj}$                ▷ LR: $\alpha$
           /* OML(+AIM):$\phi_{\theta^{PLN}}(f_{\theta^W})$, ANML(+AIM):$f_{\theta^P}, \phi_{\theta^{CLF}}(f_{\theta^W})$, MetaCLNet(Ours): $\phi_{\theta^{CLF}}$
              */
       // Get latent activations from compressed rehearsal samples
7      $S_{latent} = BitPQ_{decompress}(S_{rehearsal})$
8      Update slow weights using $\{S_{traj}, S_{latent}\}$      ▷ LR: $\beta$
       /* OML(+AIM):$f_{\theta^{RLN}}$, ANML(+AIM): $f_{\theta^{NM}}, f_{\theta^P}, \phi_{\theta^{CLF}}$, MetaCLNet(Ours): $\phi_{\theta^{CLF}}$       */
       // Get latent activations
9      $S_{latent} = f_{\theta^{NM}}(S_{traj}) \odot f_{\theta^P}(S_{traj})$
       // Store compressed latent activations for rehearsal
10     $S_{rehearsal} = \{S_{rehearsal},\ BitPQ_{compress}(S_{latent})\}$
11 $S_{test} = \mathcal{T} - S_{train}$                         // Held-out meta-test testing set
12 Evaluate on $S_{train}$                                  // Eval on meta-test training set
13 Evaluate on $S_{test}$                                    // Eval on meta-test testing set
```

Also, the backward pass is performed until the latent layer for the latent representations. Therefore, we adopt to use the latent replay as a rehearsal technique for MetaCLNet.

D.2.5 Sparse Bitmap Compression for Latent Replays

MetaCLNet achieves high accuracy in CL tasks. However, stored samples for rehearsal cause storage and memory overheads while learning new classes. Therefore, we adopt two compression techniques (sparse bitmap compression and PQ) to minimize the resource overheads of storing samples for rehearsals. In this subsection, we first describe the sparse bitmap compression.

Sparsity Observation. In DNN training, the activations for each layer are saved during the forward pass so that those activations are utilized for computing the gradients during the backward pass. As in [161], storing activations requires a large memory footprint depending on the batch size used for training. However, commonly used ReLU non-linearity

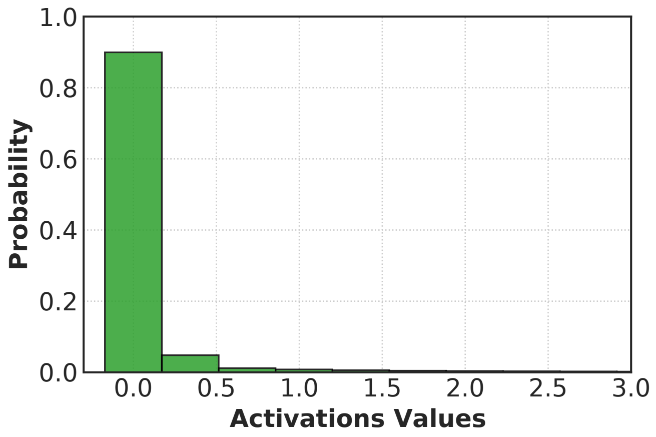


Figure D.3: The histogram of activation values (latent representations) for the latent layer of the model trained on the GSCv2 dataset (The same observation holds for the other employed datasets). More than 90% of the activation values are zero, and thus the proportion of non-zero values is very small due to the 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 (see Figure D.3).

Compression. To leverage this observation, we employ the sparse bitmap compression proposed in [154] where zero values of the activations are filtered out, and only non-zero values with corresponding indices are stored in 32-bit floats and the bitmap format, respectively. This scheme enables our system to filter out the majority of zero values (90% or more) and save the remaining non-zero values to increase the compression rate for saving latent representations. Specifically, when latent representations that will be stored as rehearsal samples are given to the system, the original latent representations are compressed into non-zero values in their original format (32-bit) and a bitmap that has the same dimensions as the latent representations and sets a bit to 1 for the indices having the non-zero values and to 0 for the remaining indices with zero.

The sparse bitmap compression is performed during the replay process on-the-fly. As illustrated in Figure D.2, when latent activations for a new class are given, we perform the sparse bitmap compression on them and store non-zero values and the corresponding bitmap. Then, during the replay process, we reconstruct the latent representations using the stored non-zero values and the bitmap. Going through each element of the bitmap (containing bits of 1 or 0) and a vector containing the stored non-zero values, we reconstruct latent activations by putting the saved non-zero value if a bitmap element is 1 or putting zero if a bitmap element is 0. The compression and decompression processes are linear in runtime, i.e., $O(n)$ where n is the total number of elements of latent activations. Also, the memory footprint can be reduced from $(4n)$ when a dense format is used for storing latent activations to $(4 \times \text{number of non-zero values} + \frac{1}{8}n)$ when the bitmap format is used.

D.2.6 Product Quantization for Compressing Latent Replays

We now introduce PQ [11] and how it is used to further compress the latent representations together with the sparse bitmap compression in our system. First of all, given a vector $\mathbf{z} \in \mathbb{R}^d$ where d is the dimension of the vector (e.g., a vector with only non-zero values after the sparse bitmap compression in MetaCLNet), PQ attempts to store $\mathbf{z}$ as s number of indices (integers) using as few bits as possible (typically 8-bit is used so that each index can be stored with 1 byte). In detail, PQ partitions d-dimensional vector, $\mathbf{z}$, into s sub-vectors with the size of d/s. After that, suppose we are given a PQ codebook that is partitioned into s columns and each column contains a set of representative vectors that can well approximate sub-vectors of $\mathbf{z}$, the given vector $\mathbf{z}$ can be approximated by s sub-vectors using the representative vectors in the PQ codebook and its associated indices. Then, with the codebook and s indices, we can reconstruct the given vector $\mathbf{z}$. Note that a codebook can be learned by running the standard k-means clustering on each partitioned column independently based on all the given vectors [11].

In this work, we apply PQ to the compressed latent activations already filtered out by the sparse bitmap compression and hence contain only non-zero values, as shown in Figure D.2. To learn a PQ codebook, we use the compressed latent representations of the training data of the meta-training phase so that the codebook contains representative vectors that could well approximate compressed latent representations after the sparse bitmap compression. For our experiments, we use 1 byte to store each PQ index (i.e., index can be ranged from 0 to 255 in this case) and set $d/s = \{128, 32, 8\}$ (d/s is the length of each sub-vector. It can be considered the compression rate as a sub-vector of length d/s is compressed to a 1 byte integer in this case) (refer to §D.4.6 for more analysis).

D.2.7 Quantizing Neural Weights

Having established the effective compression schemes on stored latent representations, we further optimize MetaCLNet by performing quantization on the DNN model itself. As our network architecture is naturally divided into the feature extractor frozen during the deployment and the classifier updated continually while learning new classes (as shown in Figure D.2), we determined to quantize the feature extractor by converting its weights and activations from 32-bit floats to 8-bit integers using the scalar quantization scheme [9, 111] to minimize the information loss in quantization. We utilize an affine mapping of integer q to real number r for constant quantization parameters S and Z, i.e., $r = S(q - Z)$. S denotes the scale of an arbitrary positive real number. Z denotes a zero-point of the same type as quantized value q, corresponding to the real value 0. As a result, the feature extractor of the deployed DNN is based on 8-bit integers, and its classifier is based on 32-bit floats continually fine-tuned during deployment. Note that the neural weight quantization significantly decreases the model size (smaller memory footprint for CL) and makes inference faster (lower end-to-end training time for learning new classes continually). Besides, our design choice of network architecture (quantizing and freezing the feature extractor while updating the classifier) enables rapid on-device learning of new classes on the fly as MetaCLNet can bypass the execution of the feature extractor during training and only update the classifier with the compressed rehearsal samples, which is computationally much lightweight than updating the whole networks.

D.3 Implementation

We now introduce the hardware and software implementation.

Hardware Platform. The meta-training stage of our system to initialize the neural weights that can enable fast adaptation during deployment scenarios is implemented and tested on a Linux server equipped with an Intel Xeon Gold 5218 CPU and NVIDIA Quadro RTX 8000 GPU. The weights of DNN models based on ANML and AIM are initialized through meta-learning in this stage. After that, in the deployment stage, we deploy the pre-trained models (i.e., feature extractor and classifier) on two embedded devices such as Jetson Nano and Pi 3B+. The first device, Jetson Nano, is equipped with a quad-core ARM Cortex-A57 processor with 4 GB of RAM. The second device, 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 and concurrent applications and operating systems.

Software Platform. We employ PyTorch 1.8 (Deep Learning Framework) and Faiss (PQ Framework) to develop and evaluate the meta-training phase of MetaCLNet on the Linux server. We implement MetaCLNet based on Python on the server and examine the accuracy of the models in the CL setup. In addition, for the meta-testing phase (i.e., actual deployment scenarios), MetaCLNet is deployed on embedded devices with limited resources. Also, we quantize the weights/activations of the feature extractor (i.e., neuromodulatory network and prediction network) of our system from 32-bit floats to 8-bit integers. Then, we utilize the QNNPACK backend engine of PyTorch to execute the quantized model on two embedded devices with ARMv8 microarchitecture. Only the classifier is performed using 32-bit floats.

D.4 Evaluation

We now present the evaluation results of MetaCLNet. We first describe the experimental setup (§D.4.1). We then show the effectiveness of our system by comparing it with other baselines in terms of accuracy on CIFAR-100, MiniImageNet, and Google Speech Commands V2 datasets (§D.4.2). After that, we analyze the memory footprint required to operate various baselines and MetaCLNet (§D.4.3). In addition, we show the results regarding end-to-end latency and energy consumption (§D.4.4). Finally, we conducted an ablation study to identify the impact of each component of MetaCLNet (§D.4.5) and did parameter analysis to study the effects of hyper-parameters such as the number of given samples (§D.4.6) on the performance.

D.4.1 Experimental Setup

Evaluation Metrics

Within the meta-testing stage, there is the training phase where the DNN model learns new classes and the testing phase where we evaluate the updated DNN, denoted as meta-test training and meta-test testing, respectively. As discussed in [152], meta-test training performance indicates the ability to memorize already seen samples of new classes. Then, the meta-test testing performance measures the generalization ability for unseen samples of new classes and is used as the key performance metric in our work. Moreover, we report the memory footprint required to perform CL over multiple classes during the meta-test phase, including memory space for model parameters, optimizers, activations, and rehearsal samples. Also, we measure the end-to-end latency and energy consumption to continually learn all the given classes to a deployed model on embedded devices.

Baselines

The following baseline systems are compared with our proposed system, **MetaCLNet**.

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 initializes the model weights based on conventional DNN training without the meta-learning procedure. Then, it finetunes the model weights using given samples in the meta-test phase, similar to other Meta CL methods.

OML+AIM [153]: OML+AIM is a Meta CL method based on OML with an Attentive Independent Mechanisms (AIM) module that captures independent concepts to learn new knowledge.

ANML [152]: ANML is the representative Meta CL method. As this method is often reported to outperform OML, we only employ ANML in our evaluation. Also, note that the proposed components of MetaCLNet build on top of ANML.

ANML+AIM [153]: 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.

Datasets

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

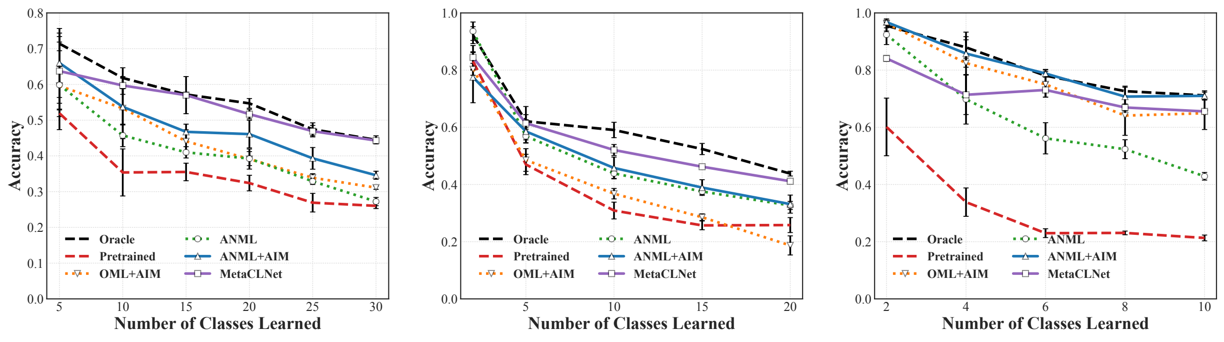
CIFAR-100 [96]: Following [153], we employ CIFAR-100 in our evaluation as it is widely used dataset. CIFAR-100 consists of 60,000 images of 100 classes. Each class has 500 train images and 100 test images. 70 classes are used for meta-training and the remaining 30 for meta-testing. During both meta-training 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, accounting for only 2.57% of all training samples.

MiniImageNet [155]: Following [153], we employ MiniImageNet which contains 64 classes for meta-training and 20 classes for meta-testing. Then, each class has 540 images for training and 60 images for testing. During meta-testing, a total of 600 samples are given to perform CL, taking up only 1.74% of all training samples.

GSCv2 [115]: To generalize 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, accounting for only 0.5% of all training samples.

Model Architecture

We follow the network architecture used in prior work [153] to allow for a fair comparison. For ANML plus other systems based on ANML architecture, the neuromodulatory network $f_{\theta^{NM}}$ and prediction network $f_{\theta^{P}}$ are a 3-layer convolutional network with 112 and 256 channels, respectively. The classifier $\phi_{\theta^{CLF}}$ has a single fully-connected layer. ANML+AIM adds AIM layers $f_{\theta^{W}}$ after the feature extractor and before the classifier. For OML and OML+AIM, the feature extractor $f_{\theta^{RLN}}$ has a 6-layer convolutional network with 112 channels, followed by the classifier $\phi_{\theta^{PLN}}$ of two fully-connected layers with an AIM module between the feature extractor and the classifier.



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

Figure D.4: The meta-test testing 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.

Table D.2: The comparison of the required memory footprint and the compression ratio for the baselines and our system to perform CL during the meta-test phase on three datasets.

Dataset	**Metrics**	**Pretrained**	**ANML**	**OML+AIM**	**ANML+AIM**	**Oracle**	**MetaCLNet**
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×**
MiniImageNet	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×**

Training Details

We follow the meta-learning procedure used in prior Meta CL works [151, 152, 153]. For instance, we use the batch size of 1 for 20,000 steps. We experimented with different learning rates for the inner loop and outer loop. For CIFAR-100 and GSCv2 datasets, the inner-loop learning rate α of 0.001 and the outer-loop learning rate β of 0.001 show the best meta-training test accuracy. For the MiniImageNet dataset, the inner-loop learning rate α of 0.001 and the outer-loop learning rate β of 0.0005 shows the best meta-training test accuracy. During the meta-test phase, ten different learning rates are tried for all the methods. Also, different hyper-parameters for MetaCLNet (e.g., the number of given samples, replay epochs, and sub-vector length of the PQ codebook) are evaluated. The best performing results are reported in the following subsections.

D.4.2 Accuracy

We start by evaluating the CL performance (meta-test testing accuracy) of MetaCLNet in comparison to the baselines on the employed datasets. During the meta-test phase, all evaluated systems are given only 30 samples per class, accounting for 2.57%, 1.74%, and 0.5% of all training samples during meta-training of CIFAR-100, MiniImageNet, and GSCv2, respectively. Figure D.4 presents the accuracy results of meta-test testing to analyze how well the baselines and our system can retain the knowledge of the previously learned classes and generalize to unseen samples of new classes. To begin with, Pretrained and Oracle serve as the performance lower bound and upper bound, respectively. The low meta-test testing 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. Prior Meta CL method, 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). Moreover, even the SOTA Meta CL methods, ANML+AIM (OML+AIM), also show a substantial accuracy drop of 9.9% (13.4%) for CIFAR-100 and 10.7% (25.1%) for MiniImageNet compared to Oracle. For GSCv2, ANML+AIM (OML+AIM) shows impressive results whose meta-test testing accuracy is 71.0% (64.9%), very close to Oracle.

MetaCLNet achieves near optimal CL performance, falling short of only 2.8% accuracy compared to Oracle (the upper bound of our experiments). Also, MetaCLNet outperforms all the Meta CL methods with substantial accuracy gains of 4.1-16.1% on average for the three datasets. Specifically, for CIFAR-100 and MiniImageNet datasets, MetaCLNet shows almost no loss of accuracy (0.2% for CIFAR-100 and 2.7% for MiniImageNet) compared to Oracle, while ANML+AIM (SOTA Meta CL method) shows notable accuracy drops (9.9% for CIFAR-100 and 10.7% for MiniImageNet). Then, for GSCv2, MetaCLNet reveals a slight accuracy loss of 5.6% compared to Oracle, while ANML+AIM shows an impressive result of only 0.2% accuracy loss compared to Oracle. Although MetaCLNet shows a slightly higher accuracy loss than SOTA for GSCv2, our system is essentially designed for edge devices to require drastically lower system resources (memory, training latency, and energy) than SOTA. As we will explain in the following subsections, the excessive resource overhead of SOTA makes it not suitable to operate on resource-constrained devices.

In addition, we observed that the meta-test training accuracy of the baselines and our system is generally higher than the meta-test testing accuracy. This result indicates that the CL systems can remember samples of new classes that they have just learned but being generalizable to unseen samples of new classes is still challenging. 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. Hence, we employ a batch size of 8 in our evaluation as a smaller batch size can reduce the required memory footprint.

Overall, this result indicates that MetaCLNet can effectively learn new classes in a continual manner based only on a few shots (30 samples per class) without experiencing catastrophic forgetting, i.e., it generalizes well to new samples of many classes unseen during the meta-train phase.

D.4.3 Memory Footprint

We now investigate the memory footprint required to perform CL over many unseen classes during the meta-test phase. Precisely, we measure the memory space required to perform training, i.e., backpropagation and replayed rehearsal samples. The memory requirement for training consists of three components: (1) model memory that stores model parameters, (2) optimizer memory that stores gradients and momentum vectors, and (3) activation memory that is comprised of the intermediate activations (stored for reuse during backpropagation). In addition, the memory requirement for rehearsal samples is included for a rehearsal-based Meta CL system, MetaCLNet.

Table D.2 shows the overall memory footprint for various baselines and our system to perform CL during the meta-test phase. First, the AIM variants (OML+AIM and ANML+AIM) require an enormous overall memory footprint of 135.2-1,051 MB and 608.2-1,562 MB, respectively, as their AIM module has a large number of parameters. This large required memory easily exceeds the RAM size of embedded devices such as Pi 3B+ (i.e., 1 GB) and barely fits on the device even with 1.7 GB of available memory space (e.g., Jetson Nano). On the other hand, other 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 continual learning. In contrast, MetaCLNet shows the impressive results that it only requires 3.40 MB for GSCv2, 15.45 MB for CIFAR-100, and 136.7 MB for MiniImageNet, demonstrating a very high compression rate of 178.7×, 70.8×, and 11.4× compared to ANML+AIM, respectively.

In summary, our results indicate that MetaCLNet has a very low memory footprint compared to existing works and enables performing CL on resource-constrained embedded devices.

D.4.4 End-to-end Latency & Energy Consumption

In this subsection, we examine run-time system efficiency regarding runtime latency and energy consumption of our system and the baselines during the deployment on two embedded devices such as Jetson Nano and Pi 3B+. Specifically, we measure the end-tn-end latency and overall energy consumption of the device to conduct CL over all the given classes during the meta-test phase on two embedded devices. To obtain the end-to-end latency, we include the time to load a pretrained model, the time to train the model continually over the given classes, and the time to compress and decompress the latent representations using our proposed compression method (i.e., sparse bitmap compression and PQ). Figure D.5 shows end-to-end latency and energy consumption results.

Latency. First, we measure the end-to-end latency of our system and the baselines on Jetson Nano to perform CL over all the given classes with 30 samples per class. As shown in Figures D.5a, D.5c, and D.5e, MetaCLNet enables a fast end-to-end latency (415 seconds for CIFAR-100, 1373 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 cannot run on Jetson Nano due to its excessive memory requirements. Furthermore, when MetaCLNet is compared to ANML which shares the same network architecture, MetaCLNet 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 MetaCLNet to perform the compression techniques like the sparse bitmap compression and PQ, the speed gains derived from using quantized neural weights and activations offset the overheads of compression techniques (see §D.4.5 for more details). After having demonstrated the efficiency of MetaCLNet on the Jetson Nano, we deployed our system on an even more resource-constrained device, Pi 3B+, with only 600-700 available memory space out of 1 GB of RAM. The end-to-end latency on Pi 3B+ largely stays similar to that on Jetson Nano as shown in Figure D.5.

Energy Consumption. We measure the energy consumption of the end-to-end CL over multiple classes of our system and the baselines on Jetson Nano and Pi 3B+ as shown in Figures D.5b, D.5d, and D.5f. We use Tegrastats on Jetson Nano to measure the power consumption with which we calculate the energy consumption by multiplying power consumption and the elapsed time for each end-to-end CL trial. Likewise to the latency results, compared to ANML+AIM, MetaCLNet 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). Moreover, compared to ANML, MetaCLNet 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+.

To summarize, our results demonstrate that MetaCLNet enables fast (low latency) and efficient (low energy consumption) CL on edge devices.

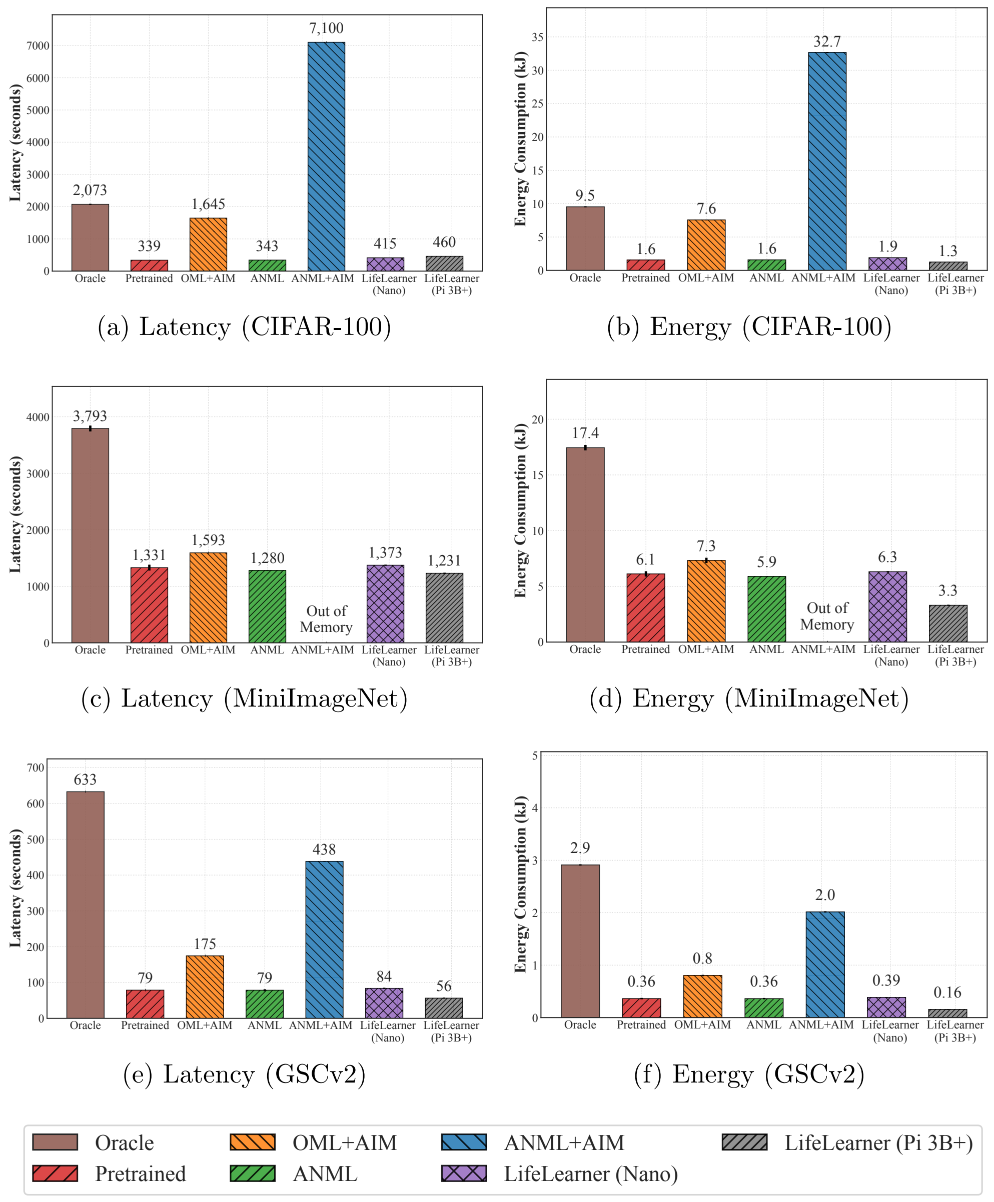


Figure D.5: The end-to-end latency and energy consumption of the baselines and MetaCLNet to perform CL over all the given classes during the meta-test phase on three datasets. All results are averaged over three runs with standard deviations.

D.4.5 Ablation Study

To investigate the role of each component of our CL system (MetaCLNet) in the performance and system efficiency, we perform an ablation study on the proposed components:

Table D.3: The comparison of the MetaCLNet and other variants of rehearsal-based Meta CL methods during the meta-test phase for the ablation study.

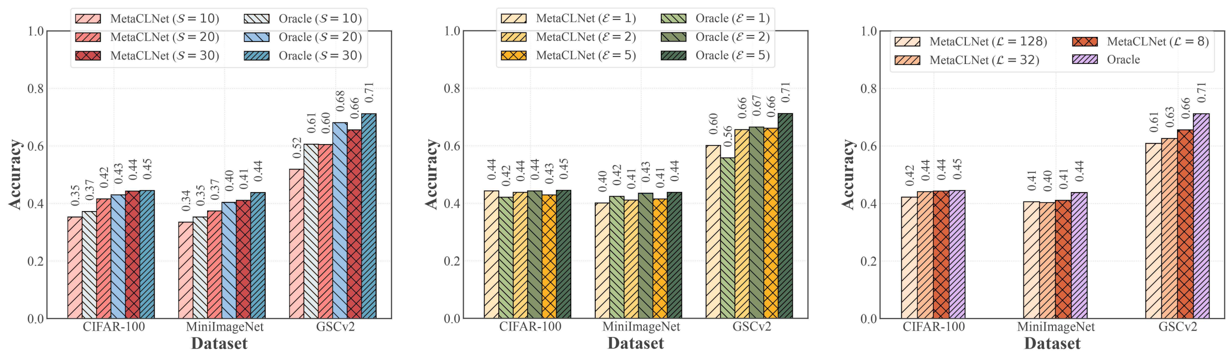
Dataset	**System**	**Accuracy**	**Memory**	**Latency**	**Energy**
CIFAR-100	ANML	0.272	39.69 MB	343.2s	1.58kJ
	Raw	0.392	110.5 MB	12,693s	58.39kJ
	Latent	0.452	53.9 MB	432.5s	1.99kJ
	Latent+Bit	0.452	41.2 MB	466.9s	2.15kJ
	Latent+PQ	0.448	41.8 MB	437.1s	2.01kJ
	Latent+Bit+PQ	0.455	40.4 MB	471.4s	2.17kJ
	MetaCLNet	**0.443**	**15.5 MB**	**414.7s**	**1.91kJ**
MiniImageNet	ANML	0.327	474.5 MB	1,280s	5.89kJ
	Raw	0.429	897.1 MB	206,234s	948.67kJ
	Latent	0.433	512.5 MB	1,492s	6.86kJ
	Latent+Bit	0.433	477.7 MB	1,551s	7.14kJ
	Latent+PQ	0.443	483.0 MB	1,501s	6.90kJ
	Latent+Bit+PQ	0.436	476.4 MB	1,560s	7.18kJ
	MetaCLNet	**0.411**	**136.7 MB**	**1,373s**	**6.32kJ**
GSCv2	ANML	0.429	10.2 MB	78.6s	0.36kJ
	Raw	0.135	50.8 MB	1,627s	3.74kJ
	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	0.708	10.3 MB	95.2s	0.44kJ
	MetaCLNet	**0.656**	**3.40 MB**	**83.8s**	**0.39kJ**

(1) rehearsal techniques, (2) sparse bitmap compression, (3) PQ, and (4) quantization. Based on these components, we formulate various rehearsal-based Meta CL systems that incrementally contain the proposed components one by one. The list of the CL systems that build on top of **ANML** is as follows.

Raw: The raw inputs of the given samples are stored and replayed when a deployed model learns new classes. Also, the neuromodulatory layers are frozen during deployment.

Latent: The latent representations of the given samples are saved and replayed later when a deployed model encounters new classes. The feature extractor (i.e., neuromodulatory and prediction networks) except for the classifier is frozen during deployment.

Latent+Bit: This CL system builds on top of Latent and applies the sparse bitmap



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

Figure D.6: The parameter analysis of MetaCLNet for all the datasets according to the three parameters.

compression to the latent representations. We include Latent+Bit to perform an ablation study with and without the sparse bitmap compression.

Latent+PQ: This CL system builds on top of Latent and applies the PQ to the latent representations. We include Latent+PQ to perform an ablation study with and without PQ.

Latent+Bit+PQ: This CL system builds on top of Latent+Bit and applies PQ on the compressed latent representations that are sifted out through the sparse bitmap compression. We include Latent+Bit+PQ to evaluate the combined effect of the sparse bitmap compression and PQ.

Latent+Bit+PQ+Int8 (MetaCLNet): This is our final CL system that contains all the components proposed in this work for the system optimization. MetaCLNetis based on Latent+Bit+PQ and further quantizes the neural weights and activations of the feature extractor into 8-bit integers, allowing us to evaluate the impact of 8-bit quantization.

Table D.3 summarizes the ablation study results regarding the accuracy, memory footprint, end-to-end latency, and energy consumption during the meta-test phase of on-device deployment. First of all, we find that the combination of rehearsal techniques with ANML drastically improves the accuracy. For example, Latent increases the accuracy of ANML by 10.6-28.4% across all the datasets with some overheads on system resources (memory, latency, and energy). Although Raw improves accuracy compared to ANML for the CIFAR-100 and MiniImageNet datasets, Raw fails to generalize to another data modality, showing very low accuracy for GSCv2. In addition, Raw requires the most amount of memory, latency, and energy among all the CL systems in the ablation study. Latent causes a modest amount of additional memory, latency, and energy compared to ANML.

These results demonstrate the effectiveness of incorporating the rehearsal strategy to Meta CL and show that latent replays can be a better design choice over native rehearsal, and hence we adopt to use of the latent replay in our system, MetaCLNet. In addition, the results of various CL systems such as Latent+Bit, Latent+PQ, and Latent+Bit+PQ show that our proposed compression techniques for latent representations do not sacrifice the accuracy of the CL systems and reduce the overall memory footprint compared to Latent. They only incur small resource overheads in terms of latency and energy. Then, our final CL system, MetaCLNet, demonstrates the 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 only minimal overheads of end-to-end latency and energy compared to ANML (i.e., costs additional 56.6s and 0.3kJ on average, respectively) but still shows lower latency and energy consumption compared to Latent (saves 47.9s and 0.2kJ on average, respectively).

Overall, the ablation study reveals that each proposed component is equally important in making our final CL system more accurate and efficient (lower memory, latency, and energy consumption).

D.4.6 Parameter Analysis

Finally, we study the impact of the various hyper-parameters that could affect the performance of our system as shown in Figure D.6. Hence, in this subsection, we choose three parameters to further analysis, namely, (1) the number of the given samples per class, (2) the number of replay epochs, and (3) the sub-vector length of the PQ codebook.

First, Figure D.6a shows the accuracy of MetaCLNet according to the number of the given samples per class ranging from 10 to 30. MetaCLNet shows the lowest accuracy when only 10 samples per class are given to conduct training. Apparently, the more samples are given for training, the higher the accuracy, which holds for both MetaCLNet and Oracle. Interestingly, the accuracy differences between MetaCLNet and Oracle are small (e.g., 1-2% for CIFAR-100, 1-3% for MiniImageNet, and 5-9% for GSCv2), demonstrating that MetaCLNet still achieves the similar accuracy of Oracle. With 30 given samples, the accuracy difference is minimal: 2.8% on average, and ranging from 1 to 5%.

Secondly, Figure D.6b shows the accuracy of MetaCLNet according to the number of the replay epochs ranging from 1 to 5. The results of MetaCLNet show that the accuracy 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 latency than our system (see §D.4.4). This result benefits us since replaying the rehearsal samples over one or two epochs is enough for MetaCLNet to reach the converging accuracy, which helps decrease the system overheads such as training latency and energy consumption.

Third, Figure D.6c shows the accuracy of MetaCLNet according to the sub-vector length of the PQ codebook (the number of values per index, i.e., compression ratio) ranging from 8 to 128. For CIFAR-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., the smaller number of values are compressed per index), the higher the accuracy during the meta-test phase.

In summary, these results show that with only 10-30 samples per class, MetaCLNet can still achieve similar CL performance to Oracle, exhibit rapid convergence with small replay epochs (at most two), and accomplish a high compression rate for latent representations.

D.5 Related Work

In this section, we review relevant prior studies regarding (1) continual learning, (2) compression techniques for DNN inference and training, and (3) mobile and embedded sensing applications.

Continual Learning. Continual Learning (CL) efforts have also been referred to as lifelong learning [3], incremental learning [13], and sequential learning [27]. Various approaches attempt to solve the typical catastrophic forgetting problem of CL [27, 39, 159]. The first group of approaches includes regularization-based methods [29, 36, 37]: these add a regularization term to the loss function to minimize changes to important weights of a model for previously learned classes to prevent forgetting. The second group of approaches includes the dynamic architecture-based methods [149, 30, 150] that dynamically expand and freeze DNN architectures to incorporate new classes and prevent forgetting. The last group of approaches among conventional CL includes rehearsal-based methods [13, 31, 14, 65]. These prevent forgetting by replaying the saved rehearsal samples from earlier classes.

In recent years, a new group of approaches, as mentioned in the previous sections, Meta CL [151, 152, 153], has been proposed to resolve the limitations of conventional CL methods. However, Meta CL also suffers from low CL performance and high resource overheads, as described in Section D.2. In this work, we have proposed a novel rehearsal-based Meta CL that achieves high CL performance with low resource overheads instead. As mentioned, MetaCLNet, can be deployed on resource-constrained IoT devices where it is not feasible to run SOTA Meta CL due to its excessive memory requirements.

Compression Techniques for DNN Inference and Training. Researchers have focused on enabling accurate yet efficient DNN inference by compressing the model [162]. As a result, many of the hand-crafted network architecture were proposed such as SqueezeNets [134], ShuffleNets [108], and MobileNets [107]. However, to make a fair comparison with prior Meta CL methods, we employ the network architecture used in prior works [151, 152, 153] instead of searching for more efficient network architectures.

Besides, many works on pruning and quantization utilize the inherent redundancy in weights and activations of DNNs [79, 137, 9, 111, 142]. In this work, we employ the widely used quantization technique based on 8-bit quantization [9, 111] for optimizing our CL system.

Another thread of research is focused on reducing the overall system resources required for DNN training [163, 161]. These methods can be generally categorized into thee directions: optimization of the model layer growth, the activation layer sparsity, and last layer fine-tuning. For example, researchers control the layerwise growth of the model structure to enable efficient DNN model execution on mobile phones [164]. Second, other methods focus on optimizing network activation sparsity to avoid redundant model weights during DNN training [154, 165]. Third, another work proposed fine-tuning the last layer to enable efficient DNN training in the wild with new datasets [166]. Moreover, as another technique to compress a vector (e.g., features, weights, and activations of DNNs), product quantization (PQ) [11, 101, 102, 14] has been widely adopted in the database and ML areas. Some works use PQ to compress weights and activations of DNNs and rehearsal samples to perform CL. Inspired by those works, we propose MetaCLNet by incorporating various compression techniques such as the sparse bitmap compression [154] and PQ [11] to further compress the latent representations in our novel rehearsal-based Meta CL for the first time.

Mobile and Embedded Sensing Applications. With the rise of deep learning, increasing number of mobile and embedded systems adopt DNN models. It is also because DNN models have demonstrated state-of-the-art performances in many real-world mobile and embedded sensing applications, namely, computer vision [93, 167], audio sensing [45], and many other applications [58, 63]. For instance, image classification is extensively studied, and its datasets [96, 155, 113, 114] are widely used in many fields including transfer learning [168], meta-learning [156], self-supervised learning [169], and federated learning [170]. Moreover, many researchers have investigated using DNN for audio sensing tasks such as keyword spotting [90], emotion recognition [6], speaker identification [50], and environmental sound classification [84]. In this work, we adopt image classification and keyword spotting as our sensing applications for a case study because image classification is widely used in the prior SOTA Meta CL work [153], and keyword spotting is one of the representative audio sensing applications on edge devices [171].

D.6 Discussion

Impact on Continual Learning. MetaCLNet is the first framework that allows CL on resource-constrained edge devices and combines meta-learning with rehearsal-based strategies to learn new classes incrementally using a small number of labeled data while maintaining high performance and preventing forgetting, which is quintessential for any

CL system. Previous efforts in Meta CL either suffer from low accuracy [152, 151] or are not suitable for deployment on edge devices due to their excessive resource overheads [153]. Thus, we envision that MetaCLNet could make CL a practical reality on embedded devices. Such CL systems will allow DNN models 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 that can help us move one inch closer to the idea of general artificial intelligence. We leave the wider deployment and performance evaluation of MetaCLNet on other more resource-constrained embedded platforms as our future work. In this context, optimizing MetaCLNet to utilize much stricter weight quantization such as 1, 2, or 4 bits will be worth pursuing.

Generalizability of MetaCLNet. We have demonstrated that MetaCLNet successfully works on three different datasets operating on two different modalities: image and audio, showing the generalizability of our framework. The evaluation of other datasets and potentially other modalities, including IMU data to further test the applicability of MetaCLNet for learning continually for other real-world applications, is left as future work.

D.7 Conclusions

We have proposed a novel rehearsal-based Meta CL system, MetaCLNet, that achieves very high accuracy by combining the realms of Meta CL and rehearsal-based learning. It also enables highly efficient end-to-end CL for edge devices by utilizing various compression techniques on rehearsal samples and neural weights/activations to reduce the system overheads. As a result, MetaCLNet outperforms the prior SOTA Meta CL method 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.

Bibliography

[1] 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.

[2] 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.

[3] 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, May 2019.

[4] 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.

[5] 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.

[6] 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 *Proc. UbiComp*, pages 281–290, September 2010.

[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] 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.

[9] 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. volume abs/1712.05877, 2017.

[10] 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.

[11] 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.

[12] 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.

[13] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proc. CVPR*, pages 2001–2010, 2017.

[14] 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*, Lecture Notes in Computer Science, pages 466–483, Cham, 2020. Springer International Publishing.

[15] 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.

[16] 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.

[17] 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.

[18] Young D. Kwon. *Efficient Meta Continual Learning on the Edge.* 2021.

[19] 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.

[20] 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.

[21] Young D. Kwon. *On-device Training at the Extreme Edge.* 2023.

[22] 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.

[23] 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.

[24] 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 *Proc. UbiComp*, pages 351–360, 2012.

[25] 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.

[26] 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.

[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] Z. Li and D. Hoiem. Learning without Forgetting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(12):2935–2947, December 2018.

[29] 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.

[30] Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong Learning with Dynamically Expandable Networks. February 2018.

[31] David Lopez-Paz and Marc\textquotesingle Aurelio Ranzato. Gradient Episodic Memory for Continual Learning. In *Proc. NIPS*, pages 6467–6476. 2017.

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

[33] Monika Schak and Alexander Gepperth. A study on catastrophic forgetting in deep LSTM networks. page 14.

[34] 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.

[35] Sang-Woo Lee, Jin-Hwa Kim, Jaehyun Jun, Jung-Woo Ha, and Byoung-Tak Zhang. Overcoming Catastrophic Forgetting by Incremental Moment Matching. 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 4652–4662. 2017.

[36] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual Learning Through Synaptic Intelligence. In *Proc. ICML*, pages 3987–3995, 2017.

[37] 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 *International Conference on Machine Learning*, pages 4535–4544, 2018.

[38] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] Nicholas D. Lane, Petko Georgiev, and Lorena Qendro. DeepEar: Robust Smartphone Audio Sensing in Unconstrained Acoustic Environments Using Deep Learning. In *Proc. UbiComp*, pages 283–294, 2015.

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

[47] 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.

[48] 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.

[49] 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.

[50] Sourav Bhattacharya and Nicholas D Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 176–189, 2016.

[51] 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. IMWUT*, 1(3):50:1–50:19, September 2017.

[52] 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.

[53] Gido M. van de Ven and Andreas S. Tolias. Three scenarios for continual learning. *arXiv:1904.07734 [cs, stat]*, April 2019.

[54] 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.

[55] 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.

[56] 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.

[57] Gary King and Langche Zeng. Logistic Regression in Rare Events Data. *Political Analysis*, 9(2):137–163, 2001.

[58] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kj\a ergaard, 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, pages 127–140, 2015.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] Petko Georgiev, Nicholas D Lane, Kiran K Rachuri, and Cecilia Mascolo. Dsp.ear: Leveraging co-processor support for continuous audio sensing on smartphones. In *Proc. SenSys*, pages 295–309, 2014.

[67] 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.

[68] Arslan Chaudhry, Puneet K. Dokania, Thalaiyasingam Ajanthan, and Philip H. S. Torr. Riemannian Walk for Incremental Learning: Understanding Forgetting and Intransigence. pages 532–547, 2018.

[69] Rahaf Aljundi, Punarjay Chakravarty, and Tinne Tuytelaars. Expert Gate: Lifelong Learning With a Network of Experts. pages 3366–3375, 2017.

[70] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul,

Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

[71] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems*, pages 7675–7684, 2018.

[72] Justin Salamon, Christopher Jacoby, and Juan Pablo Bello. A Dataset and Taxonomy for Urban Sound Research. In *Proc. ACM MM*, pages 1041–1044, November 2014.

[73] 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.

[74] 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.

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

[76] 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.

[77] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *Proc. IPSN*, pages 1–12, April 2016.

[78] 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.

[79] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]*, February 2016.

[80] 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. *Proc. MLSys*, 2:1–13, March 2020.

[81] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. On-Device Neural Net Inference with Mobile GPUs. *arXiv:1907.01989 [cs, stat]*, July 2019.

[82] PyTorch.

[83] 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.

[84] 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.

[85] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, January 2017.

[86] 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.

[87] 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.

[88] 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.

[89] Igor Fedorov, Ryan P. Adams, Matthew Mattina, and Paul Whatmough. SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers. pages 4977–4989, 2019.

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

[91] Wearable Device for Blind People Could be a Life Changer | NVIDIA Blog, October 2016.

[92] 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.

[93] 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.

[94] 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.

[95] 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. pages 184–199, 2018.

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

[97] 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.

[98] 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.

[99] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing Deep Convolutional Networks using Vector Quantization. *arXiv:1412.6115 [cs]*, December 2014.

[100] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized Convolutional Neural Networks for Mobile Devices. pages 4820–4828, 2016.

[101] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. And the Bit Goes Down: Revisiting the Quantization of Neural Networks. September 2019.

[102] Pierre Stock, Angela Fan, Benjamin Graham, Edouard Grave, Rémi Gribonval, Herve Jegou, and Armand Joulin. Training with Quantization Noise for Extreme Model Compression. September 2020.

[103] 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. pages 15699–15708, 2021.

[104] R. Gray. Vector quantization. *IEEE ASSP Magazine*, 1(2):4–29, April 1984.

[105] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[106] 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.

[107] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, June 2018.

[108] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. pages 116–131, 2018.

[109] 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.

[110] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. MCUNet: Tiny Deep Learning on IoT Devices. *arXiv:2007.10319 [cs]*, July 2020.

[111] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv:1806.08342 [cs, stat]*, June 2018.

[112] 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.

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

[114] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The german traffic sign recognition benchmark: a multi-class classification competition. In *The 2011 international joint conference on neural networks*, pages 1453–1460. IEEE, 2011.

[115] Pete Warden. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv:1804.03209 [cs]*, April 2018.

[116] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[117] 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.

[118] 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.

[119] Rich Caruana. Multitask Learning. *Machine Learning*, 28(1):41–75, July 1997.

[120] 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.

[121] 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.

[122] 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.

[123] Andrew M McDonald, Massimiliano Pontil, and Dimitris Stamos. Spectral k-support norm regularization. In *NIPS*, pages 3644–3652, 2014.

[124] Lei Han and Yu Zhang. Learning multi-level task groups in multi-task learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[125] Zhuoliang Kang, Kristen Grauman, and Fei Sha. Learning with whom to share in multi-task feature learning. In *ICML*, 2011.

[126] Giwoong Lee, Eunho Yang, and Sung Hwang. Asymmetric multi-task learning based on task relatedness and loss. In *International conference on machine learning*, pages 230–238. PMLR, 2016.

[127] Mingsheng Long, Zhangjie Cao, Jianmin Wang, and Philip S Yu. Learning multiple tasks with multilinear relationship networks. *arXiv preprint arXiv:1506.02117*, 2015.

[128] 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.

[129] 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.

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

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

[132] 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.

[133] 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.

[134] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, and Kurt Keutzer. SqueezeNext: Hardware-Aware Neural Network Design. pages 1638–1647, 2018.

[135] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. pages 5687–5695, 2017.

[136] 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.

[137] 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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):4876–4883, April 2020.

[138] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning Filters for Efficient ConvNets. November 2016.

[139] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained Ternary Quantization. November 2016.

[140] Fengfu Li, Bo Zhang, and Bin Liu. Ternary Weight Networks. *arXiv:1605.04711 [cs]*, November 2016.

[141] 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.

[142] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, Lecture Notes in Computer Science, pages 525–542, Cham, 2016.

[143] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D. Lane, and Yarin Gal. An Empirical study of Binary Neural Networks' Optimisation. September 2018.

[144] 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. pages 315–332, 2018.

[145] 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.

[146] 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.

[147] Tiezheng Ge, Kaiming He, and Jian Sun. Product Sparse Coding. pages 939–946, 2014.

[148] Hessam Bagherinezhad, Mohammad Rastegari, and Ali Farhadi. LCNN: Lookup-Based Convolutional Neural Network. pages 7120–7129, 2017.

[149] 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.

[150] 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.

[151] Khurram Javed and Martha White. Meta-Learning Representations for Continual Learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 1820–1830. Curran Associates, Inc., 2019.

[152] Shawn Beaulieu, Lapo Frati, Thomas Miconi, Joel Lehman, Kenneth O. Stanley, Jeff Clune, and Nick Cheney. Learning to Continually Learn. *arXiv:2002.09571 [cs, stat]*, March 2020.

[153] 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.

[154] 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)*, pages 14–25, 2021.

[155] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, koray kavukcuoglu, and Daan Wierstra. Matching Networks for One Shot Learning. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3630–3638. 2016.

[156] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 1126–1135, Sydney, NSW, Australia, August 2017. JMLR.org.

[157] Maruan Al-Shedivat, Trapit Bansal, Yura Burda, Ilya Sutskever, Igor Mordatch, and Pieter Abbeel. Continuous adaptation via meta-learning in nonstationary and competitive environments. In *International Conference on Learning Representations*, 2018.

[158] Anusha Nagabandi, Chelsea Finn, and Sergey Levine. Deep online learning via meta-learning: Continual adaptation for model-based RL. In *International Conference on Learning Representations*, 2019.

[159] 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*, 44(7):3366–3385, 2022.

[160] Lorenzo Pellegrini, Gabriele Graffieti, Vincenzo Lomonaco, and Davide Maltoni. Latent replay for real-time continual learning. *arXiv preprint arXiv:1912.01100*, 2019.

[161] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-Memory Neural Network Training: A Technical Report. *arXiv:1904.10631 [cs, stat]*, April 2019.

[162] 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.

[163] Han Cai, Ji Lin, Yujun Lin, Zhijian Liu, Haotian Tang, Hanrui Wang, Ligeng Zhu, and Song Han. Enable Deep Learning on Mobile Devices: Methods, Systems, and

Applications. *ACM Transactions on Design Automation of Electronic Systems*, 27(3):20:1–20:50, March 2022.

[164] Yu Zhang, Tao Gu, and Xi Zhang. MDLdroidLite: a release-and-inhibit control approach to resource-efficient deep neural networks on mobile devices. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, pages 463–475, New York, NY, USA, November 2020. Association for Computing Machinery.

[165] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce memory, not parameters for efficient on-device learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 11285–11297. Curran Associates, Inc., 2020.

[166] Seulki Lee and Shahriar Nirjon. Learning in the Wild: When, How, and What to Learn for On-Device Dataset Adaptation. In *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, AIChallengeIoT '20, pages 34–40, New York, NY, USA, November 2020. Association for Computing Machinery.

[167] 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 '20, pages 476–488, New York, NY, USA, November 2020. Association for Computing Machinery.

[168] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A Survey on Deep Transfer Learning. In Věra Kůrková, Yannis Manolopoulos, Barbara Hammer, Lazaros Iliadis, and Ilias Maglogiannis, editors, *Artificial Neural Networks and Machine Learning – ICANN 2018*, Lecture Notes in Computer Science, pages 270–279, Cham, 2018. Springer International Publishing.

[169] Longlong Jing and Yingli Tian. Self-supervised Visual Feature Learning with Deep Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2020. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[170] Ang Li, Jingwei Sun, Xiao Zeng, Mi Zhang, Hai Li, and Yiran Chen. FedMask: Joint Computation and Communication-Efficient Personalized Federated Learning via Heterogeneous Masking. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, pages 42–55, New York, NY, USA, November 2021. Association for Computing Machinery.

[171] 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.