

On-device Training: A First Overview on Existing Systems

SHUAI ZHU, RISE Research Institutes of Sweden & Uppsala University, Sweden

THIEMO VOIGT, Uppsala University & RISE Research Institutes of Sweden, Sweden

FATEMEH RAHIMIAN, RISE Research Institutes of Sweden, Sweden

JEONGGIL KO, Yonsei University & POSTECH, South Korea

The recent breakthroughs in machine learning (ML) and deep learning (DL) have catalyzed the design and development of various intelligent systems over wide application domains. While most existing machine learning models require large memory and computing power, efforts have been made to deploy some models on resource-constrained devices as well. A majority of the early application systems focused on exploiting the inference capabilities of ML and DL models, where data captured from different mobile and embedded sensing components are processed through these models for application goals such as classification and segmentation. More recently, the concept of exploiting the mobile and embedded computing resources for ML/DL model training has gained attention, as such capabilities allow (i) the training of models via local data without the need to share data over wireless links, thus enabling privacy-preserving computation by design, (ii) model personalization and environment adaptation, and (iii) deployment of accurate models in remote and hardly accessible locations without stable internet connectivity. This work summarizes and analyzes state-of-the-art systems research that allows such on-device model training capabilities and provides a survey of on-device training from a systems perspective.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Computing methodologies** → **Machine learning**; **Learning paradigms**; • **Hardware** → **Sensor devices and platforms**.

Additional Key Words and Phrases: Machine Learning, IoT devices, On-device Training

ACM Reference Format:

Shuai Zhu, Thiemo Voigt, Fatemeh Rahimian, and JeongGil Ko. 2024. On-device Training: A First Overview on Existing Systems. 1, 1 (September 2024), 38 pages. <https://doi.org/XXXXXXXX.XXXXXXX>

1 Introduction

Machine learning (ML) has been extensively studied in recent years and has matured enough to be practically integrated into real-world computing systems. Data-driven ML-based system design has catalyzed the development of countless applications in many application domains, including those targeted by Internet of Things (IoT) systems. Concurrently, with the fast-growing interest and advancements in technology that support IoT systems, the number of IoT devices deployed in our everyday environments is significantly increasing. GSMA Intelligence¹ forecasts that the number of

¹<https://www.gsmainelligence.com/>

Authors' Contact Information: Shuai Zhu, RISE Research Institutes of Sweden & Uppsala University, Stockholm, Sweden, shuai.zhu@ri.se; Thiemo Voigt, Uppsala University & RISE Research Institutes of Sweden, Uppsala & Stockholm, Sweden, thiemo.voigt@angstrom.uu.se; Fatemeh Rahimian, RISE Research Institutes of Sweden, Stockholm, Sweden, fatemeh.rahimian@ri.se; JeongGil Ko, Yonsei University & POSTECH, Seoul & Pohang, South Korea, jeonggil.ko@yonsei.ac.kr (Corresponding Author).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

Manuscript submitted to ACM

IoT devices will increase to 25 billion by 2025. IoT devices with various sensing components are being ubiquitously adopted, taking the role of capturing data for various application scenarios, including healthcare, surveillance, and environmental monitoring. Data generated from these devices can provide opportunities for applying data-driven ML methods to accomplish their application goals. Advances in ML, including Deep Learning (DL), have resulted in tremendous improvements in solving various types of problems in computer vision, natural language processing, machine translation, etc., and across different application domains such as biology and healthcare, automotive industries, smart cities, and many more. To fully leverage the advances in both IoT and ML, on-device ML integrates ML algorithms on IoT devices and is becoming an active area of research.

Largely, on-device ML consists of on-device inference and training operations. On-device inference refers to deploying pre-trained ML models on devices and locally performing inference operations such as classification or regression. Researchers have presented extensive works using such an ‘inference-embedded’ system architecture [23, 56, 66, 120, 126–129, 131, 133]. On the other hand, on-device training takes a step further and targets to *train* the models locally. Note that, typically, due to computational power limitations, cloud-based (or server-based training) is the mainstream approach, where local data is shared with the server, and the training operations execute at remote platforms. Later, updated models can be re-distributed to local IoT devices to perform local inference operations. However, since holding the capability to locally train an ML model can preserve the precious network bandwidth and limited battery budgets and at the same time contain the raw data locally to preserve privacy, researchers have recently started to propose schemes to train ML models on-device, despite their computational resource limitations [6, 27, 74, 94, 97, 100, 121, 125]. Still, these works are in their early stages, and we believe that a comprehensive view of what is done and what is left, can catalyze further research ahead. Unlike previous work that analyzes the research space from an algorithmic perspective [17], we take a systematic view and present a survey on systems and frameworks for supporting on-device training. On-device training comes with the following benefits:

- From the device perspective, the main advantages are that on-device training can take place even in the absence of an Internet connection. Secondly, there is no need to upload data to the cloud and/or download an updated model back, and that in itself saves bandwidth and reduces latency and energy. The latter is important for low-power IoT devices, where communication is typically far more expensive than computing. It is also important when dealing with time-critical applications that should react quickly without communicating with a server or over a network.
- From a data perspective, training on local devices is privacy-preserving by design. Privacy is a critical issue in many real-world use cases and has slowed down the deployment of AI models in such cases [22]. Enabling on-device training removes this obstacle and enables the utilization of ML models in reality.
- From a modeling perspective, devices can become smarter as they can cope with model drift problems [116] and retrain the deployed pre-trained models in order to adapt to the environment and even the end-user. For example, a medical device can over time learn to provide personalized predictions or services that fit the specific conditions of a certain patient.

1.1 Challenges of On-device Training

On-device training entails the following three main technical challenges.

- **Mismatch between hardware resources availability and demand:** IoT devices typically own limited hardware resources, for example, the memory capacity is in the order of kilobytes or megabytes. However, ML model

training requires considerable hardware, including computing and memory resources. Concretely, model training could be described as an optimization problem. The optimization goal is to minimize the loss, which measures the difference between the output of the model and the ground truth. During the training process, ML methods keep tuning model parameters by following the gradient direction. Gradient-based training normally applies backpropagation algorithms to compute the gradient. The backpropagation algorithm stores a large amount of intermediate loss and output of hidden layers, which could consume memory on the order of tens of gigabytes or more. This contradiction makes on-device training challenging.

- **High heterogeneity of IoT devices:** IoT devices are diverse and heterogeneous. Specifically, apart from the heterogeneity in vendors, IoT devices cover a wide spectrum in terms of their capability: from severely constrained microcontrollers, such as Nordic Semiconductor nRF9160², to single-board computers with more capable hardware, such as Raspberry Pi 4³. Therefore, it is challenging to propose a single and generally applicable solution for all devices.
- **Limited existing work:** Most existing model training optimization algorithms, such as batch normalization (BN) [44] and Adam [57], either target to achieve higher accuracy or converge faster. These works often overlook the resource constraints of edge devices, such as limited memory and battery power.

1.2 Scope of this Survey

This survey summarizes current state-of-the-art (SOTA) systems research on on-device training. Such systems aim to enable neural network training on resource-constraint devices, ranging from modern mobile platforms to microcontrollers. Our survey also includes modern smartphones but does not include the NVIDIA Jetson platform [12]. Some modern smartphones are equipped with dedicated accelerators, such as GPUs and NPUs. However, smartphones are user-centric and integrate multiple functions, such as communication, entertainment, and photography, which share hardware resources. Excessive resource consumption by one application can lead to degradation of the user experience. In addition, smartphones are powered by built-in batteries, so applications also need to consider power consumption. Systems designed for smartphones must contend with various resource constraints, although smartphones appear to involve abundant hardware resources. Therefore, this survey includes systems designed for smartphones. On the other hand, the NVIDIA Jetson platform is a powerful tool for specialized artificial intelligence and robotics applications. Hence, this survey will not provide a detailed analysis of systems that are only evaluated on NVIDIA Jetson platforms. In addition, this work focuses on single-device training and does not include the body of work in collaborative model training (e.g., federated learning).

The articles we survey are published in renowned conferences on relevant topics, including mobile computing, wireless and mobile networking, and ML. Specifically, this survey focuses on recently published papers (2019-2023) from MobiCom, SenSys, IPSN, MobiSys, and EWSN, as well as AAAI, ICLR, ICML, NeurIPS, and IJCAI. Apart from these conferences, we also use Google Scholar and Microsoft Research to identify other relevant work. Keywords used to obtain these papers are on-device/edge/device + learning/training/adaptation/update. Overall, we choose and direct our focus to analyzing eleven representative systems.

²<https://www.nordicsemi.com/products/nrf9160>

³<https://www.raspberrypi.com/products/raspberry-pi-4-model-b/>

1.3 Outline

The rest of the paper is organized as follows. Section 2 provides an overview of current on-device training systems and the target devices of these systems. We present an analysis from the Machine Learning perspective in Section 3 and Section 4 presents optimization techniques to enable on-device model training. Section 5 discusses the results reported from current SOTA systems. Before presenting the conclusions, Section 6 elaborates on the methods and ideas presented in the current SOTA system works for on-device training. Finally, Section 7 discusses potential future directions and concludes this article.

2 Overview of Existing Systems

We survey the following systems for on-device training: POET [94], Mandheling [125], MiniLearn [97], Melon [121], Sage [27], TinyTL [6], TTE [74], ElasticTrainer [41], MDLdroidLite [134], LifeLearner [65], and zTT [55]. In summary, existing systems operate within four paradigms: one-stage, two-stage, continual learning, and reinforcement learning paradigms. Firstly, the one-stage paradigm systems apply various optimization techniques, such as pruning [90] and quantization [46], to perform model training. Secondly, unlike the one-stage paradigm, the two-stage paradigm has an additional preparation stage, where systems prepare the computing graph and generate the training plan before performing model training. The plan schedules when and how to use specific techniques during training. Thirdly, continual learning-based systems enable Deep Neural Network (DNN) models to learn new tasks throughout their lifetime. MDLdroidLite and LifeLearner are two continual learning-based systems that apply continuous growth and memory replay techniques to facilitate on-device training, respectively. Details are introduced in Section 2.6. Another system, Miro [80], also applies continual learning to enable on-device training. However, the authors only evaluate the feasibility of the system on NVIDIA Jetson platforms (NVIDIA Jetson TX2 and NVIDIA Jetson Xavier NX). Therefore, this survey only provides a brief summary of Miro. Fourthly, zTT employs reinforcement learning to enable on-device training. The system focuses on learning-based Dynamic Voltage and Frequency Scaling (DVFS) without thermal throttling for mobile devices to dynamically adjust CPU and GPU voltage-frequency levels during runtime, which effectively manages heat generation and power consumption. This approach aims to strike a balance between energy efficiency and performance.

This section begins by providing an overview of relevant review papers on on-device training. It then delves into the distinction between on-device training and general acceleration techniques for DNN training. Furthermore, it provides an overview of the devices targeted by existing systems. Finally, the section outlines the key design choices adopted by these systems.

2.1 Relevant Reviews on On-Device Training

This paper provides a comprehensive overview of on-device training from a *system perspective*, focusing on enabling model training on single resource-constrained devices. While Zhao et al.[135] offer a summary and comparison of various DL applications on mobile devices, emphasizing on-device inference for tasks like computer vision, speech/speaker recognition, and transportation mode detection, they only touch briefly on training methodologies, particularly federated learning. Similarly, Saha et al.[106] concentrate on techniques for ML inference on microcontroller-class devices, discussing features like model pruning, quantization, and lightweight neural network architecture design. Their discussions provide a high-level overview of training methods, including last-layer transfer learning, quantized continual learning, and training specialized operators (e.g., TinyTL [6]). In contrast, our work provides a more detailed summary

and analysis of on-device training techniques, offering insights into how these methods enable efficient model training directly on resource-constrained devices.

In addition, Dhar et al. [17] provides a comprehensive review of on-device training from an algorithms and theory perspective. Unlike the focus on system-level considerations in our paper, Dhar et al. delve into the algorithmic and theoretical aspects of on-device training. They categorize SOTA designs into five approaches, including lightweight ML algorithms, model complexity reduction, optimization routine modifications (e.g., quantization), data compression, and new protocols for data observation. Furthermore, the authors offer insights into theoretical considerations, urging researchers to analyze resource constraints, assess applicability, focus on algorithm development, and extend existing theories. To the best of our knowledge, this represents the first review of on-device training with a focus on algorithms and theory.

2.2 Comparison with General Speed-up Techniques for DNN Training

The efficiency of DNN training is a critical area of research, given the extensive hardware resources, both memory and computation resources, required for such tasks. This section discusses the prevalent techniques for high-efficiency DNN training and the distinctions between them and on-device training. Specifically, the efficiency metrics for DL consist of two representative types: computation-related and memory-related metrics. Hence, general speeding-up techniques for training aim to optimize either one or both of these. Concretely, these approaches aim to reduce training time and resource usage while maintaining (or even improving) model performance. In contrast, while on-device training systems generally share similar rationale and goals, they face an additional layer of constraints, including energy resources, memory, and computation resources constraints, which pose additional challenges for on-device training.

2.2.1 Computational Efficiency Optimization Approaches. Optimizing computational efficiency for DL model training is crucial for speeding up experiments, reducing costs, and enabling the training of larger models. Those approaches generally optimize computation in three aspects: hardware, data, and algorithms, which are the three pillars of DL [28].

Hardware acceleration: Hardware acceleration techniques aim to design specialized hardware units to speed-up the computation time for both training and inference operations. The most common hardware accelerators are graphics processing units (GPUs) [86] and tensor processing units (TPUs) [51]. For example, the NVIDIA H100 Tensor Core GPU⁴ can accelerate exascale workloads. However, hardware accelerators for edge devices are significantly less powerful than accelerators in the server/cloud, and many processing components, such as microcontrollers, are not equipped with and do not support dedicated hardware accelerators. In addition, platforms with limited energy resources, such as those that are battery-powered, cannot afford to perform highly parallel computations with accelerators. In contrast, on-device training systems enable highly resource-efficient on-device training by leveraging other types of assisting hardware, such as Digital Signal Processors (DSPs) [125]. Overall, while both general speed-up techniques for DNN training and on-device training leverage assisting hardware, they focus on different performance aspects. General techniques mainly concern the training speed, while on-device training systems aim to *complete* the training with constrained computing resources.

Data optimization and management: Data optimization and management for DNN training aims to optimize the data pipeline and task processing between memory units and computing units to enhance computing performance and efficiency by exploiting techniques such as data preprocessing, caching strategies, data pipeline optimization for I/O

⁴<https://www.nvidia.com/en-us/data-center/h100/>

bottleneck minimization, and datasets organization to facilitate easy access and manipulation. In this regard, on-device training and general speed-up techniques share similar goals.

Algorithm efficiency: Algorithm efficiency in DNN model training refers to how effectively a given algorithm uses computational resources, such as GPUs, to achieve training objectives within a reasonable amount of time. One typical general speed-up technique is to leverage parallelism and distributed strategies for training. For example, SALIENT [53] achieves a speed-up of 3× over a standard PyTorch-Geometric [24] implementation with a single GPU and a further 8× parallel speed-up with 16 GPUs. The goal is to achieve high accuracy and consume less computational resources. For instance, Ekya [4] is a continuous learning system for video analytics models on edge servers. The system solves the challenge of jointly supporting inference and retraining tasks, which trades-off the accuracy of the retrained model with the inference accuracy. Compared to the baseline [132], Ekya achieves 29% higher accuracy, and to reach the same accuracy as Ekya, the baseline requires 4× more GPU resources. Unfortunately, on-device training systems do not have the luxury of utilizing such powerful hardware resources. Overall, general speed-up techniques aim to accelerate the training process and achieve start-of-the-art performance, while the goal of many on-device training schemes is to enable training and achieve reasonably good performance.

2.2.2 Memory Reduction Approaches. As DL research has evolved towards employing deeper models and larger mini-batches to pursue higher accuracy, the memory blowup issue in DNN training has become a significant research problem. Specifically, the memory usage of training operations can be 5× to 100× higher compared to inference depending on the input and batch sizes [27]. Even for server-scale GPUs, the increasing memory demand in model training becomes challenging to accommodate. To bridge the gap between memory demand and supply, researchers have made various efforts to reduce model training memory overhead. Specifically, memory reduction approaches can be broadly classified in five categories [27]: (1) gradient accumulation, (2) gradient checkpointing, (3) reducing activation size, (4) reducing activation count, and (5) memory virtualization. Concretely, the first four categories pertain to software- and algorithm-level memory usage optimization for DNN training speed-up. These approaches are applicable to both general training speed-up and on-device training. On the other hand, the fifth class leverages memory virtualization to speed up training. Specifically, this approach simultaneously utilizes both GPU and CPU memory for training speed-up. However, differing from servers where the CPU and GPU are paired with separate hardware memory, most mobile System-on-Chips (SoCs) adopt a unified memory architecture, which means different processors share a common physical memory. Furthermore, the mobile operating system, background workloads, and applications can cause unpredictable memory churn that leads to scarce and dynamic memory availability for DNN training. Therefore, memory virtualization approaches do not apply to on-device training.

Gradient accumulation: This approach aims to optimize memory usage by adopting incremental computation techniques for mini-batch operations [112]. The computation of gradients for an entire mini-batch is advantageous for maximizing hardware utilization through enhanced parallelism. However, platforms with limited resources may face challenges meeting the memory requirements for processing mini-batches in their entirety. In theory, mini-batches can be partitioned into smaller sub-mini-batches, referred to as micro-batches. In this approach, gradients for individual micro-batches are computed separately, and their results are aggregated to derive the gradient for the complete mini-batch. A critical aspect lies in determining the appropriate size for splitting the mini-batch, as excessively small batch sizes can result in under-utilization of hardware resources.

Gradient checkpointing: This method selectively retains a subset of activations in memory while recomputing demanded activations that are not stored [30]. This strategy involves a trade-off between memory utilization and

additional computation. The key challenge lies in effectively determining which activations to store and which to discard. Previous research efforts have explored computing optimal schedules to minimize computation overhead within a given hardware budget [47, 63]. More recently, dynamic gradient checkpointing has emerged as a heuristic approach to identifying on-memory activations with minimal computing overhead [58]. Gradient checkpointing effectively reduces memory footprints across activation layers, while accumulation operates over mini-batches. Importantly, these approaches are orthogonal and can seamlessly coexist [27].

Activation Size Reduction: This strategy is widely employed to mitigate the memory consumption of models and primarily involves quantization and sparsification techniques. Quantization methods utilize lower-precision floating-point or integer units for calculating and storing intermediate states, thereby reducing model operation costs [85]. Additionally, these methods are hardware-friendly, as most GPUs, including mobile GPUs, support 16-bit floating-point or 8-bit integer arithmetic, which offers efficient computation compared to the commonly used 32-bit floating-point arithmetic. However, while sparsification approaches also reduce gradient size, current hardware has exhibited inefficiencies in storing and computing sparse matrices [26].

Activation Number Reduction: Another viable approach for alleviating model computation memory is by reducing the number of intermediate states in a model. For example, researchers merge a series of arithmetic operations in a computational graph into a single composite operation to consolidate multiple gradients into a unified state. Such graph-level optimizations are employed for faster and memory-efficient inference [9, 103], although their application in training remains largely unexplored.

Memory Virtualization: This method leverages host-side memory in GPUs to logically extend usable memory. Several studies have employed such schemes for server-scale DNN training [39, 95, 101]. However, this approach is not suitable for mobile environments, as mobile SoCs typically employ a unified memory architecture for both the GPU and CPU.

2.3 Selected Systems and their device categories

The scope of this survey covers on-device training systems tailored for resource-constrained devices, with a focus on memory limitations. Specifically, we classify devices into three categories based on their hardware resources: high-constraint devices, middle-constraint devices, and low-constraint devices. High-constraint devices typically have hundreds of kilobytes of memory, while middle-constraint devices possess hundreds of megabytes. Low-constraint devices, on the other hand, have less than ten gigabytes of memory. Table 1 outlines the hardware specifications of the devices utilized in previous studies, while Table 2 provides an overview of the target devices for the systems research works we review.

Table 1. The hardware feature of targeted devices. All information from corresponding official websites.

Device	Memory	Computing Resource
NVIDIA Jetson TX2	8 GB RAM	Quad-Core ARM Cortex-A57 MPCore CPU 256-core NVIDIA Pascal GPU
NVIDIA Jetson Nano	4 GB RAM	Quad-core ARM Cortex-A57 MPCore CPU 128-core NVIDIA Pascal GPU
Samsung Galaxy S10	8 GB RAM	Exynos 9820 SoC: <i>Octa-Core CPU with two Exynos M4, two Cortex-A75 and four Cortex-A55 cores, and Mali-G76 GPU</i>

Continuation of Table 1		
Device	Memory	Computing Resource
Samsung Note 20	8 GB RAM	Snapdragon 865 SoC: <i>Octa-core Kryo 585 CPU, Adreno 650 GPU and Hexagon 698 DSP</i>
Samsung Note 10	8 GB RAM	Snapdragon 855 SoC: <i>Octa-core Kryo 485 CPU, Adreno 640 GPU and Hexagon 690 DSP</i>
Xiaomi 11 Pro	8 GB RAM	Snapdragon 888 SoC: <i>Octa-core Kryo 680 CPU, Adreno 660 GPU and Hexagon 780 DSP</i>
Xiaomi 10	8 GB RAM	Snapdragon 865 SoC: <i>Octa-core Kryo 585 CPU, Adreno 650 GPU and Hexagon 698 DSP</i>
Redmi Note 9 Pro (Mandheling [125])	8GB RAM	Snapdragon 750G SoC: <i>Octa-core CPU with two Kryo 570 cores and six Kryo 570 cores, Adreno 619 GPU and Hexagon 694 DSP</i>
Redmi Note 9 Pro (Melon [121])	6 GB RAM	Snapdragon 720G SoC: <i>Octa-core CPU with two Kryo 260 Gold cores and six Kryo 260 Silver cores, Adreno 618 GPU and Hexagon 692 DSP</i>
Redmi Note 8	4 GB RAM	Snapdragon 665 SoC: <i>Octa-core CPU with four Kryo 260 Gold cores and four Kryo 260 Silver cores, Adreno 610 GPU and Hexagon 686 DSP</i>
Meizu 16T	6GB RAM	Snapdragon 855 SoC: <i>Octa-core Kryo 485 CPU, Adreno 640 GPU and Hexagon 690 DSP</i>
vivo iQOO Neo3	6 GB RAM	Snapdragon 865 SoC: <i>Octa-core Kryo 585 CPU, Adreno 650 GPU and Hexagon 698 DSP</i>
Huawei nova 6 SE	8 GB RAM	Kirin 810 SoC: <i>Octa-core CPU with two ARM Cortex-A76 and six ARM Cortex-A55 cores, Mali-G52 GPU and Da Vinci NPU based DSP</i>
Google Pixel 3a	4 GB RAM	Snapdragon 670 SoC: <i>Octa-core Kryo 360 CPU, Adreno 615 GPU and Hexagon 685 DSP</i>
Google Pixel 2 XL	4 GB RAM	Snapdragon 835 SoC: <i>Octa-core Kryo 280 CPU, Adreno 540 GPU and Hexagon 682 DSP</i>
Google Pixel	4 GB RAM	Snapdragon 821 SoC: <i>Quad-core CPU with two 2.34 GHz Kryo cores and two 2.19GHz Kryo cores, Adreno 530 GPU and Hexagon 680 DSP</i>
Raspberry Pi 4B	4 GB RAM	Quad-Core ARM Cortex-A72 CPU
Raspberry Pi 4B+	2 GB RAM	Quad-Core ARM Cortex-A72 CPU
Raspberry Pi 3B+	1 GB RAM	Quad-Core ARM Cortex-A53 CPU
Samsung Gear S3	768 MB RAM	Exynos 7 Dual 7270 CPU
Raspberry Pi 1	256 MB RAM	ARM1176JZF-S 700 MHz CPU
Nordic Semiconductor nRF-52840	256 KB RAM	ARM Cortex-M4F CPU

Continuation of Table 1		
Device	Memory	Computing Resource
STMicroelectronics STM32F746	320 KB SRAM	ARM Cortex-M7 CPU
Arduino Nano 33 BLE	256 KB SRAM	ARM Cortex-M4 CPU
Arduino MKR1000	32 KB SRAM	SAMD21 Cortex-M0+ ARM MCU
End of Table 1		

2.4 One-Stage Systems

This section provides a summary and overview of two one-stage systems: MiniLearn [97] and TinyTL [6]. One-stage systems share a common characteristic in that they are designed to address training scenarios requiring fewer computations. Specifically, they either train simpler NN architectures, such as CNNs with fewer than ten layers, or focus on tuning only a subset of parameters within a complex NN model, such as biases in CNNs while keeping the filter parameters fixed. The main ideas behind these systems revolve around two key approaches. Firstly, there is a focus on minimizing the size of the model, as exemplified by MiniLearn’s retraining of quantized models and pruning of filters [90]. Secondly, efforts are made to reduce the number of parameters requiring training. For example, TinyTL exclusively tunes biases, while MiniLearn fine-tunes only the fully connected layers. These two aspects represent common efficiency metrics in ML. It is imperative for future researchers to bear these considerations in mind, given that on-device training research inherently intersects with the realm of efficient embedded ML.

2.4.1 Training Simple Neural Networks. MiniLearn re-trains Convolutional Neural Networks (CNNs) on IoT devices to empower on-device ML applications to adapt dynamically to real deployment environments, overcoming the gap between pre-collected training datasets and real-world data. The target device in this work is the nRF-52840 SoC microcontroller, which features a 32-bit ARM Cortex-M4 with FPU at 64 MHz, 256 KB of RAM, and 1 MB of Flash. MiniLearn applies quantization and dequantization techniques to enable model re-training on resource-constrained devices. The key idea is to store the weights and intermediate outputs in integer precision and dequantize them to floating-point precision during training. MiniLearn consists of three steps: (1) Dequantizing and pruning filters, (2) Training filters, and (3) Fine-tuning fully connected layers.

Dequantizing and pruning of filters: First, MiniLearn performs dequantization, keeping the first layer in integer precision format to retain the input shape (which is a tensor with a specific shape) of the neural network (NN), while dequantizing the remaining layers to floating-point format. Next, during training, MiniLearn prunes less significant filters, using the L1-Norm as a static method for selecting filters. Filters with small L1 norms are considered less important in the network. Finally, MiniLearn adjusts the output layer based on the fine-tuning demand, i.e., it adjusts the number of neurons in the output layer according to the number of target output classes.

Training the filters: MiniLearn preprocesses the training set and filters by collecting the output of the first layer in compressed quantized (INT8) format as the training set, which is converted to a floating-point format during training. Additionally, MiniLearn initializes models based on quantized pre-trained models and converts them to floating-point format. After pre-processing, MiniLearn performs its training operations.

Table 2. Overview of targeted devices and major design features of current systems.

Research	Device Category	Device	Design Features
LifeLearner [65]	Low-constraint	NVIDIA Jetson Nano Raspberry Pi 3B+	Continual Learning-based System, Meta-CL [48]
Sage [27]	Low-constraint	Smartphones: <i>Samsung Galaxy S10</i> and <i>Samsung Note 20</i>	Two-Stage System, Micro batch [42] and Recomputation [10]
Mandheling [125]	Low-constraint	Smartphones: <i>Xiaomi 10</i> , <i>Xiaomi 11 Pro</i> and <i>Redmi Note 9 Pro</i>	Two-Stage System, Digital Signal Processors
Melon [121]	Low-constraint	Smartphones: <i>Samsung Note 10</i> , <i>Redmi Note 8</i> , <i>Redmi Note 9 Pro</i> , <i>vivo IQOO Neo3</i> , <i>Meizu 16T</i>	Two-Stage System, Micro batch and Recomputation
zTT [55]	Low-constraint	NVIDIA Jetson TX2	Reinforcement Learning Based System
ElasticTrainer [41]	Low-constraint Middle-constraint	Smartphones: <i>Google Pixel 3a</i> NVIDIA Jetson TX2 Raspberry Pi 4B	Two-Stage System
MDLdroidLite[134]	Low-constraint Middle-constraint	Smartphones: <i>Huawei nova 6 SE</i> , <i>Google Pixel</i> , <i>Google Pixel 2 XL</i> and <i>Samsung Gear S3</i>	Runtime Elastic Tensor Selection Online Continuous Growth-Based System
TinyTL [6]	Middle-constraint	Raspberry Pi 1	One-Stage System, Fine-tuning
MimiLearn [97]	High-constraint	Nordic Semiconductor nRF-52840	One-Stage System, Quantization [46] Pruning [90]
TTE [74]	High-constraint	STMicroelectronics STM32F746	Two-Stage System, Quantization-Aware Scaling
POET [94]	Low-constraint Middle-constraint High-constraint	NVIDIA Jetson TX2 Raspberry Pi 4B+ Arduino MKR1000 Nordic Semiconductor nRF-52840	Two-Stage method, Mixed Integer Linear Programming

Fine-tuning fully connected layers: MiniLearn quantizes the pruned filters back to integer precision format and freezes them. It then performs a few extra epochs of fine-tuning on the fully connected layers to compensate for the potential loss of information caused by pruning and quantization.

2.4.2 Tuning Part of the Neural Network. TinyTL aims to enable on-device training of deep CNNs on memory-constrained devices. The authors demonstrate this by training the ProxylessNAS-Mobile model [7] on a Raspberry Pi 1, which only has 256 MB RAM. The authors argue that techniques like pruning, which reduces trainable parameters, do not fully solve the memory footprint problem in on-device training. To address this issue, TinyTL only tunes the biases of the CNN while freezing the parameters of filters to significantly reduce the memory footprint. This is because the number of bias parameters is far less than the number of filter parameters in a CNN.

However, only tuning the biases can lead to limited generalization ability. To address this issue, the authors propose the lite residual module, which employs group convolution [61] and a 2×2 average pooling to downsample the input feature map and reduce the number of channels. This module refines the intermediate feature maps and helps to improve the model’s generalization ability. The workflow for the lite residual module is similar to the biased module. TinyTL adds the lite residual module to the CNN model to reduce the memory footprint of training while maintaining high model performance.

2.5 Two-Stage Systems

The two-stage paradigm is popular in current systems research, with POET [94], Mandheling [125], Melon [121], Sage [27], and TTE [74] following this paradigm. As mentioned earlier, two-stage systems have an additional preparation stage compared to one-stage systems. During the preparation stage, these systems pre-process models, generate the computing graph (DAG), and/or find an optimal or suboptimal execution plan for model training. Two-stage systems exhibit a variety of characteristics that can be categorized into three distinct types. Firstly, some systems primarily focus on addressing the memory constraints inherent in on-device training. Unlike one-stage systems, which may only fine-tune or partially train models, these systems tackle the challenge of training full-fledged NN models, such as BERT-small [16] and MobileNetV1 [38], where memory limitations pose the primary bottleneck. Secondly, certain systems leverage auxiliary hardware to facilitate on-device training. Recognizing that resource-constrained devices often lack powerful GPUs, these systems explore the feasibility and potential of utilizing available hardware components such as DSPs to accelerate on-device training tasks. Lastly, there are systems designed specifically to enable model training on high-constraint devices. These systems face unique challenges, including energy concerns and extremely limited memory, when training models on devices with more constrained resources.

It is important to note that the execution plan schedules the training process by applying suitable techniques according to the characteristics of operations. However, this does not mean that the training will strictly follow one chosen schedule during runtime. The budgets of resources are dynamic during runtime, and hence, the optimal schedule is also dynamically changing. Therefore, the schedule should also be adapted correspondingly. For example, Melon generates several execution plans during the first stage and switches between them during runtime according to the dynamic runtime budgets. Melon’s evaluation shows that this achieves better results than statically following one schedule.

2.5.1 Systems Focusing on Memory Bottleneck. This section provides an overview of systems that target solving memory bottlenecks in on-device training, including Melon [121] and Sage [27]. While techniques explored in cloud-based systems may also be applicable to on-device training, the design of on-device training systems needs to be judicious. Melon and Sage both leverage two well-explored techniques from cloud-based systems, namely micro-batching and recomputation, to cope with memory constraints, as discussed in Section 2.2. These papers jointly apply these techniques to address memory limitations. However, because the micro-batch method can interfere with training results when

models include batch normalization (BN) layers due to the introduction of cross-sample dependencies within a batch, additional optimization designs are introduced in on-device training systems to facilitate training. For instance, Melon employs a tensor-lifetime-aware algorithm for memory layout optimization, while Sage performs both graph- and operator-level optimizations for the computation graph (DAG).

Melon: Melon is designed based on the observation that constrained memory hinders training performance. Specifically, large enough batch sizes and batch normalization layers are two key factors for guaranteeing the convergence accuracy of large NNs. However, resource-constrained devices cannot support large batch sizes during training, as this leads to large peak memory usage. When training a model with batch normalization layers, the mean and deviation of small-batch samples are not representative enough of the distribution of the whole dataset. This slows down the training process and reduces the model’s generalization ability. Melon aims to solve these two problems. The authors present a memory-calibrated progressive recomputation method based on the tensor-lifetime-aware memory pool. The method takes the whole operator DAG as input, and to estimate the benefit of recomputing each tensor, the authors define a metric called Triangle Per Second (TPS) as shown in Equation 1.

$$TPS = \frac{TensorSize \times FreedLifetime}{ResomputationTime} \quad (1)$$

where *TensorSize* refers to the size of activations, *FreedLifetime* denotes the lifetime span between discarding and recomputing, and *ResomputationTime* indicates the time it costs to recompute the discarded activation.

Melon integrates micro-batch [42] and recomputation [10] to reduce memory consumption. The main idea is to minimize the recomputation overhead by scheduling when and which tensors (intermediate output of hidden layers) should be discarded or recomputed. Melon employs a tensor-lifetime-aware algorithm for memory layout optimization since the performance of recomputation highly depends on memory management. Melon defines the tensors into two categories and manages them respectively. Firstly, for tensors with long lifetimes, such as a tensor produced at the forward pass and released at the backward pass, Melon follows a First Produced Last Release order. For others, it follows a greedy way to produce and release them. Overall, the key idea is to place those long-lifetime tensors beneath short-lifetime ones to consolidate the overall memory layout. When performing recomputation, Melon will continuously discard tensors with maximal TPS and calibrate the memory pool until the pool size is smaller than the budget.

First stage: Melon denotes its first stage as the decision stage. At this stage, it generates the optimal execution plan based on the given budget. Specifically, it obtains runtime information via an execution profiler that contains NN operators and intermediate tensors. The tensor information consists of data flow dependencies, the computation time of each operator, as well as the size and lifetime of each tensor. Based on the profiler, Melon generates an execution plan that determines (1) where each tensor is placed in a large memory pool, (2) which operators need to be recomputed, and (3) how to split the batch. Melon generates multiple plans for different budgets but does so only once.

Second stage: In the execution stage, Melon performs the training based on a proper plan. Notably, Melon can switch to a more proper plan when the memory budget changes. The switching strategy is a "lazy strategy", i.e., switching to a new plan when the current training batch ends.

Sage: Sage aims to enable SOTA DNN training on smartphones. Specifically, the authors implemented Sage to train ResNet-50 [33], DenseNet-121 [40], MobileNetV2 [107], and BERT-small [16] on smartphones with different SoCs, including Samsung Galaxy S10 and Note 20. The authors also demonstrate that the scarce and dynamic memory of mobile devices is the major bottleneck of on-device training, with memory usage during training being 5 to 100 times higher than during inference. To address this issue, Sage employs dynamic gradient checkpointing [58] and dynamic

gradient accumulation to dynamically adjust the memory usage to the available system memory budget. Dynamic gradient checkpointing shares the same idea with materialization, dropping intermediate results and recomputing them when needed. For dynamic gradient accumulation, the authors use a traditional micro-batch technique [112] that can dynamically adjust the size of the micro-batch according to the available memory during runtime to reduce peak memory usage.

First Stage: In its first stage, Sage constructs a computation graph (DAG) with automatic differentiation. Concretely, a DNN node is expressed by differentiable operations and computable operations abstraction. In this way, the DAG decouples the evaluation and differentiation processes of DNN computation graphs. Then, Sage performs both graph- and operator-level optimizations for the DAG.

Second stage: This stage is also called the graph execution stage. Sage employs a hybrid strategy to combine gradient checkpointing and gradient accumulation during runtime adaptively to execute the DAG.

2.5.2 Systems Leveraging Auxiliary Hardware. Hardware acceleration is a common approach as a general training speedup technique. However, resource-constrained devices typically lack accelerators dedicated to on-device training, which are prevalent in many specialized servers. As discussed in Section 1.2, some devices, such as smartphones, are equipped with hardware accelerators, e.g., mobile GPUs. Nevertheless, these accelerators typically do not possess dedicated peripherals or resources, such as memory and power. Instead, they share many of the resources with other components consisting the system. This means that the excessive use of resources at the accelerators can lead to overall system inefficiency and potentially lead to degradation of the user experience [96, 109].

As an alternative, on-device training systems can leverage other auxiliary hardware to accelerate their training tasks, for example, Mandheling [125] enables on-device training with DSP offloading. By utilizing DSPs or similar hardware, on-device training systems can overcome the limitations imposed by the absence of GPUs and improve the efficiency of model training on resource-constrained devices.

Mandheling: is designed to train DNNs on modern smartphones. The author’s starting point is that DSPs are particularly suitable for integer operations, and DSPs are ubiquitously available on modern smartphones. Therefore, the authors leverage CPU-DSP co-scheduling with mixed-precision training algorithms to enable on-device training.

First stage: Denoted as the preparing stage, here, Mandheling initiates the model transformation. The target models could either be pre-trained or randomly initialized based on different frameworks, such as TensorFlow and PyTorch. Given a model, Mandheling translates it into FlatBuffer-format, a format with quantization used in TensorFlow Lite. In the execution stage, Mandheling generates CPU and DSP compute subgraphs, then performs compute-subgraph execution on devices.

Second stage: In the execution stage, Mandheling fully leverages DSPs for efficient model training by using four key techniques: (i) CPU-DSP co-scheduling, (ii) self-adaptive rescaling, (iii) batch splitting, and finally, (iv) DSP-computes subgraph reuse.

CPU-DSP co-scheduling reduces the overhead of CPU-DSP context switching by avoiding the offloading of DSP-unfriendly operators to the DSP. Examples of such operators include normalization, dynamic rescaling, and compute-graph preparation. Normalization is considered DSP-unfriendly due to its irregular memory access.

Self-adaptive rescaling refers to periodically performing rescaling instead of performing it every batch. Rescaling involves adjusting the scale factor, which is a trainable parameter used to scale the output of each layer. Dynamic rescaling adjusts the scale factor per batch and guarantees convergence accuracy but adds at least two times the latency compared to static rescaling, which uses a fixed scale factor. In their experiments, the authors observed that the scale

factor jumps between 10 and 11, and the frequency of these changes is low, occurring once every 10 to 60 batches. Based on this observation, the authors propose to rescale periodically instead of after each batch.

The batch-splitting phase splits batches based on their workload. Specifically, Mandheling first identifies abnormal batches with noticeably higher workloads, then splits these batches into multiple micro-batches to execute them individually. DSP-compute subgraph reuse is the fourth technique used in Mandheling. The DSP-compute subgraph comprises operators with inputs, outputs, and parameters. In the current training method, a new compute subgraph is prepared for each training batch. However, the authors find that models are rarely modified during training. Therefore, reusing the compute subgraph eliminates the preparation overhead. Additionally, Mandheling releases the most recently used (MRU) memory when the memory runs out. This is because the allocation/deallocation of memory for subgraph reuse always follows the execution order of DNNs, which means that the MRU memory region has the longest reuse distance.

2.5.3 Systems Designed for High-Constrained Devices. This section provides an overview of systems designed to enable model training on high-constrained devices, such as Private Optimal Energy Training (POET) [94] and Tiny Training Engine (TTE) [74]. These systems focus on optimizing the training process and applying general optimization techniques to overcome the constraints of resource-intensive tasks on such devices. POET approaches the challenge by abstracting training memory and computation costs into Mixed Integer Linear Programming (MILP) problems. By solving these MILP problems, POET generates a training schedule optimized for efficiency. Additionally, POET applies techniques like paging and gradient checkpointing to further optimize the training process. On the other hand, TTE takes a different approach by separating auto-differentiation from runtime and moving it to compile time. By doing so, TTE aims to streamline the training process and improve efficiency. Additionally, TTE leverages pruning techniques to accelerate training even further, making it suitable for high-constrained devices where computational resources are limited.

POET: POET aims to enable the training of SOTA neural networks on memory-scarce and battery-operated edge devices. The authors devised POET to allow for on-device training of DNNs, such as ResNet-18 [33] and BERT [16]. The target devices include the ARM Cortex M0 class MKR1000, ARM Cortex M4F class nrf52840, A72 class Raspberry Pi 4B+, and Nvidia Jetson TX2.

First stage: Firstly, given an NN model, POET profiles the memory and computation costs of its operators. Operators refer to the computational operations of the ML model, such as non-linear functions and convolutions. Based on the resource cost of operators, POET selects a suitable technology to optimize them during training. Secondly, taking into account hardware constraints, POET generates a Mixed Integer Linear Programming (MILP) problem to search for the optimal schedule of paging [115] and rematerialization [10]. Finally, POET sends the schedule to the target edge devices to carry out memory-efficient ML training. The objective function of the MILP is denoted as follows:

$$\min \sum_T [R\Phi_{compute} + M_{in}\Phi_{pagein} + M_{out}\Phi_{pageout}]T \quad (2)$$

where $\Phi_{compute}$, Φ_{pagein} , and $\Phi_{pageout}$ denote the energy consumption of each computing node, page in, and page out operations, respectively. R refers to recomputing operations, M_{in} represents paging a tensor from secondary storage to RAM, and M_{out} refers to the opposite operation.

Second stage: POET performs model training by following the paging and rematerialization schedule generated in the first stage.

TTE: The authors present the TTE to tackle the two main challenges of on-device training, namely: (1) the difficulty in optimizing quantized models. The deployed model is quantized, which leads to lower convergence accuracy than the unquantized model due to low precision and lack of batch normalization layers. (2) Limited hardware resources of tiny devices. The memory usage of full back-propagation can easily exceed the memory of microcontrollers. The authors conduct experiments using three popular TinyML models: MobileNetV2, ProxylessNAS, and MCUNet [73]. The experiments are performed on a microcontroller STM32F746 that features 320KB SRAM and 1MB Flash using a single batch size. The authors verify that the quantization process distorts the gradient update, causing the lower convergence accuracy problem of quantified models. To address this problem, they propose quantization-aware scaling (QAS). The key idea of QAS is to multiply the square of scaling factors corresponding to precision with intermittent output to relieve the disorder caused by quantization. Then, TTE combines QAS and sparse update techniques to overcome the second challenge.

First stage: In the first stage, the TTE engine works in three steps. Firstly, TTE moves the auto-differentiation to the compile time from the runtime and generates a static backward computing graph. Secondly, TTE prunes away the gradient nodes with the sparse layer update method. Thirdly, TTE reorders the operators to immediately apply the gradient update to a specific tensor before back-propagating to earlier layers so that the memory occupied by the gradient tensors can be released as soon as possible.

Second stage: In the second stage, TTE executes the computing graph from the first stage.

2.5.4 Systems Focusing on Training Speedup. On-device training can be time-consuming due to constrained hardware resources. Hence, training speedup is essential for on-device training. To speed up the on-device training process, previous research focuses on three main approaches. The first approach offloads the entire DNN training operation to the cloud and then only trains a selected portion of DNN on the device [18, 108], by performing transfer learning. However, this approach reduces the learning capability of models, which leads to a drop in accuracy [6]. The second approach starts with a tiny structure and gradually grows the DNN during training until satisfactory accuracy is achieved. This method is called continuous growth [45]. Continuous growth has been applied in MDLdroidLite [134], which we discuss in the next section. However, a continuous growth-based system must train newly added NN layers from scratch, requiring over 30% extra training epochs [134]. The third approach adaptively adjusts the trainable portion of a DNN. This approach gradually removes less important layers on the fly during training [34, 89]. However, this selection of layers is an unrecoverable process, which means that the removed pruned portions can never be selected again even if they may have more importance later. Therefore, this approach cannot flexibly adapt to changing requirements.

ElasticTrainer: ElasticTrainer [41] presents an elastic runtime adaptation approach for speedup of on-device training, where every DNN substructure can be freely removed from or added to the trainable DNN portion at any time in training. To achieve elasticity, the key is to select the optimal trainable NN portion at runtime, which meets the desired training speedup with the minimum accuracy loss. The authors design a two-stage system to achieve this target.

First stage: In the first stage, ElasticTrainer exploits a tensor timing profiler to profile the estimated training time for the selected DNN sub-architecture. Concretely, the authors first convert the original DNN structure into a tensor-level computing graph, which retains the execution order of all tensor operations during training. Then, they use a standard DNN profiler⁵ to measure the execution time of each operation, such as matrix multiplication and convolution. Finally, the training time of each tensor is computed according to the timings of related operations.

⁵<https://www.tensorflow.org/guide/profiler>

Second stage: In the second stage, ElasticTrainer considers both accuracy and training time in the selection of the optimal trainable DNN portion. However, it is practically difficult to determine whether the required accuracy could be achieved within reasonable training time in advance. Therefore, ElasticTrainer uses the training loss reduction at the desired training time as the selection metric instead. ElasticTrainer defines the selection problem as a constrained optimization problem as follows:

$$\max \Delta_{\text{loss}}(\mathcal{M}) \quad \text{s.t.} \quad T_{\text{selective}}(\mathcal{M}) \leq \rho T_{\text{full}}, \quad (3)$$

where \mathcal{M} refers to a binary mask for tensor selection, with its j -th element indicating if the j -th tensor is selected. For example, if $\mathcal{M} = [0, 1, 1, 0]$, the second and third tensors will be trainable tensors, and others will not be trained. $T_{\text{selective}}$ denotes the total accumulated estimated training time of selected tensors. The estimated training time of each tensor is computed by the tensor timing profiler in the first stage. T_{full} refers to the full training time (the time required to train the full DNN). ρ depicts the expected training time reduction ratio, with $\rho = 50\%$ referring that the training time should be reduced to half of the full training time.

2.6 Continual Learning-Based Systems

Continual learning enables DNN models to learn new tasks over longer durations. These models retain and use the knowledge learned from previous tasks to learn new tasks more efficiently and effectively [92, 110]. However, they generally suffer from the catastrophic forgetting (CF) issue [83], which means the model severely forgets previously learned knowledge while learning new data. To alleviate CF, researchers propose three common approaches [92]: regularization approaches, dynamic architectures, and memory replay. Regularization approaches impose constraints on updating the weights of neural networks to alleviate CF. This method generally penalizes differences between the weights for the old and the new tasks to slow down the update of task-relevant weights. The dynamic architecture approach dynamically adapts the model’s architecture in response to new tasks, such as re-training the model with an increased number of neurons or network layers. MDLdroidLite [134] uses the dynamic architectures approach in its on-device training process. The memory replay approach leverages a dual-memory learning mechanism to cope with CF, which is inspired by the complementary learning systems theory [62, 82]. In the dual-memory learning mechanism, the model has two types of weights: fast change weight for temporary knowledge and slow change weight, which stores long-term knowledge [35]. LifeLearner [65] and Miro [80] both apply the memory replay approach. However, as mentioned before, the mismatch between hardware resource availability and demand is one of the key challenges of on-device training. Continual lifelong learning-based on-device training systems also optimize the continual learning approaches for resource efficiency.

MDLdroidLite: MDLdroidLite uses the dynamic architectures approach for on-device training. Concretely, MDLdroidLite starts with a tiny structure, which is only 1%-10% of the full model size, and gradually grows the DNN during training until achieving satisfactory accuracy. The method is called continuous growth. The inspiration is that traditional DNN models often have large redundant parameters that can be reduced through pruning techniques, as shown in previous studies [25]. This continuous growth approach has been explored in previous works that apply knowledge transfer to fast-tune newly grown neurons or layers using techniques such as net2net [14, 20, 45]. However, current methods may still generate models with redundant parameters that do not fit mobile devices well and exhibit slow convergence rates that cannot guarantee convergence on resource-constrained devices.

To address these challenges, the authors introduce a release-and-inhibit (RIC) control and RIC-adaption pipeline. The RIC control manages the layer-level growth of DNN models to enable independent layer growth and avoid redundant parameter growth. The RIC-adaption pipeline addresses the slow convergence rate issue by adapting the RIC control during training.

RIC control: The reward function used in the RIC control of MDLdroidLite is defined as the difference in accuracy achieved by growing a layer and the accuracy achieved without growing the layer. More formally, the reward function is defined as follows:

$$Reward = \frac{G(\cdot)}{C(\cdot)} \quad (4)$$

where $G(\cdot)$ and $C(\cdot)$ refer to the state-value and state-cost function, respectively. These two functions measure the difference in value and cost between states before and after one growth action. The state-value function measures the gains of growth and the state-value function measures the extra cost from growth, that defines as $C(\cdot) = (1 - \beta)Flops(\cdot) + \beta Size(\cdot)$, where $Flops(\cdot)$ and $Size(\cdot)$ denote FLOPs of the grown layers and the size of the grown layers respectively. β refers to a normalization coefficient.

RIC adaption pipeline: To overcome the low convergence rate caused by parameter growth, the authors propose a three-step adaptation pipeline that safely adapts newly added parameters without sacrificing accuracy in each growth step. Firstly, they select neurons or channels with small variance using a cosine similarity function to add to the NN. Secondly, they preserve loss reduction by applying a safe parameter scale function. Specifically, they initialize the parameters using the Kaiming method [32], which is based on the distribution of parameters in the full-sized model. Lastly, they map the layer with newly added neurons to subsequent layers to maintain the relationship between layers. This three-step adaptation pipeline ensures that the newly added parameters are adapted safely and effectively, enabling faster convergence and higher accuracy. Overall, MDLdroidLite enables the on-device training of SOTA DNNs from scratch.

LifeLearner: LifeLearner leverages meta-continual learning (Meta-CL) and the memory replay approach for its on-device training operations. Meta-CL is a specialized branch of continual learning that incorporates meta-learning techniques to enhance adaptability and mitigate forgetting across sequential tasks [3, 48, 102]. Meta-learning helps create a model that can quickly adapt to new tasks with minimal updates, which consists of two phases: meta-training and meta-testing. The meta-training phase seeks to find an optimal initialization for weights of the DNN model, which is generally performed on offline servers. The meta-testing phase further tunes the initialized DNN model with a few new data samples. This phase could run on embedded computing platforms. To alleviate the forgetting problem, prior works in Meta-CL [3, 48, 69] split the DNN architecture into a feature extractor and a classifier, and adapt the idea of memory replay at an architecture level. During the meta-training phase, prior works in Meta-CL update the feature extractor (slow change weights) in an outer loop with random samples from learned classes to store long-term knowledge, and the classifier (fast change weights) in the inner loop (fast weights) to learn new temporary knowledge. However, after deployment, prior works in this domain rely on the inner loop (updating the classifier) in the meta-testing phase for learning, which leads to limited generalization capability [65]. Furthermore, those methods incur high computation and memory costs.

To address these challenges, the authors of LifeLearner first design a deployment-time training optimization mechanism for utilizing memory replay-based Meta-CL to resolve the accuracy degradation problem, referred to as the Co-utilization of Meta-Learning and Rehearsal Strategy. Secondly, they present an optimization method that provides a

computation-efficient replay strategy to minimize the computation overhead along with a compression module for the compression and storage of replay samples to induce memory usage, termed as CL-tailored Algorithm/Software Co-Design.

Co-utilization of Meta-Learning and Rehearsal Strategy: The authors introduce a replay buffer for storing samples of classes already learned, which are mixed with samples from new classes. This buffer prevents the catastrophic forgetting issue of continual learning. In addition, they design a deployment-time training optimization mechanism. After deployment, the system performs inner-loop training with samples of new classes to learn new temporary knowledge and also outer-loop training with samples from learned classes for retaining long-term knowledge. This mechanism brings extra costs both in computation and memory. To alleviate the overhead cost, the authors present a compression module as described below.

CL-tailored Algorithm/Software Co-Design: LifeLearner exploits a replay strategy and a compression module to minimize the overhead cost from the deployment-time training optimization mechanism. Concretely, the replay strategy is to store intermediate activations instead of raw data samples. LifeLearner stores the output of the last layer of the model’s feature extractor as replay data samples for inner-loop updating. During inner-loop updating, it freezes the feature extractor and only performs training on the classifier. In addition, on the memory front, the authors observe that the activations are sparse due to using the ReLU non-linearity activation function, i.e., more than 90% of the activation values of the hidden layer are zero. The sparsity facilitates the compression and subsequent efficient storage of activations on-device.

Therefore, the authors design a compression module for replay samples. The compression module consists of two stages: sparse bitmap compression and product quantization. The sparse bitmap compression stage filters out the majority of zero values in activations. Specifically, for compression, it firstly creates a bitmap with the same dimensions as the activations, then sets a bit to 1 for non-zero values’ indices and 0 for the remainders, then stores the non-zero values in a vector with 32-bit floats format and the indices bitmap in bitmap format. For decompression it firstly traverses the indices bitmap and the non-zero values vector, then reconstructs the activations by using either the saved non-zero value or zero according to the indice is 1 or 0. In the second stage, the authors leverage production quantization [49] to further compress the vector stores non-zero values in the first stage.

Miro: As previously mentioned, the authors of Miro only evaluate their system using NVIDIA Jetson platforms, which is outside the scope of this survey. However, we believe that Miro holds potential for further optimization. Future research could explore its feasibility on more resource-constrained devices, such as Raspberry Pi. Therefore, we also provide a summary of the Miro system. Concretely, Miro enables on-device training by leveraging and optimizing hierarchical episodic memory (HEM) [70], an optimized variant of episodic memory (EM) [79]. EM is a memory replay method that alleviates the CF issue. With EM, the system stores old data samples in the memory and replays them together with new data samples during continuous learning. However, EM incurs large memory costs that limit its applications in hardware-constrained devices. HEM is an optimized variant of EM that leverages hierarchical memory to reduce memory usage. Concretely, HEM stores a small set of old data samples in memory with fast access and a large number of old data samples in storage with slow access. During training, HEM fetches old samples in memory to train the model and performs an online data swapping to swap old data samples between memory and storage. With this strategy, HEM balances efficiency and accuracy.

2.7 Reinforcement Learning-Based Systems

Indeed, reinforcement learning (RL) presents a promising approach for on-device training. In RL, an agent learns to make decisions by interacting with an environment to maximize some notion of cumulative reward [113]. This iterative process of adjusting the agent’s policy based on feedback from the environment aligns well with the concept of on-device training. By leveraging RL techniques, on-device training systems dynamically adapt their model training strategies based on real-time feedback from the device’s environment. This allows for a flexible and adaptive learning process, particularly suitable for resource-constrained devices where traditional training methods may not be feasible due to limited computational resources or memory constraints. Overall, RL represents a powerful paradigm for on-device training, offering the potential for efficient and adaptive model training tailored to the specific constraints and requirements of resource-constrained devices.

zTT: zTT [55] presents a novel approach to DVFS for mobile devices, which employs Deep Q-Learning (DQN) [31, 87, 122] to overcome the shortcomings of traditional DVFS techniques. Previous DVFS methods for mobile devices, including algorithmic-based approaches, graphics-based approaches, dynamic techniques, and thermal-aware energy management, have encountered various difficulties: Firstly, they are not capable of adjusting power distribution between CPU and GPU based on application Quality of Experience (QoE) because they are application-agnostic. Secondly, most DVFS methods lead to overheating issues in mobile devices due to a lack of cooling methods. Thirdly, supervised learning-based DVFS methods are limited in their performance because mobile devices operate in complex and changing external environments. ZTT tackles these limitations by leveraging DQN to learn and adapt power distribution based on actual performance, while also managing heat generation and adjusting to changing external environments.

The authors’ proposed system is based on three key observations. Firstly, the performance of mobile device applications relies heavily on the CPU and GPU and can be optimized by controlling their clock frequencies using DVFS methods that consider the specific performance characteristics of each application, such as CPU- or GPU-intensive tasks. Secondly, the temperature of the mobile processor rises with increased heat generation due to the thermal coupling between the CPU and GPU in the confined space of mobile devices. Finally, mobile devices are susceptible to environmental changes, such as ambient temperature and whether the device is in a protective case.

System Design: The authors designed and implemented zTT using DQN to address the aforementioned issues. Their aim is to obtain the optimal policy that controls the CPU/GPU frequency to achieve the best possible performance with the least power consumption. The definition of performance may vary depending on the application type. In this study, the authors focus on video and gaming applications as the target applications and employ frame rate as the performance metric. The design of the Deep RL-based method is outlined below.

State: zTT defines a state using seven parameters: $\{f_C(t), f_G(t), T_C(t), T_G(t), P_C(t), P_G(t), x(t)\}$, where $f_C(t)$ and $f_G(t)$ refer to the configured frequency of CPU and GPU at time step t , $T_C(t)$ and $T_G(t)$ denote the temperature of CPU and GPU at time step t , $P_C(t)$ and $P_G(t)$ represent the power consumption of CPU and GPU at time step t , and $x(t)$ refers to the frame rate at time step t .

Action: The action set of zTT consists of three actions: exploration, exploitation, and cool-down. Actions adjust the clock frequency of the CPU and GPU. Concretely, zTT follows the ϵ -greedy fashion. Within the temperature threshold, zTT respectively takes exploration and exploitation action with probability ϵ and $1 - \epsilon$. When the temperature gets close to the threshold, zTT does a cool-down action that randomly selects clock frequencies lower than the current ones.

Rewards: The reward function is defined as Equation 5, where $U(t)$ refers to the performance (frame rate) at time step t , $P(t)$ represents power consumption at time step t and β denotes a trade-off weight between performance and power consumption. In addition, $W(t)$ is a compensation function that gives a positive reward to actions that result in temperature under the threshold and a negative reward to actions that lead to temperature exceeding the threshold.

$$R = U(t) + \frac{\beta}{P(t)} + W(t) \quad (5)$$

Neural Networks: zTT is based on transfer learning. The approach applies a pre-trained fully connected neural network model as the initial Q-function.

3 Review from the Machine Learning Perspective

This section analyzes current systems from the ML perspective. The analysis focuses on two aspects: the categories of NNs evaluated in the selected papers and the training strategy.

3.1 Evaluated Neural Networks

Table 3 shows neural networks used to evaluate on-device training systems, which consist of four categories: convolutional neural networks (CNN), transformer [117]-based NN, unsupervised learning NN, and neural architecture search (NAS)-based NN. Generally, these neural networks cover the most widely used NN in various fields. Concretely, CNNs include VGG [111], ResNet [33], InceptionV3 [114], DenseNet [40], MobileNet [38], SqueezeNet [43], and LeNet [68]. The transformer-based NNs include BERT [16], BERT-small [16], and Vision Transformer (ViT) [19]. Evaluated unsupervised learning NN is Autoencoder [36], a kind of unsupervised learning technique for representation learning. It is widely used in knowledge distillation [29]. In addition, the NAS-based NNs include ProxylessNAS [7], ProxylessNAS-Mobile, and MCUNet [73]. NAS is widely used in the automatic design of NNs and outperforms hand-designed architectures in many cases [136, 137]. In particular, POET [94], Mandheling [125], Sage [27], Melon [121], TTE [74], and MDLdroidLite [134] are all evaluated by training various CNNs. In addition, MiniLearn [97] is designed for training CNNs on devices. ElasticTrainer [41] is evaluated with CNNs models and ViT. In addition, for a fair comparison, LifeLearner [65] employs CNNs for evaluation, which are used in its prior relevant works. Apart from CNNs, BERT is another popular model for on-device training works, such as POET and Sage, which is widely used and significantly successful in natural language processing tasks. Furthermore, the authors of TinyTL [6] and TTE [74] also evaluated their system on ProxylessNAS.

3.2 Deep Neural Network Training Strategies

3.2.1 Transfer Learning. In terms of training strategies, all systems, except for Mandheling, utilize transfer learning. The authors of Mandheling perform both transfer learning and training from scratch, which involves initializing the model with random numbers. The authors conclude that transfer learning is more suitable for resource-constrained devices as it converges in fewer iterations, resulting in lower resource usage.

The workflow of transfer learning involves first pre-training models on cloud/servers, then deploying the model, and finally, updating the model through training on the device. Training models from scratch is too resource-intensive and energy-consuming for devices, even with SOTA optimization methods [125]. However, it is unnecessary to train models from scratch as the ML community has presented numerous NNs, datasets, and pre-trained models. Depending on the fact at hand, system researchers can select a suitable NN and train a model in the cloud on a large and general dataset, then deploy the model to devices and fine-tune the model on a new and small dataset. This workflow is more

Table 3. Overview of evaluated Neural Networks by current systems research.

Systems	Neural Networks
POET	VGG16 [111], ResNet-18 [33] and BERT [16]
Mandheling	VGG-11/16/19 [111], ResNet-18/34 [33], LeNet [68] and InceptionV3 [114]
Sage	ResNet-50 [33], DenseNet-121 [40], MobileNetV2 [107] and BERT-small [16]
Melon	MobileNetV1 [38], MobileNetV2, SqueezeNet [43] and ResNet-50
TinyTL	ResNet-50 and MobileNetV2
MiniLearn	CNNs with three to four convolutional layers and two to three fully connected layers
TTE	MobileNetV2, ProxylessNAS [7] and MCUNet [73]
zTT	Deep Q Network [88]
MDLdroidLite	LeNet, MobileNet and VGG
LifeLearner	CNNs with fully-connected layers at the end
ElasticTrainer	ResNet-50, VGG16, MobileNetV2 and Vision Transformer [19]

efficient and practical than deploying a randomly initialized model to the device and training from scratch. Figure 1 presents a generic workflow of on-device training.

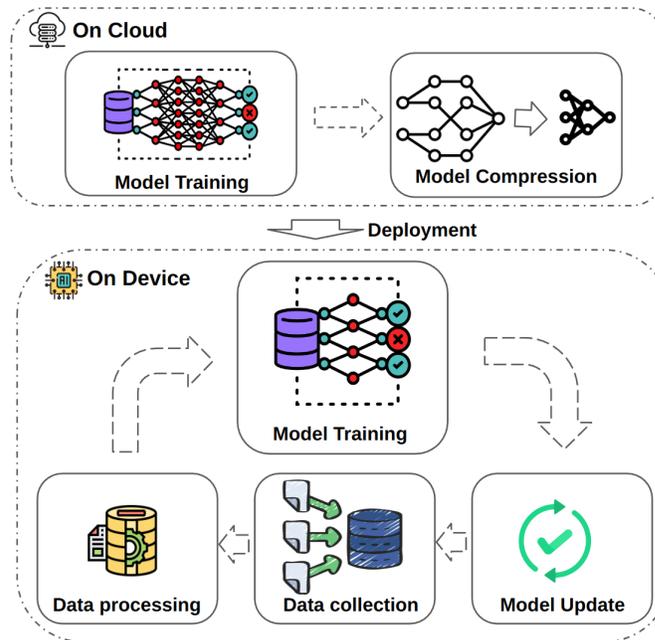


Fig. 1. A generic workflow of on-device training.

3.2.2 Reinforcement Learning. zTT is implementing the on-device training application using DQN, an ML-based variant of Q-learning [124] that employs an NN model to approximate the Q-function. Q-learning is a value-based reinforcement learning algorithm [52], in which the agent takes actions leading to optimal accumulated Q-values. The

paradigm of reinforcement learning (RL) differs from that of standard supervised learning. Supervised learning requires a dataset with ground truths, and data samples are generally assumed to be independent and identically distributed (i.i.d.). However, in RL, the learning process depends on the interaction between the agent and the environment without ground truths. Furthermore, data samples are not i.i.d. because previous outputs influence future inputs. Specifically, during the RL learning process, the agent takes an Action based on the State of the environment and receives a Reward from the environment after taking the Action; the Action, in turn, affects the State of the environment, and the agent adjusts the policy according to the Reward. The learning process does not necessitate ground truth (or labels), which is a core advantage of RL.

In zTT, the authors exploit a pre-trained, fully connected NN model to estimate the Q-function. The training process follows three steps. Firstly, the DQN agent takes action, calculates the reward, and observes the resulting next state. Secondly, it collects samples and puts them into the replay memory until termination. Samples consist of state, action, reward, and next state. Thirdly, random batch sampling from the replay memory and using these samples to train the NN model.

4 Training optimization strategies

To reduce memory consumption, the surveyed works apply or combine different optimization strategies, which can be categorized as (i) memory optimization, (ii) model optimization, and (iii) additional-hardware assistance.

4.1 Memory Optimization

All of the works mentioned above, with the exception of TinyTL, enable on-device training by utilizing various memory optimization techniques. During model training, peak memory usage is significantly higher than during inference, ranging from four to 103 times larger [27, 74, 98], primarily due to a large number of activations, including those of the hidden layers. To avoid out-of-memory issues and improve training throughput, memory optimization techniques are essential for on-device training. Current works mainly employ recomputation, paging, and micro-batch techniques for this purpose.

Recomputation: Recomputation is a memory optimization technique that involves deleting intermediate activations to free up memory and then recomputing them when needed, such as the output of hidden layers and the results of a non-linear function. POET, Melon, and Sage are examples of works that employ recomputation techniques to reduce their peak memory usage during on-device training.

Paging: Paging is a memory management technique widely used in various operating systems. It divides physical memory into fixed-size pages and logical memory of a program into equally sized "page frames". Each page frame is mapped to a physical page. During the program execution, the OS retrieves the needed page frame from secondary storage if it is not in physical memory. Meanwhile, the OS pages out the unneeded page frame to the secondary storage from memory in order to release memory. POET utilizes paging as a memory management technique to reduce peak memory usage by paging out the unneeded intermediate activations into the secondary storage during on-device training.

Micro-batch: Micro-batch is a training optimization technique that involves using smaller batch sizes during training to reduce peak memory usage. However, using a smaller batch size may negatively impact the performance of the model with batch normalization layers. As a result, there is a trade-off between feasibility and accuracy when using this technique. Mandheling, Sage, and Melon are examples of works that utilize micro-batch techniques in their systems for on-device training optimization.

4.2 Model Optimization

In addition to the memory optimization techniques described above, on-device training systems also leverage model-side optimization techniques, with a target to simplify the model to reduce memory consumption and the number of floating number operations (FLOPs). These techniques include quantization, dequantization, mixed-precision, sparse updates, continuous growth (CG), and adjusting the architecture of neural networks, such as replacing or inserting new layers into the network. These techniques are used to further optimize the on-device training process and improve the efficiency and accuracy of the resulting models.

Quantization: Quantization for DL refers to using low-precision formats, such as INT8 and INT16, to represent the parameters of neural network models and intermediate activations, which are normally stored using a floating-point format. The quantization technique quantizes the float format values by multiplying them by a scale factor and rounding them to the nearest integer. Quantization can reduce the memory usage of both inference and training but at the cost of reduced accuracy of the resulting models. Researchers widely use this technique to design efficient ML algorithms. Mandheling, MiniLearn, TTE, and TinyTL are examples of works that apply quantization for on-device training optimization.

Dequantization: Dequantization is the process of converting quantized parameters of models and intermediate activations back to a high-precision format, which is the converse process of quantization. Quantization uses a scaling factor as a multiplier to map floating-point numbers into a smaller range of values, such as integers. Dequantization techniques normally represent the quantized values by dividing the scale factor to recover the original continuous values. By using dequantization, it is possible to mitigate the accuracy loss incurred during quantization, allowing for improved accuracy while still achieving the memory savings of quantization. MiniLearn combines quantization and dequantization techniques to enable on-device training.

Mixed-precision algorithms: Researchers have proposed many mixed-precision algorithms [13, 46, 75, 76] for training, where the weights and intermediate activations are represented not only by FP32 but also by lower precision formats such as INT8 and INT16. The mixed-precision algorithm is the key technique used in Mandheling.

Pruning: Pruning techniques can be broadly classified as network pruning and activation pruning [77]. Network pruning removes less important units of an NN, while activation pruning builds a dynamic sparse computation graph to prune activations during training. MiniLearn uses network pruning as one of its main techniques.

Sparse update: The sparse update refers to updating only a subset of the NN parameters during training while freezing the other parameters. For example, for a CNN model, TinyTL only updates the bias and freezes the parameters of filters. MiniLearn and TinyTL apply this strategy.

Continual Learning: A typical ML pipeline can be simplified into three steps: model definition/selection, model training, and model inference. Various optimization techniques have been proposed to improve efficiency. For example, pruning techniques reduce the model size to enhance inference efficiency, while micro-batch techniques accelerate the training process. Unlike these techniques, continual learning enables DNN models to learn new tasks with a few data samples over a lifetime. As aforementioned, when applying continual learning, researchers need to address the catastrophic forgetting issue. The three common approaches are regularization approaches, dynamic architectures, and memory replay. The continuous growth technique is an example of the dynamic architecture approach. The technique optimizes the pipeline from the initial step. Rather than choosing and training a model with redundant parameters, the continuous growth approach starts with a small neural network and continuously increases its size during training until achieving a balance between accuracy and efficiency.

Adjusting the architecture of NNs: TinyTL and TinyOL enable on-device training by modifying the architecture of NNs. TinyTL modifies the architecture by adding new layers to the network, while TinyOL designs an extra layer and then inserts it into the NN or uses the extra layer to replace one original layer of the NN. During training, both TinyTL and TinyOL only tune the additional layers to achieve high-efficient model training on resource-constrained devices.

4.3 Auxiliary hardware

Traditional model training leverages the GPU for its high parallel computation capability. When performing on-device training, there normally are no powerful GPUs, but auxiliary hardware is still available for workload offload. For example, the DSPs are particularly suitable for integer operations and ubiquitously available on modern SoCs. Mandheling leverages a DSP to enable model training on smartphones.

5 Performance Gains of On-device Training Systems

This section presents the performance gains of the surveyed systems. Overall, researchers use one or several metrics from the following four metrics to measure their systems: accuracy improvement, memory usage reduction, energy consumption reduction, and training acceleration. Section 5.1 summarises the baselines against which the systems compare their performance. Section 5.2 describes the accuracy improvement brought by these on-device training methods. One of the goals of on-device training is to keep improving the accuracy of the deployed model. Therefore, this is the key outcome of on-device training. Section 5.3 presents the optimized memory usage of the different systems. Limited memory is the main bottleneck of on-device training systems. Hence, this is one of the most important metrics for evaluating the system. Section 5.4 shows the energy consumption of model training, which is another concern for IoT devices that are normally battery-powered. Section 5.5 describes the training speed improvement, which is another important evaluation metric for on-device training. The training tasks are expected to be completed within a reasonable time. Table 4 summarizes the performance gain of the discussed on-device training systems.

5.1 Baselines

The baselines used for evaluation in current works mainly consist of four categories: Firstly, the ideal baseline, or cloud baseline. This baseline represents the situation with sufficient hardware resources, such as training models on the cloud or powerful servers, where sufficient memory and computing resources are provided. This baseline provides the best training accuracy and the corresponding resource consumption, such as memory. With this baseline, researchers show the potential accuracy gap and the reduction in resource consumption between on-device training and the ideal case. Secondly, training models without any optimization techniques or using pre-trained models without any further training. This baseline is used to show the performance gain of the proposed method, for example, accuracy improvement and memory consumption reduction. Thirdly, existing model training optimization techniques and frameworks, this type of baseline is used to show the improvements of the proposed on-device training schemes, such as TensorFlow Lite [72] and MNN [50], two efficient and lightweight DL frameworks that are commonly used for on-device training. Fourthly, baselines that only use one single optimization technique used in the system design. This aims to evaluate the impact of individual techniques on the results.

5.2 Accuracy Improvement

A primary objective of on-device training is to uphold or even enhance the accuracy of deployed models while operating under computational constraints. Consequently, accuracy improvement stands as a crucial performance metric for

on-device training. Existing studies assess the accuracy gains achieved by on-device training systems across a spectrum of representative ML tasks encompassing computer vision (CV), audio recognition, and human activity recognition (HAR). Despite the utilization of diverse neural network architectures and benchmark datasets, we noticed that on-device training can contribute to accuracy enhancements. While the magnitude of accuracy gains may vary depending on the task, the findings collectively demonstrate the potential to achieve accuracy improvements without reliance on external servers or internet connectivity while also mitigating some critical privacy risks. Furthermore, a notable advantage of on-device training is its capability to address the data drift problem. Data drift is a prominent challenge in ML systems after deployment [116], which refers to the phenomenon where the distribution of the features changes after training, leading to decrement in the accuracy of models.

Computer Vision (CV): CV stands as a foundational domain within ML, underpinning critical advancements spanning from autonomous vehicles to automated diagnostics, thus establishing itself as one of the most impactful areas within the field. Evaluated CV tasks encompass image classification, object detection, and human facial attribute classification. These evaluations employ a diverse array of neural network architectures, including transformers and convolutional neural networks (CNNs), such as ImageNet [15], MobileNetV1 [38], MobileNetV2, SqueezeNet [43], ResNet-50 and Vision Transformer [19]. Correspondingly, a wide spectrum of benchmark datasets is utilized, including Cars [59], Flowers [91], Aircraft [81], CUB [119], Pets [93], Food [5], CIFAR10 [60], CIFAR100 [60], VGGFace2 [8], MiniImageNet [118], TinyImageNet [67], ImageNet1k [105], Stanford Dogs [54] and CelebA [78].

In terms of accuracy enhancement, MiniLearn adopts a strategy of on-device model retraining with 600 data samples, resulting in significant accuracy improvements compared to pre-trained models and achieving comparable accuracy levels with the ideal baseline. Moreover, MiniLearn demonstrates the capability for a pruned model (with pruning percentages up to 75%) to outperform pre-trained models and attain commendable accuracy levels with 400 to 500 samples. TinyTL, when applied to ImageNet models, achieves substantial accuracy improvements exceeding 30% on benchmark datasets such as Cars and Aircraft, over 10% on CIFAR100, Food, and CIFAR10, and enhancements ranging from 0.5% to 5.4% on datasets including Flowers, CUB, CelebA, and Pets. Furthermore, Melon conducts model training for other prominent CV architectures, including MobileNetV1, MobileNetV2, SqueezeNet, and ResNet-50, in both centralized and federated scenarios. In centralized settings, Melon achieves accuracy gains of 1.98% and 2.04% on MobileNetV2 and SqueezeNet, respectively. In federated scenarios, Melon achieves accuracy improvements of 3.94% and 3.20% over the baseline. For LifeLearner, the system achieves near-optimal accuracy of continual learning. In continual learning, the oracle baseline represents the optimal performance, because it can access all the classes in an i.i.d. manner and train the DNN model as many epochs as possible until converge. Compared to the Oracle baseline, LifeLearner only loses 0.2% accuracy for CIFAR-100 dataset and 2.7% accuracy for MiniImageNet dataset. In contrast, the previous SOTA Meta-CL method shows a 9.9% accuracy drop for CIFAR-100 and 10.7% for MiniImageNet compared to the Oracle baseline. Miro achieves 1.35–15.37% higher accuracy across various memory budgets compared to the baseline systems with small to medium-scale datasets, and achieves 23.37% higher accuracy compared to a SOTA HEM-based system CarM [70] when using a larger dataset (ImageNet1k).

Audio Recognition: Audio recognition holds substantial importance within the ML domain, serving as a fundamental component in various sound-based intelligent applications, including voice-activated assistants. Harnessing the capabilities of on-device training, MiniLearn achieves significant accuracy gains in the Google Keyword Spotting (KWS) benchmark dataset [123], reaching comparable levels to the ideal baseline. LifeLearner shows a 5.6% accuracy drop compared to the Oracle baseline for Google Speech Command V2 (GSCv2) [11], while the previous SOTA Meta-CL method only reveals a 0.2% accuracy reduction. However, the previous SOTA Meta-CL method is not suitable for

resource-constrained devices. LifeLearner is an on-device continuous learning system that requires much lower system resources. Hence, the difference is acceptable.

Human Activity Recognition (HAR): HAR plays an important role in enabling smart devices to intuitively comprehend and respond to human movements, thereby facilitating advancements in health monitoring, fitness tracking, and home automation. By enhancing technology’s responsiveness and personalization to human activities, HAR contributes to improving the overall user experience. In a manner akin to audio recognition, MiniLearn leverages on-device training techniques to continually refine CNN models on-device, ultimately achieving accuracy levels on par with the ideal baseline in the WISDM-HAR benchmark dataset [64].

5.3 Memory Usage Reduction

Memory scarcity poses a significant bottleneck in on-device training of DL models, particularly for high-constraint devices, which typically possess only kilobyte-level memory. While middle-constraint and low-constraint devices offer larger memory capacities compared to their high-constraint counterparts, the imperative to reduce memory usage remains paramount in training DNN models. Even server-scale GPUs, with their substantial memory resources, are grappling with escalating memory requirements for model training. Our review indicates the feasibility of on-device training across all device types. On-device training systems facilitate on-device training by adapting and refining well-established techniques utilized in server-based training, including micro-batching [42] and recomputation [10], alongside general memory management strategies like paging [115]. Furthermore, for high-constraint devices, general efficient ML techniques such as pruning [90] and quantization [46] play a pivotal role in memory optimization. Collectively, the diverse array of techniques aimed at enhancing memory efficiency holds immense potential in advancing on-device training capabilities.

High-constraint Devices: Several recent works, including MiniLearn, TTE, and POET, have demonstrated the feasibility of model training on high-constraint devices. In these studies, researchers either train simple NNs with fewer than ten layers or employ fine-tuning and transfer learning techniques to train more complex models. This approach is justified, as high-constrained devices lack the resources to support training from scratch.

For training simple NN models, MiniLearn conducts separate evaluations of Flash and RAM consumption. When retraining with 100 samples, MiniLearn achieves reductions of up to 58%, 77%, and 94% in Flash consumption for KWS, CIFAR, and WISDM-HAR, respectively, compared to cloud-based training. With 600 samples, Flash consumption reduction rates are 8% for KWS, 17% for CIFAR, and 80% for WISDM-HAR. Regarding RAM consumption, retraining without pruning consumes 196KB, 128KB, and 92KB on KWS, CIFAR, and WISDM-HAR, respectively. Fine-tuning only the fully connected layers reduces RAM consumption to 12KB, 10KB, and 6KB on KWS, CIFAR, and WISDM-HAR.

For complex model training, current systems employ fine-tuning to facilitate on-device training. In TTE, MobileNetV2, ProxylessNAs, and MCUNet are evaluated, using datasets such as Cars, CIFAR-10, CIFAR-100, CUB, Flowers, Food, and Pets. Peak memory usage is compared across three settings: fine-tuning the full model, sparse updates-only, and sparse updates with TTE graph reordering. Sparse updates effectively reduce peak memory usage by 7-9× compared to full fine-tuning while maintaining or exceeding transfer learning accuracy. Incorporating TTE graph reordering achieves total memory savings of up to 20-21×. On the other hand, POET evaluates memory consumption for one training epoch on VGG16, ResNet-18, and BERT. Compared to the baseline POFO, POET reduces memory usage by 8.3% and improves throughput by 13%.

Middle-constraint Devices: Middle-constraint devices typically boast memory capacities ranging from megabytes to gigabytes but lack a dedicated GPU. Fine-tuning and transfer learning serve as viable strategies for enabling

on-device training on these devices. For instance, TinyTL achieves memory savings of less than 20% compared to the baseline by fine-tuning the normalization layers and the last layer, while delivering up to 7.2% higher accuracy. Additionally, compared to fine-tuning the entire neural network, TinyTL achieves a 6× reduction in training memory while maintaining the same level of accuracy.

Low-constraint Devices: Low-constraint devices are typically robust devices equipped with several gigabytes of memory and GPUs. However, efficient memory management remains essential for training on such devices. In the case of Melon, the authors assess the impact of memory budget adaptation by training MobileNet V2 and SqueezeNet on a Samsung Note 10, with memory varying from 6GB to 5GB and from 4GB to 3GB, respectively. Melon achieves significant overhead memory savings ranging from 4.27% to 54.5% compared to a simple stop-restart approach, where the entire memory pool is reallocated, and all intermediate activations are discarded upon changes in memory budget. Another system, Sage, demonstrates memory usage reductions of up to 50% compared to the DGA-only baseline [112] when training BERT-small [16], ResNet-50, DenseNet-121 [40], and MobileNetV2 on smartphones. Additionally, Sage achieves peak memory consumption reductions of up to 420% compared to the DGC-only baseline [58].

In the domain of continual learning systems, MDLdroidLite is evaluated by training LeNet, MobileNet, and VGG-11 models on four Personal Mobile Sensing (PMS) datasets (sEMG, MHEALTH, HAR, and FinDroidHR) and two image datasets (MNIST and CIFAR-10). While direct measurement of memory consumption reduction is not conducted, the evaluation focuses on model size reduction, which inherently leads to reduced memory consumption. Specifically, MDLdroidLite achieves parameter reductions of 28× to 50× and FLOPS reductions of 4× to 10× when training LeNet on PMS datasets. For MobileNet training, parameter and FLOPS reductions of 4× to 7× and 2× to 7×, respectively, are observed. Although training a large-scale VGG-11 on-device proves costly and leads to battery drain, MDLdroidLite successfully trains a backbone VGG-11 model to achieve over 75% accuracy with a minimal battery consumption of 1328mAh.

5.4 Energy Consumption Reduction

Minimizing energy consumption during ML model training is essential for fostering sustainable AI development, mitigating environmental impact, and cutting operational costs [1]. This concern is particularly pertinent in the context of on-device training, where many devices rely on battery power and cannot sustain intensive energy-consuming tasks. Energy usage in model training primarily stems from two sources: computation and memory access. While advancements in processor technology enhance computing capabilities, they also exacerbate energy consumption issues. Similarly, while parallelism boosts computing performance, it also escalates energy consumption. Furthermore, memory access operations are significant contributors to energy consumption [37]. On-device training systems address these challenges by optimizing both computation and memory access during model training. For instance, Sage [27] achieves notable energy consumption reductions of up to 49.43% compared to baselines, thanks to efficient memory management practices, all while maintaining or even improving accuracy.

High-constraint devices: High-constraint devices face limitations in computational resources and battery life, making faster training essential for optimizing their scarce resources. In MiniLearn, the authors conduct experiments across each use case 30 times, varying the number of data samples from 100 to 600. As expected, higher sample counts lead to increased energy consumption. For instance, with 100 samples, model retraining consumes $254 \pm 0.2mJ$, $203 \pm 0.2mJ$, and $138 \pm 0.2mJ$ on KWS, CIFAR, and WISDM-HAR, respectively, while with 600 samples, these figures rise to $1486 \pm 0.1mJ$, $1016 \pm 0.2mJ$, and $786 \pm 0.1mJ$. MiniLearn also demonstrates energy savings in communication compared to an ideal baseline, which involves uploading training samples to the cloud and downloading the retrained model. For instance,

with 600 samples and BLE communications, energy consumption amounts to 45 mJ, 462 mJ, and 210 mJ for WISDM, CIFAR, and KWS, respectively.

Middle-constraint devices: Middle-constraint devices face similar energy consumption challenges as high-constraint devices. POET, an on-device training system designed for middle-constraint devices, demonstrates lower energy consumption compared to baselines. For instance, when training ResNet-18 on the Raspberry Pi 4, POET consumes only 41% to 58% of the energy compared to the Capuchin baseline [95]. Compared to solutions relying solely on paging or rematerialization, POET achieves energy savings of up to 40%.

Low-constraint devices: Low-constraint devices, such as smartphones, are expected to maintain usability over extended periods between charging cycles. Consequently, model training should not excessively drain battery life, potentially disrupting user experience by, for example, causing missed calls due to battery depletion.

Two-stage systems, Melon and Sage, are tailored for smartphone environments. Melon’s evaluation involves training two neural networks, MobileNetV2 and SqueezeNet with a Batch Normalization layer, on a Meizu 16t smartphone. Melon consistently outperforms baselines, achieving energy savings ranging from 22% to 49.43%. In comparison to the ideal baseline, Melon exhibits an average energy consumption of only 11.4% and as low as 2.1% in the best-case scenario. Sage, on the other hand, is assessed by measuring energy usage per training iteration of ResNet-50 on a Samsung Galaxy S10. With each iteration consuming approximately 4.9J, a 1000-iteration training session only utilizes 8% of the device’s entire battery capacity, suggesting Sage as a suitable on-device training system for mobile devices. In addition, ElasticTrainer also shows high energy efficiency. In contrast to training full DNN, ElasticTrainer achieves up to three-fold energy consumption reduction without noticeable accuracy loss.

For continual learning systems, the energy consumption evaluations of Mandheling span across various NN architectures, including VGG-11/16/19, ResNet-18/34, and Inception V3. Mandheling consistently outperforms baselines in per-batch training scenarios, achieving significant energy consumption reductions compared to competitors such as MNN-FP32 and TFlite-FP32. Notably, Mandheling demonstrates energy consumption reductions ranging from 3.21 to 11.2 times compared to MNN-FP32 and 2.01 to 12.5 times compared to TFlite-FP32. Additionally, in convergence scenarios, Mandheling exhibits substantial energy consumption reductions of 5.96 to 9.62 times in single-device settings and even greater reductions in distributed scenarios, reaching up to 13.1 times for a single client on datasets like FEMNIST and CIFAR-100. For LifeLearner, compared to the previous SOTA Meta-CL method, the system reduces its energy consumption over 80.9% and 94.2%, for the CIFAR-100 and GSCv2 datasets on NVIDIA Jetson Nano, and 92% and 96% on Raspberry Pi 3+, respectively. For Miro, the authors report that its energy cost is always the lowest compared to baselines across benchmarks and memory budgets.

Finally, the reinforcement learning-based system zTT is evaluated on both Google Pixel 3a and NVIDIA JETSON TX2 platforms, covering a range of applications, including Aquarium, YOLO, Video rendering, Showroom VR, Skype, and Call of Duty 4. Compared to baseline methods such as Maestro and default DVFS methods, zTT demonstrates superior power consumption management while maintaining strict performance guarantees. Specifically, when utilizing JETSON TX2, zTT achieves a reduction in power consumption of 37.4% and 23.9% for FPS rates of 20 and 30, respectively, compared to the default method. Furthermore, zTT consistently meets target FPS requirements, a feat unattainable by the default method or Maestro. Although none of the techniques reach 60 FPS on Pixel 3a, zTT performs closest to the target FPS among the evaluated methods.

5.5 Training Acceleration

Accelerating the training speed of ML models is crucial for enabling quicker experimentation and reducing both computational costs and energy use. Low-constraint devices, such as smartphones, are expected to run multiple processes to provide a good user experience. Meanwhile, they feature limited computational power and battery life. Quick model training boosts user experience by supporting online model updates and instant decision-making, independent of other functionalities. Training acceleration is another gain from on-device training. With efficiency optimization, on-device training systems markedly improve the training speed compared to baselines. In particular, Mandheling [125] increases the converge speed over 10 \times .

Low-constraint devices: Mandheling, Melon, MDLdroidLite, and ElasticTrainer involve the assessment of training acceleration. In Mandheling, the authors measure the training time of Mandheling on VGG-11 [111], ResNet-18 [33], LetNet [68] and InceptionV3 [114]. Baselines are based on MNN [50] and TFLite [72]. Concretely, four baselines are used in the evaluation. (1) MNN-FP32: the Floating Point 32-bit-based training method of MNN. (2) MNN-INT: the Integer 8-bit-based training method based on MNN. (3) MNN-FP32-GPU: the Floating Point 32-bit-based training method on mobile GPU through the OpenCL backend. (4) TFLite-FP32: the Floating Point 32-bit-based training method provided by TFLite. In the single-device scenario, compared to MNN-FP32, Mandheling takes between 5.05 \times to 6.27 \times less time for convergence with a small loss in accuracy (1.9% - 2.7%). In contrast to MNN-INT8, Mandheling reaches the same accuracy and spends 3.55 \times less time. In a federated learning (FL) scenario, the authors measure Mandheling with training datasets FEMNIST [99] and CIFAR-100. In addition, they apply an FL simulation platform [130] and two baselines that use the traditional FL protocol FedAvg [84]: FloatFL and Int8FL. Mandheling uses 5.25 \times and 10.75 \times less time to converge on FEMNIST and CIFAR-100, respectively.

For Melon, the authors measure the convergence speed with training throughput, which is defined as Equation 6. The throughput represents the number of data samples trained every second. The higher the throughput of the system, the higher the convergence speed. Overall, Melon always outperforms other baselines except for the ideal baseline, and it often achieves a similar throughput as the ideal baseline. In particular, the authors respectively evaluate Melon on NN with and without BN layers. For training NN without BN layers, Melon achieves 1.51 \times - 3.49 \times higher throughput than vDNN [101]. In contrast to Sublinear [10], Melon reaches 13 \times - 3.86 \times higher throughput. And it also achieves 1.01 \times - 4.01 \times higher throughput than the Capuchin baseline. For NN with BN layers, Melon almost reaches the same performance as the ideal baseline (below 1%) and significantly outperforms the other baselines.

$$throughput = \frac{BatchSize}{PerBatchLatency} \quad (6)$$

For MDLdroidLite, the authors compare its performance against three SOTA parameter adaption methods in CG: NeST-bridge, CGaP-select, and Net2WiderNet. Each experiment is conducted five times with training spanning 30 epochs. MDLdroidLite demonstrates notable speed improvements, achieving an average acceleration of 4.88 \times , 2.84 \times , and 3.12 \times compared to NeST-bridge, CGaP-select, and Net2WiderNet, respectively. Moreover, MDLdroidLite maintains superior convergence accuracy stability in contrast to the evaluated baselines.

For ElasticTrainer, when using the ResNet50 model and $\rho = 50\%$ (2 times training speedup), ElasticTrainer can achieve similar accuracy in contrast to training the full model by training a much smaller model portion on both Jetson TX2 and Raspberry Pi 4B. Specifically, on the Pets and Stanford Dogs dataset, ElasticTrainer even achieves 1%-2% higher accuracy than training the full model. The reason is that training a smaller trainable DNN portion leads to less overfitting than training the full model. Comparatively, compared to training the full model, only training the last

prediction module loses over 20% accuracy on the CUB dataset, and training the last prediction module together with bias parameters and batch normalization layers loses over 10% accuracy. Specifically, on Jetson TX2, ElasticTrainer can achieve a 2-3.4 \times speedup compared to training the full model. Its training speedup also outperforms up to 30% compared to training the last prediction module, bias parameters, and batch normalization layers. Although only training the final prediction module brings 20% more speedup, it provides the lowest accuracy.

Middle-constraint devices: The authors of ElasticTrainer also evaluated their system on a middle-constraint device, i.e., Raspberry Pi 4B. Generally, as on Jetson TX2, ElasticTrainer achieves an accuracy similar to training the full model with less training time. Especially, ElasticTrainer outperforms training the full model on the Pets dataset by achieving 1%-2% higher accuracy. In contrast, only training the final prediction module, bias parameters, and batch normalization layers provides 25% training speedup at the cost of 30% accuracy loss.

High-constraint Devices: For MiniLearn and TTE, the authors evaluate the training speed of their systems. In MiniLearn, the authors repeat experiments on each use case 30 times and report the average training time using MiniLearn with a different number of data samples, from 100 to 600 samples. The more samples used, the more time cost. For example, when using 100 samples to retrain models, the training tasks $10 \pm 0.2s$, $8 \pm 0.2s$, and $6 \pm 0.2s$ on KWS, CIFAR, and WISDM-HAR, respectively, while the time prolongs to $56 \pm 0.1s$, $40 \pm 0.2s$ and $30 \pm 0.3s$ when using 600 samples. Meanwhile, for TTE, the authors contrast three methods: fine-tuning the full model with TFLite Micro, sparse updates with TFLite Micro kernels, and sparse updates with TTE kernels. The sparse updates plus TTE kernels method increases the training speed by 23 \times - 25 \times times significantly compared to the sparse update with the TF-Lite Micro method.

6 Summary of State-of-the-art Solutions

In this section, we elaborate on the methods and ideas presented in the current SOTA systems for on-device training (Section 2). These works target and address systematic issues raised by devices' resource limitations via different approaches or dimensions. Overall, the core mechanisms presented in SOTA works can be summarized in the following four categories.

Run-time optimization: This approach refers to applying various techniques to reduce resource consumption during training, which consists of two directions. The first one is using memory consumption optimization techniques to reduce peak memory consumption. Researchers exploring this direction of research conclude that memory is the main bottleneck for on-device training due to backpropagation causing a large number of intermediate variables. Hence, several works, including POET, Melon, and Sage incorporate memory optimization techniques, such as paging and materialization (Section 2.5). The second direction is to reduce resource consumption from the model perspective. For example, model quantization techniques can significantly reduce the size of the model and, as a result, decrease the resource requirement. However, quantized models naively are more challenging to train due to precision reduction (e.g., from float32 to int8). Minilearn and TTE, respectively, present solutions for this problem to allow for effective on-device training (Section 2.4 and Section 2.5). In addition, selectively training the parameters of the model can also reduce the training time and resource consumption. Unfortunately, this leads to limited generalization ability. TinyTL addresses this issue by designing a lighter residual model (Section 2.4). Overall, run-time optimization is the most common strategy for on-device training. It helps on-device training systems achieve efficient memory usage (Section 5.3), which represents one of the most significant challenges of on-device training. In addition, it can also reduce computational complexity with optimization techniques from the model perspective, such as model quantization techniques.

Table 4. Performance Gains for the publications discussed in this work.

Systems	Accuracy Improvement	Memory Usage	Energy Consumption	Training Speed
MiniLearn [97]	Increases accuracy for a sub-set by 3% to 9% of the original DNN.	Reduction of up to 50% memory compared to the original DNN.	Consumes 138-254 mJ (100 training samples), 786-1016 mJ (600 training samples).	Converges in 6s - 10s (100 training samples), 30s - 56s (600 training samples).
Sage [27]	-	Reduces memory use by more than 20-fold compared to a baseline.	For each iteration, Sage consumes around 4.9J.	-
MDLdroidLite [134]	-	Achieves 28× to 50× parameter reduction and 4× to 10× FLOPS reduction.	-	Speed up training by up to 4.88×.
Melon [121]	Accuracy improvement, in centralized training, 2.04%, in federated training, 3.94%, compared to MNN [50].	Achieves up to 4.33× larger batch size under the same memory budget.	Saves up to 49.43% energy compared to baseline.	Achieves 1.89× on average (up to 4.01×) higher training throughput.
Mandheling [125]	-	-	Consumes 5.96 – 9.62× less energy for convergence.	Takes up to 6.27× less time for convergence.
TinyTL [6]	Accuracy improvement up to 34.1%.	Saves the memory up to 6.5×.	-	-
TTE [74]	-	Reduces peak memory usage by 20× - 21×.	-	Increases the speed by 23× - 25×.
POET [94]	-	Reduces memory usage by 8.3%.	Saves up to 59% energy compared to baselines.	Improves the throughput by 13%.
zTT [55]	-	-	23.9% less average power consumption.	-
LifeLearner [65]	Accuracy improvement up to 8% compared to the SOTA.	Saves the memory up to 178.7× compared to the SOTA.	Saves the energy up to 96% compared to the SOTA.	-
ElasticTrainer [41]	Accuracy improvement 1%-2% compared to full training.	Consumes 10% more memory than full training.	Saves the energy up to 3× compared to full training.	Increases the speed by 3.4× compared to full training.

Workload offloading: Leveraging other available hardware is another possibility. For example, the authors of Mandheling observe that DSPs are particularly suitable for integer operation and ubiquitously available on modern smartphones. Hence, they apply CPU-DSP co-scheduling with a mixed-precision training algorithm to enable on-device training (Section 2.5.2). Even though auxiliary hardware is not commonly available for high-constraint devices, systems designed for low-constraint devices, such as smartphones, can gain both energy consumption reduction and training acceleration by offloading workload to auxiliary hardware (Section 5.4 and Section 5.5).

Continuous growth: This approach tries to reduce resource consumption from the very beginning with the understanding that massive resource consumption is caused by the large size of NN models. Given a target NN, Frankle et al. [25] show that traditional DNN models often have a large number of redundant parameters that can be reduced with little accuracy deduction. Based on this observation, MDLdroidLite starts from training a tiny NN and gradually grows the architecture until the accuracy satisfies the application’s requirements (Section 2.6). Continuous growth significantly reduces the number of parameters by up to 50 times and FLOPs by up to 10 times. In addition, it speeds up training by up to 4.88 times (Section 5.3 and Section 5.5). Hence, this is one feasible mechanism for enabling on-device training.

Reinforcement learning: RL is another approach to enable on-device training, where an agent interacts with an environment to learn a behavior or task through trial and error. The agent in the system learns to maximize the cumulative reward provided by the environment by selecting different possible actions. This paradigm implies continuous learning during runtime and does not require newly collected and labeled data. For example, zTT applies RL to design and implement a learning-based DVFS solution on mobile devices (Section 2.7).

As discussed above, from the ML perspective, current systems cover ML model-side optimization, training time optimization, and choosing ML methods with suitable paradigms (e.g., reinforcement learning). From the device side, most system designers configure their target systems using devices with sufficient computational capability (e.g., smartphones), which offers them flexibility in system design (e.g., GB-level memory and dedicated hardware such as DSP), as shown in Table 1. An additional benefit of using such commercially available (relatively) powerful platforms is the fact that they operate on accessible OSes (e.g., Android OS), providing a friendly development environment to migrate and distribute the scheme to many devices. Therefore, one possible research direction is to combine different approaches to achieve a more efficient hybrid solution. For example, one can combine continuous growth or RL and optimized memory management techniques to enable on-device training. Given that many of these approaches are orthogonal to each other and can be used together, understanding and combining the benefits of each approach can be an attractive future research direction.

7 Conclusion and Future Directions

In this work, we present a survey of existing systems for on-device training. Although on-device learning is still an emerging field of research, a number of systems have been introduced to support general on-device training for mobile and IoT platforms. Mobile platforms play a crucial role in people’s daily lives and have sufficient hardware capabilities to apply various optimization techniques for model training, including several gigabytes of memory, powerful CPUs, and hardware accelerators such as GPUs, NPUs, and DSPs. On-device training, therefore, can facilitate the development of intelligent applications for smartphones and migrate the integration from the cloud infrastructure to the devices themselves. On the other hand, for many IoT platforms, which are microcontroller-based and hold tighter resource limitations, model training becomes more challenging due to their extreme resource constraints. Hence, on-device training systems for microcontroller platforms typically require both model-side and resource-consumption

optimizations. Overall, on-device learning allows various devices to locally leverage the power of ML, and we anticipate it to become an important aspect of many applications for industry and society.

At the same time, our survey also leaves us with a list of future research directions, that are yet to be deeply explored. Specifically, fewer efforts have been made for enabling on-device model training for highly constrained IoT devices, such as those that feature only a couple hundred MBs of memory and clock speeds in the sub-GHz range. Despite their significance and adaptation in designing a wide range of smart applications, the lack of systems tailored for such platforms is mainly due to two reasons. Firstly, such constrained hardware gives less flexibility for system design. Secondly, there is no one-size-fits-all OS for these IoT devices; thus, requiring per-system/implementation optimization. Designing a common framework that enables per-platform customized on-device learning for resource-constrained IoT platforms can be an interesting direction for future research. Potentially, we can think of approaches such as (i) exploring the possibility of applying existing optimized techniques on tiny devices, (ii) designing specific efficient NN for tiny devices with considerations on both inference and training efficiency, and (iii) integrating ML features, both inference and training, on the OS level to ease future development and migration. Future research can be initiated by exploiting mature embedded OSes, such as Contiki [21], TinyOS [71], FreeRTOS [2], or Android Things [104].

Acknowledgements

This work was partially supported by the Swedish Science Foundation (SSF), Korean Ministry of Science and ICT (MSIT) and the IITP under grants IITP-2024-2020-0-01461 (ITRC Program) and IITP-2022-0-00420.

References

- [1] Jiafu An, Wenzhi Ding, and Chen Lin. 2023. ChatGPT: tackle the growing carbon footprint of generative AI. *Nature* 615, 7953 (2023), 586–586.
- [2] Richard Barry et al. 2008. FreeRTOS. *Internet, Oct 4* (2008).
- [3] Shawn Beaulieu, Lapo Frati, Thomas Miconi, Joel Lehman, Kenneth O Stanley, Jeff Clune, and Nick Cheney. 2020. Learning to continually learn. In *ECAI 2020*. IOS Press, 992–1001.
- [4] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanhao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. 2022. Ekyra: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 119–135.
- [5] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. 2014. Food-101—mining discriminative components with random forests. In *Computer Vision—ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6–12, 2014, Proceedings, Part VI 13*. Springer, 446–461.
- [6] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. 2020. Tinyt: Reduce memory, not parameters for efficient on-device learning. *Advances in Neural Information Processing Systems* 33 (2020), 11285–11297.
- [7] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).
- [8] Qiong Cao, Li Shen, Weidi Xie, Omkar M Parkhi, and Andrew Zisserman. 2018. Vggface2: A dataset for recognising faces across pose and age. In *2018 13th IEEE international conference on automatic face & gesture recognition (FG 2018)*. IEEE, 67–74.
- [9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
- [10] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).
- [11] Speech Commands. 1804. A Dataset for Limited-Vocabulary Speech Recognition. URL: <https://arxiv.org/abs/1804.03209> (cs: 28.12. 2020) (1804).
- [12] NVIDIA Corporation. 2024. NVIDIA Jetson Platform. <https://developer.nvidia.com/embedded-computing>. [Computer hardware].
- [13] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems* 28 (2015).
- [14] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. 2019. NeST: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Trans. Comput.* 68, 10 (2019), 1487–1497.
- [15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

- [16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [17] Sauprik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. 2021. A survey of on-device machine learning: An algorithms and learning theory perspective. *ACM Transactions on Internet of Things* 2, 3 (2021), 1–49.
- [18] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. [n. d.]. A deep convolutional activation feature for generic visual recognition. *UC Berkeley & ICSI, Berkeley, CA, USA* 1, 2 ([n. d.]).
- [19] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).
- [20] Xiaocong Du, Zheng Li, and Yu Cao. 2019. CGaP: Continuous growth and pruning for efficient deep learning. *arXiv preprint arXiv:1905.11533* (2019).
- [21] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki—a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*. IEEE, 455–462.
- [22] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.
- [23] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. 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 (New Delhi, India) (MobiCom '18)*. Association for Computing Machinery, New York, NY, USA, 115–127. <https://doi.org/10.1145/3241539.3241559>
- [24] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).
- [25] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635* (2018).
- [26] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.
- [27] In Gim and JeongGil Ko. 2022. Memory-Efficient DNN Training on Mobile Devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (Portland, Oregon) (MobiSys '22)*. Association for Computing Machinery, New York, NY, USA, 464–476. <https://doi.org/10.1145/3498361.3539765>
- [28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.
- [29] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129, 6 (2021), 1789–1819.
- [30] Andreas Griewank and Andrea Walther. 2000. Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Trans. Math. Softw.* 26, 1 (mar 2000), 19–45. <https://doi.org/10.1145/347837.347846>
- [31] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. 2016. Continuous deep q-learning with model-based acceleration. In *International conference on machine learning*. PMLR, 2829–2838.
- [32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*. 1026–1034.
- [33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- [34] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*. 1389–1397.
- [35] Geoffrey E Hinton and David C Plaut. 1987. Using fast weights to deblur old memories. In *Proceedings of the ninth annual conference of the Cognitive Science Society*. 177–186.
- [36] Geoffrey E Hinton and Richard Zemel. 1993. Autoencoders, minimum description length and Helmholtz free energy. *Advances in neural information processing systems* 6 (1993).
- [37] Mark Horowitz. 2014. 1.1 Computing’s energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14. <https://doi.org/10.1109/ISSCC.2014.6757323>
- [38] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
- [39] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355.
- [40] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.
- [41] Kai Huang, Boyuan Yang, and Wei Gao. 2023. Elastictrainer: Speeding up on-device training with runtime elastic tensor selection. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*. 56–69.
- [42] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

- [43] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).
- [44] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR abs/1502.03167* (2015). arXiv:1502.03167 <http://arxiv.org/abs/1502.03167>
- [45] Ozan Irsoy and Ethem Alpaydin. 2019. Continuously constructive deep neural networks. *IEEE transactions on neural networks and learning systems* 31, 4 (2019), 1124–1133.
- [46] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2704–2713.
- [47] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems 2* (2020), 497–511.
- [48] Khurram Javed and Martha White. 2019. Meta-learning representations for continual learning. *Advances in neural information processing systems* 32 (2019).
- [49] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.
- [50] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. 2020. Mnn: A universal and efficient inference engine. *Proceedings of Machine Learning and Systems 2* (2020), 1–13.
- [51] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.
- [52] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.
- [53] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. 2022. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems 4* (2022), 172–189.
- [54] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Fei-Fei Li. 2011. Novel dataset for fine-grained image categorization: Stanford dogs. In *Proc. CVPR workshop on fine-grained visual categorization (FGVC)*, Vol. 2.
- [55] Seyeon Kim, Kyungmin Bin, Sangtae Ha, Kyunghan Lee, and Song Chong. 2021. ZTT: Learning-Based DVFS with Zero Thermal Throttling for Mobile Devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (Virtual Event, Wisconsin) (MobiSys '21)*. Association for Computing Machinery, New York, NY, USA, 41–53. <https://doi.org/10.1145/3458864.3468161>
- [56] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. uLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019 (Dresden, Germany) (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 45, 15 pages. <https://doi.org/10.1145/3302424.3303950>
- [57] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [58] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616* (2020).
- [59] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 2013. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE international conference on computer vision workshops*. 554–561.
- [60] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).
- [61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>
- [62] Dharshan Kumaran, Demis Hassabis, and James L McClelland. 2016. What learning systems do intelligent agents need? Complementary learning systems theory updated. *Trends in cognitive sciences* 20, 7 (2016), 512–534.
- [63] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. 2019. A graph theoretic framework of recomputation algorithms for memory-efficient backpropagation. *Advances in Neural Information Processing Systems* 32 (2019).
- [64] Jennifer R Kwapisz, Gary M Weiss, and Samuel A Moore. 2011. Activity recognition using cell phone accelerometers. *ACM SigKDD Explorations Newsletter* 12, 2 (2011), 74–82.
- [65] Young D Kwon, Jagmohan Chauhan, Hong Jia, Stylianos I Venieris, and Cecilia Mascolo. 2023. LifeLearner: Hardware-Aware Meta Continual Learning System for Embedded Computing Platforms. In *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*. 138–151.
- [66] Stefanos Laskaridis, Stylianos I. Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D. Lane. 2020. SPINN: Synergistic Progressive Inference of Neural Networks over Device and Cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking (London, United Kingdom) (MobiCom '20)*. Association for Computing Machinery, New York, NY, USA, Article 37, 15 pages. <https://doi.org/10.1145/3372224.3419194>
- [67] Ya Le and Xuan Yang. 2015. Tiny imagenet visual recognition challenge. *CS 231N* 7, 7 (2015), 3.
- [68] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

- [69] Eugene Lee, Cheng-Han Huang, and Chen-Yi Lee. 2021. Few-shot and continual learning with attentive independent mechanisms. In *Proceedings of the IEEE/CVF international conference on computer vision*. 9455–9464.
- [70] Soobee Lee, Minindu Weerakoon, Jonghyun Choi, Minjia Zhang, Di Wang, and Myeongjae Jeon. 2022. CarM: Hierarchical episodic memory for continual learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1147–1152.
- [71] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. *Ambient intelligence* (2005), 115–148.
- [72] Shuangfeng Li. 2020. Tensorflow lite: On-device machine learning framework. *J. Comput. Res. Dev* 57 (2020), 1839.
- [73] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. McuNet: Tiny deep learning on IoT devices. *Advances in Neural Information Processing Systems* 33 (2020), 11711–11722.
- [74] Ji Lin, Ligeng Zhu, Wei-ming Chen, Wei-chen Wang, Chuang Gan, and Song Han. 2022. On-Device Training Under 256KB Memory. In *Annual Conference on Neural Information Processing Systems*.
- [75] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards accurate binary convolutional neural network. *Advances in neural information processing systems* 30 (2017).
- [76] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009* (2015).
- [77] Liu Liu, Lei Deng, Xing Hu, Maohua Zhu, Guoqi Li, Yufei Ding, and Yuan Xie. 2018. Dynamic sparse graph for efficient deep learning. *arXiv preprint arXiv:1810.00859* (2018).
- [78] Ziwei Liu, Ping Luo, Xiaoogang Wang, and Xiaoou Tang. 2018. Large-scale celebfaces attributes (celeba) dataset. Retrieved August 15, 2018 (2018), 11.
- [79] David Lopez-Paz and Marc’Aurelio Ranzato. 2017. Gradient episodic memory for continual learning. *Advances in neural information processing systems* 30 (2017).
- [80] Xinyue Ma, Suyeon Jeong, Minjia Zhang, Di Wang, Jonghyun Choi, and Myeongjae Jeon. 2023. Cost-effective on-device continual learning over memory hierarchy with Miro. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. 1–15.
- [81] Subhransu Maji, Esa Rahtu, Juho Kannala, Matthew Blaschko, and Andrea Vedaldi. 2013. Fine-grained visual classification of aircraft. *arXiv preprint arXiv:1306.5151* (2013).
- [82] James L McClelland, Bruce L McNaughton, and Randall C O’Reilly. 1995. 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 (1995), 419.
- [83] Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*. Vol. 24. Elsevier, 109–165.
- [84] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*. PMLR, 1273–1282.
- [85] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Aleksii Kuchaev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).
- [86] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stolic, Dusan Stolic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378* (2021).
- [87] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
- [88] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.
- [89] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. 2019. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 11264–11272.
- [90] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440* (2016).
- [91] Maria-Elena Nilsback and Andrew Zisserman. 2008. Automated flower classification over a large number of classes. In *2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*. IEEE, 722–729.
- [92] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural networks* 113 (2019), 54–71.
- [93] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. 2012. Cats and dogs. In *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 3498–3505.
- [94] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. 2022. POET: Training neural networks on tiny devices with integrated rematerialization and paging. In *International Conference on Machine Learning*. PMLR, 17573–17583.
- [95] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 891–905.
- [96] Pijush Kanti Dutta Pramanik, Nilanjan Sinhababu, Bulbul Mukherjee, Sanjeevikumar Padmanaban, Aranyak Maity, Bijoy Kumar Upadhyaya, Jens Bo Holm-Nielsen, and Prasenjit Choudhury. 2019. Power consumption analysis, measurement, management, and issues: A state-of-the-art review of smartphone battery and energy usage. *IEEE Access* 7 (2019), 182113–182172.

- [97] Christos Profentzas, Magnus Almgren, and Olaf Landsiedel. 2022. MiniLearn: On-Device Learning for Low-Power IoT Devices. In *Proceedings of the 2022 International Conference on Embedded Wireless Systems and Networks (Linz, Austria) (EWSN '22)*. Junction Publishing, USA.
- [98] Zhongnan Qu, Zimu Zhou, Yongxin Tong, and Lothar Thiele. 2022. p-meta: Towards on-device deep model adaptation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 1441–1451.
- [99] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and H Brendan McMahan. 2020. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295* (2020).
- [100] Haoyu Ren, Darko Anicic, and Thomas A Runkler. 2021. Tinyol: Tinyml with online-learning on microcontrollers. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.
- [101] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.
- [102] Matthew Riemer, Ignacio Cases, Robert Ajemian, Miao Liu, Irina Rish, Yuhai Tu, and Gerald Tesaro. 2018. Learning to learn without forgetting by maximizing transfer and minimizing interference. *arXiv preprint arXiv:1810.11910* (2018).
- [103] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhubarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).
- [104] Rohit Roy, Sayantika Dutta, Sagnick Biswas, and Jyoti Sekhar Banerjee. 2020. Android things: A comprehensive solution from things to smart display and speaker. In *Proceedings of International Conference on IoT Inclusive Life (ICIL 2019), NITTTR Chandigarh, India*. Springer, 339–352.
- [105] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115 (2015), 211–252.
- [106] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. 2022. Machine learning for microcontroller-class hardware: A review. *IEEE Sensors Journal* 22, 22 (2022), 21362–21390.
- [107] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.
- [108] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. 2014. CNN features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 806–813.
- [109] Daves Shingari, Akhil Arunkumar, and Carole-Jean Wu. 2015. Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 22–33.
- [110] Daniel L Silver, Qiang Yang, and Lianghao Li. 2013. Lifelong machine learning systems: Beyond learning algorithms. In *2013 AAAI spring symposium series*.
- [111] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
- [112] Charles M Stein, Dinei A Rockenbach, Dalvan Griebler, Massimo Torquati, Gabriele Mencagli, Marco Danelutto, and Luiz G Fernandes. 2021. Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units. *Concurrency and Computation: Practice and Experience* 33, 11 (2021), e5786.
- [113] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
- [114] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.
- [115] Andrew Tanenbaum. 2009. *Modern operating systems*. Pearson Education, Inc.,
- [116] Alexey Tsymbal. 2004. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin* 106, 2 (2004), 58.
- [117] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).
- [118] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. 2016. Matching networks for one shot learning. *Advances in neural information processing systems* 29 (2016).
- [119] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. 2011. The caltech-ucsd birds-200-2011 dataset. (2011).
- [120] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. 2021. AsyMo: Scalable and Efficient Deep-Learning Inference on Asymmetric Mobile CPUs. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (New Orleans, Louisiana) (MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 215–228. <https://doi.org/10.1145/3447993.3448625>
- [121] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: Breaking the Memory Wall for Resource-Efficient on-Device Machine Learning. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (Portland, Oregon) (MobiSys '22)*. Association for Computing Machinery, New York, NY, USA, 450–463. <https://doi.org/10.1145/3498361.3538928>
- [122] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. 2016. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1995–2003.
- [123] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209* (2018).
- [124] Christopher John Cornish Hellaby Watkins. 1989. Learning from delayed rewards. (1989).

- [125] Daliang Xu, Mengwei Xu, Qipeng Wang, Shangguang Wang, Yun Ma, Kang Huang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2022. Mandheling: Mixed-Precision on-Device DNN Training with DSP Offloading. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking* (Sydney, NSW, Australia) (*MobiCom '22*). Association for Computing Machinery, New York, NY, USA, 214–227. <https://doi.org/10.1145/3495243.3560545>
- [126] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. 2019. A First Look at Deep Learning Apps on Smartphones. In *The World Wide Web Conference* (San Francisco, CA, USA) (*WWW '19*). Association for Computing Machinery, New York, NY, USA, 2125–2136. <https://doi.org/10.1145/3308558.3313591>
- [127] Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, and Xuanzhe Liu. 2020. DeepWear: Adaptive Local Offloading for On-Wearable Deep Learning. *IEEE Transactions on Mobile Computing* 19, 2 (2020), 314–330. <https://doi.org/10.1109/TMC.2019.2893250>
- [128] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. 2020. Approximate Query Service on Autonomous IoT Cameras. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services* (Toronto, Ontario, Canada) (*MobiSys '20*). Association for Computing Machinery, New York, NY, USA, 191–205. <https://doi.org/10.1145/3386901.3388948>
- [129] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. 2018. DeepCache: Principled Cache for Mobile Deep Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (New Delhi, India) (*MobiCom '18*). Association for Computing Machinery, New York, NY, USA, 129–144. <https://doi.org/10.1145/3241539.3241563>
- [130] Chengxu Yang, Qipeng Wang, Mengwei Xu, Zhenpeng Chen, Kaigui Bian, Yunxin Liu, and Xuanzhe Liu. 2021. Characterizing impacts of heterogeneity in federated learning upon large-scale smartphone data. In *Proceedings of the Web Conference 2021*. 935–946.
- [131] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. 2020. NEMO: Enabling Neural-Enhanced Video Streaming on Commodity Mobile Devices. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (London, United Kingdom) (*MobiCom '20*). Association for Computing Machinery, New York, NY, USA, Article 28, 14 pages. <https://doi.org/10.1145/3372224.3419185>
- [132] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. 2017. Live video analytics at scale with approximation and {Delay-Tolerance}. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 377–392.
- [133] Qiyang Zhang, Xiang Li, Xiangying Che, Xiao Ma, Ao Zhou, Mengwei Xu, Shangguang Wang, Yun Ma, and Xuanzhe Liu. 2022. A Comprehensive Benchmark of Deep Learning Libraries on Mobile Devices. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) (*WWW '22*). Association for Computing Machinery, New York, NY, USA, 3298–3307. <https://doi.org/10.1145/3485447.3512148>
- [134] Yu Zhang, Tao Gu, and Xi Zhang. 2020. 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*. 463–475.
- [135] Tianming Zhao, Yucheng Xie, Yan Wang, Jerry Cheng, Xiaonan Guo, Bin Hu, and Yingying Chen. 2022. A survey of deep learning on mobile devices: Applications, optimizations, challenges, and research opportunities. *Proc. IEEE* 110, 3 (2022), 334–354.
- [136] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).
- [137] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.