Federated Machine Learning with Resource-Constrained Devices

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik des

Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Kilian Pfeiffer

aus Amberg

Tag der mündlichen Prüfung: 28. April 2025
1. Referent: Prof. Dr.-Ing. Jörg Henkel, Karlsruher Institut für Technologie (KIT)
2. Referent: Prof. Dr. Jian-Jia Chen, Technische Universität (TU) Dortmund

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen – die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kilian Pfeiffer

Acknowledgements

I would like to thank Prof. Dr.-Ing. Jörg Henkel for supervising my dissertation and for his valuable advice throughout the years. I greatly appreciate the research environment he provided and the academic freedom that allowed me to explore new fields of interest. I would also like to thank Prof. Dr. Jian-Jia Chen for kindly agreeing to be the co-advisor of my dissertation.

I am deeply grateful to Dr.-Ing. Martin Rapp, who invested a great deal of time in guiding me early on, helping me avoid many pitfalls in academia. His support significantly accelerated both my research and personal growth. Many thanks to Dr. Ramin Khalili for his continuous support, interest in my research ideas, and—most importantly—his optimism about my work, which helped me endure the challenges of my Ph.D.

Thank you to Mohamed Ahmed for insightful discussions and feedback on my dissertation. Thanks also to all my colleagues at the Chair for Embedded Systems for their extensive help with my Ph.D. defense preparation, the enjoyable work environment, engaging discussions, and the many memorable events we shared.

I am immensely grateful to my parents for their patience and enduring support throughout my education. To my friends, thank you for your constant encouragement and for visiting me in Karlsruhe. Finally, my deepest gratitude goes to my girlfriend Jana, for supporting me throughout the years, for listening patiently to my thoughts and ideas, and for being by my side through all the ups and downs.

Kilian Pfeiffer

Abstract

Federated learning (FL) is a recently introduced machine learning (ML) training scheme, in which devices train locally with their data but exchange knowledge through the parameters of the trained neural network (NN) model. The exchanged parameters are aggregated centrally, forming a global NN model. Compared to centralized training of NNs, FL improves privacy as it does not require sending raw user data to a centralized entity for training. However, while privacy is improved, FL imposes the resource-intensive training task on typically constrained edge devices, such as internet of things (IoT) devices or smartphones. These devices often vary in their ability to perform training on the NN due to differences in the communication channel, computational capabilities, or memory availability. Training FL systems remains a major challenge due to the heterogeneous nature of devices, as current production deployments often exclude devices that do not meet the required capabilities from the FL training process. This dissertation presents several techniques that allow constrained devices to participate in FL training, enhancing the effectiveness of FL systems and increasing fairness among participants.

This dissertation examines in detail common devices and how their capabilities impact their ability to perform training. Further, the implications of these constraints for participation in FL are discussed, as well as the extent to which existing techniques tackle these problems. Importantly, the issue of interdependencies between specific types of data distributions (non-independent and identically distributed (non-iid)) present on devices and resource heterogeneity among devices is explored.

To combat these challenges, this dissertation introduces a set of techniques. First, a technique is introduced that enables constrained devices to participate in synchronous FL despite round-varying resource constraints. This is achieved by allowing devices to partially freeze and quantize the NN during a round of training, enabling them to adjust to communication, computation, and peak memory constraints independently by modifying the specific layer freezing setup. Furthermore, this technique is the first to evaluate the existing interdependencies between resource heterogeneity and data distribution heterogeneity (non-iid), demonstrating that the proposed algorithm promotes high fairness among devices despite differences in resources, which state-of-the-art solutions fail to provide.

The technique is extended to incorporate centrally available existing pre-training datasets. As a consequence of having a pre-trained NN, not all of its layers require federated training. An algorithm is presented that selects a suitable NN structure, trainable by all heterogeneous devices, while maximizing achievable accuracy.

Second, to tackle fixed peak memory constraints, a technique is proposed that allows the NN model to grow step by step while freezing early layers. This enables constrained devices in FL to train a large global FL model that no device is capable of training end-to-end on its own. Communication and computation should thereby be used efficiently; however, they are not treated as per-round constraints. While adhering to memory constraints, FL training is significantly accelerated and achieves higher accuracies compared to existing works and end-to-end training of feasible smaller models.

Third, a hardware-design-based technique is introduced that tackles heterogeneity at design time, specifically energy, by proposing designs for resource-specific ASIC-based accelerators. These accelerators utilize compressed arithmetic formats and approximate computing in convolutional and linear operations, significantly reducing the energy overhead of computations and memory accesses for constrained devices, with minimal impact on accuracy. The evaluation shows that approximate computing principles are suitable for FL, enabling constrained devices to participate and ensuring their training impacts the global FL model.

Ultimately, the contributions of this dissertation enhance the practical deployment of FL in real-world scenarios, improving FL's resilience in heterogeneous settings. The proposed techniques enable resource-constrained devices to meaningfully participate in FL, ensuring their knowledge is effectively integrated into the global FL model—an aspect where previous works have fallen short. This promotes a fairer FL framework, thereby improving the quality of services for a broader range of devices and users.

Zusammenfassung

Föderiertes Lernen (FL) ist ein kürzlich eingeführtes Trainingsschema für maschinelles Lernen (ML), bei dem Geräte lokal mit ihren eigenen Daten trainieren, aber gelerntes Wissen durch die Parameter des trainierten neuronalen Netzwerks (NN) austauschen. Die ausgetauschten Parameter werden zentral aggregiert und bilden ein globales NN. Im Vergleich zum zentralisierten Training neuronaler Netzwerke verbessert föderiertes Lernen die Privatsphäre, da es nicht erforderlich ist, Rohdaten an eine zentrale Instanz zum Training zu senden. Während die Privatsphäre verbessert wird, erfordert föderiertes Lernen jedoch ressourcenintensives Training auf typischerweise eingeschränkten Edge-Geräten wie IoT-Geräten oder Smartphones. Diese Geräte unterscheiden sich häufig in ihrer Fähigkeit, ein NN zu trainieren, aufgrund von Unterschieden in der Kommunikationsverbindung, den Rechenkapazitäten oder im verfügbaren Hauptspeicher. Das Training von FL-Systemen bleibt aufgrund der heterogenen Geräte eine große Herausforderung, da in aktuellen Anwendungen häufig Geräte vom FL-Trainingsprozess ausgeschlossen werden, die die erforderlichen Kapazitäten nicht erfüllen. Diese Dissertation präsentiert mehrere Techniken, die es eingeschränkten Geräten ermöglichen, am FL-Training teilzunehmen, wodurch die Effektivität von FL-Systemen verbessert und die Fairness unter den Teilnehmern erhöht wird.

Diese Dissertation untersucht im Detail gängige Geräte und wie ihre Fähigkeiten ihre Trainingsmöglichkeiten beeinflussen. Darüber hinaus werden die Auswirkungen dieser Einschränkungen auf die Teilnahme an föderiertem Lernen sowie das Ausmaß, in dem bestehende Techniken diese Probleme bewältigen, diskutiert. Wichtig ist zudem die Untersuchung der Wechselwirkungen zwischen spezifischen Arten von nicht unabhängig und identisch verteilten (non-iid) Daten auf den Geräten und der Ressourcenheterogenität zwischen den Geräten.

Um diese Herausforderungen zu adressieren, stellt diese Dissertation eine Reihe von Techniken vor. Erstens wird eine Technik eingeführt, die es eingeschränkten Geräten ermöglicht, trotz rundenvariierender Ressourcenbeschränkungen am synchronen FL teilzunehmen. Dies wird erreicht, indem Geräte während einer Trainingsrunde das NN teilweise einfrieren und quantisieren, sodass sie sich unabhängig an Kommunikations-, Rechen- und Spitzenarbeitsspeicherbeschränkungen anpassen können. Darüber hinaus ist diese Technik die erste, die die bestehenden Wechselwirkungen zwischen Ressourcenheterogenität und non-iid-Daten untersucht und zeigt, dass der vorgeschlagene Algorithmus trotz unterschiedlicher Ressourcen der Geräte eine hohe Fairness unter den Geräten fördert, was aktuelle Lösungen nicht gewährleisten können.

Die Technik wird erweitert, um zentral verfügbare bestehende Pretraining-Datensätze einzubeziehen. Da ein vortrainiertes NN vorliegt, erfordern nicht alle seine Schichten ein föderiertes Training. Es wird ein Algorithmus vorgestellt, der eine geeignete Netzwerkstruktur auswählt, die von allen heterogenen Geräten trainiert werden kann und gleichzeitig die erreichbare Genauigkeit maximiert.

Zweitens wird zur Bewältigung fester Arbeitsspeicherbeschränkungen eine Technik vorgeschlagen, die es ermöglicht, das neuronale Netzwerk schrittweise zu erweitern, während frühe Schichten eingefroren werden. Dies ermöglicht es eingeschränkten Geräten im föderierten Lernen, ein großes globales FL-Modell zu trainieren, das kein einzelnes Gerät eigenständig von Anfang bis Ende trainieren könnte. Kommunikation und Rechenleistung sollten dabei effizient genutzt werden, werden jedoch nicht als rundenabhängige Beschränkungen betrachtet. Während die Speicherbeschränkungen eingehalten werden, wird das FL-Training erheblich beschleunigt und erreicht höhere Genauigkeiten im Vergleich zu bereits existierenden Techniken sowie zum vollständigen Training praktikabel kleinerer Modelle.

Drittens wird eine hardwaredesignbasierte Technik eingeführt, die Heterogenität bereits zur Designzeit, insbesondere im Hinblick auf den Energieverbrauch, adressiert. Hierbei werden anwendungsspezifische integrierte Schaltkreis (ASIC)-basierte Beschleuniger entworfen, die komprimierte arithmetische Formate und Approximationsberechnungen für die Berechnung von Faltungs- und linearen Operationen nutzen. Dadurch wird der Energieverbrauch von Berechnungen und Speicherzugriffen für eingeschränkte Geräte erheblich reduziert – bei minimalen Auswirkungen auf die Genauigkeit. Die Evaluierung zeigt, dass Prinzipien des Approximationsrechnens für föderiertes Lernen geeignet sind, indem sie eingeschränkten Geräten die Teilnahme ermöglichen und sicherstellen, dass ihr Training einen Einfluss auf das globale FL-Modell hat.

Letztendlich verbessern die Beiträge dieser Dissertation die praktische Einsetzbarkeit von FL in realen Anwendungsszenarien und erhöhen die Widerstandsfähigkeit von FL in heterogenen Umgebungen. Die vorgeschlagenen Techniken ermöglichen es ressourcenbeschränkten Geräten, sinnvoll an FL teilzunehmen, sodass ihr Wissen effektiv in das globale FL-Modell integriert wird – ein Aspekt, bei dem frühere Arbeiten nicht überzeugen konnten. Dies fördert ein faireres FL-Framework und verbessert somit die Qualität der Dienste für eine breitere Palette von Geräten und Nutzern.

List of Publications

First-authored publications that make major contributions to this dissertation

[1] Kilian Pfeiffer, Martin Rapp, Ramin Khalili, and Jörg Henkel. "CoCoFL: Communication- and Computation-Aware Federated Learning via Partial NN Freezing and Quantization". In: *Transactions on Machine Learning Research (TMLR)* (2023). ISSN: 2835-8856. URL: https://openreview.net/forum?id=XJIg4kQbkv.

[2] Kilian Pfeiffer, Ramin Khalili, and Jörg Henkel. "Aggregating Capacity in FL through Successive Layer Training for Computationally-Constrained Devices". In: *Advances in Neural Information Processing Systems (NeurIPS)* 36 (2024).

[3] Kilian Pfeiffer, Martin Rapp, Ramin Khalili, and Jörg Henkel. "Federated Learning for Computationally Constrained Heterogeneous Devices: A Survey". In: *ACM Computing Surveys (ACM CSUR)* 55.14s (2023), pp. 1–27.

[4] Kilian Pfeiffer, Konstantinos Balaskas, Kostas Siozios, and Jörg Henkel. "Energy-Aware Heterogeneous Federated Learning via Approximate Systolic DNN Accelerators". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2024).

[5] Kilian Pfeiffer, Mohamed Aboelenien Ahmed, Ramin Khalili, and Jörg Henkel. Efficient Federated Finetuning of Tiny Transformers with Resource-Constrained Devices. 2024. arXiv: 2411.07826 [cs.LG].

Co-authored publications

[6] Martin Rapp, Ramin Khalili, Kilian Pfeiffer, and Jörg Henkel. "Distreal: Distributed Resource-Aware Learning in Heterogeneous Systems". In: *AAAI Conference on Artificial Intelligence*. Vol. 36. 7. 2022, pp. 8062–8071.

[7] Mohamed Aboelenien Ahmed, Kilian Pfeiffer, Ramin Khalili, Heba Khdr, and Jörg Henkel. Efficient Zero-Order Federated Finetuning of Language Models for Resource-Constrained Devices. 2025. arXiv: 2502.10239 [cs.LG].

[8] Mohamed Aboelenien Ahmed, Kilian Pfeiffer, Heba Khdr, Osama Abboud, Ramin Khalili, and Jörg Henkel. Accelerated Training on Low-Power Edge Devices. 2025. arXiv: 2502.18323 [cs.LG].

Contents

1 Introduction

In recent years, machine learning (ML) has been gaining more and more traction in many daily life applications, showing impressive results in the fields of smart assistants, healthcare, autonomous driving, robotics, and many more. Slowly, training-based techniques, most notably neural networks (NNs), are replacing previously used heuristics or analytical models. Especially in problems like language processing, language generation, object classification, and object detection, the improvements of training-based algorithms have enabled tremendous progress. The success of ML is based on two important factors. Firstly, in order to train an ML model that accurately predicts a desired output, a large quantity of high-quality training data must be available. Secondly, training ML models, as it is commonly an iterative process, requires a lot of time and computational resources to adjust every parameter in the model to eventually make high-quality predictions. The contributing factors to the computational complexity of the training are the NN topology, which usually boils down to many matrix-matrix dot products or convolution operations, and the quantity of training data that is used. Both factors have been massively increasing throughout the last years.

1.1 Resource-Intensiveness of Machine Learning

In 1998, with the introduction of one of the first many-layered NNs called LeNet [9], the number of parameters in the respective model was 70 thousand, while the data used for classifying handwritten digits consisted of 60 thousand labeled samples with a very low resolution of 28×28 pixels (roughly 188 MB of data). In 2012, significant progress in the field of object classification was achieved through the introduction of very deep NN structures. These architectures [10] consist of 60 million parameters and were trained on an image dataset [11] with 1000 classes and 14 million samples, $3 \times 256 \times 256$ pixels each (a total of ~ 1.1 TB of data). Training such a deep NN to convergence could take up to one week using, at that time, state-of-the-art graphics processing units (GPUs) [10]. These examples demonstrate that the computational requirements in ML are growing at a massive pace, so much so that hardware development following Moore's Law, i.e., the number of transistors on an integrated circuit doubling every two years, cannot keep up with the demands of ML.

With the popularization of transformer architectures [12], which can be trained auto-regressively, vast quantities of unlabeled text can now be utilized to train very deep NN

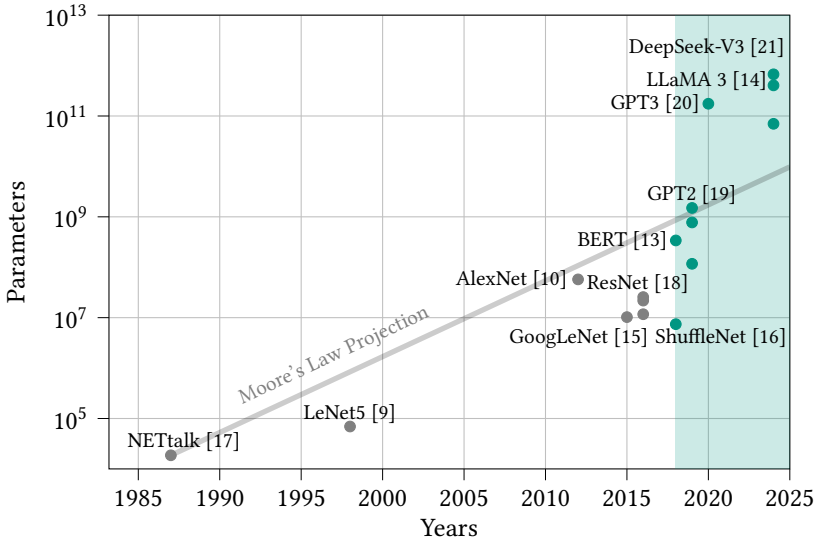


Figure 1.1: Increase in parameters of ML models over the years compared to the progress in ML hardware represented by Moore's Law. In recent years, the size of NNs has outpaced Moore's Law.

models. Since 2018, with the introduction of BERT [13], the growth in parameters of these models has far outpaced hardware advances. For example, LLaMA [14], consisting of 70 billion parameters, was trained on 4.8 TB of data (specifically, text available on the internet), which equates to 1.4 trillion samples. Training models with such a large number of parameters on such vast amounts of data requires a massive amount of energy and results in training costs in the range of millions of dollars.

These examples highlight that ML is a resource-intensive task, and the resource requirements are continuously growing, outpacing the rapid advancements in training hardware. Figure 1.1 visualizes the continuous growth of parameters, which serve as a proxy for computational complexity in ML, and compares it to the progress in hardware represented by Moore's Law.

1.2 On-Device Machine Learning

The excessive resource requirements described above demand training in a centralized environment (e.g., a cloud server farm), where sufficient energy supply, cooling, and low-latency access to data are available. However, for many applications such as healthcare, transportation, or personal assistants on smartphones, centralizing training data is problematic due to the presence of private information that must not be shared with others. Additionally, there are cases where sending the data to a centralized entity creates

a significant communication burden, as the number of internet of things (IoT) devices is expected to grow [22, 23]. As a consequence, there is a significant effort to process the data (e.g., training an NN) locally on the devices that produce the data. This trend is further driven by legislation that strengthens data privacy, such as the European Union's GDPR [24] and California's CCPA [25]. However, these edge devices are usually not as powerful as server GPUs that consume several hundreds of watts, but rather low-powered IoT devices such as healthcare devices, smartphones, robots, or sensors.

While on-device training enables privacy-preserving training of ML models, it still faces many challenges. Most importantly, IoT devices are heavily constrained in their training capabilities. Unlike server GPUs, which are optimized for parallel processing of typical NN workloads, edge devices utilize less powerful central processing units (CPUs) designed for general-purpose tasks. Moreover, these devices usually require CPU time to fulfill their main tasks (e.g., reacting to user interaction in the case of a smartphone), thus they have to share resources with the NN training task. While more recent devices often have custom application-specific integrated circuits (ASICs) for NN inference [26], the reduced-precision arithmetic of these accelerators generally does not allow for training of NNs. Unlike server farms, edge devices such as smartphones or wearable health devices are battery-powered. Performing energy-intensive NN training tasks on such devices can rapidly deplete the battery life and significantly worsen the user's experience. Such devices can consume at most a few watts, whereas server GPUs can consume several hundreds of watts. Further, utilizing their embedded processors for extended periods, as required for training, leads to inadequate heat dissipation, as these devices usually have no active cooling. Devices heat up and deplete their batteries faster. To stay within thermal limits, devices employ thermal throttling of their CPU, reducing its clock speed and further slowing down training. Another factor that heavily impacts training capabilities is the availability of system memory (specifically, random access memory (RAM)). For cost savings, many of these devices are equipped with little RAM, enough to fulfill their primary task. However, training requires parameters, gradients, data, and intermediate activations to be kept in RAM, in many cases significantly exceeding the available RAM. Lastly, storage (e.g., flash storage) on such devices is limited, restricting the storage of NN topologies and data.

As a consequence, training on such devices is challenging, and significantly smaller NN models must suffice for these devices. Training smaller NN models limits the NN's capacity, which can impact the NN's achievable prediction performance. Since the device is training solely on its own limited data (a small subset of a potentially large centralized dataset), the resulting ML model suffers from weak generalization properties, which might induce accuracy losses.

1.3 Federated Learning

Federated learning (FL) [27] is a recently introduced distributed training approach that aims to address the main issues associated with isolated on-device training. In FL, training is done in a distributed manner on each device, but devices collaborate with each other by exchanging knowledge. Thus, FL improves privacy compared to the centralized training paradigm, as no data is shared with a centralized entity [28, 29]. In order to still be able to exchange knowledge, only the trained NN parameters are shared with a centralized entity in FL. The parameters are uploaded from the devices to the server in a synchronous, round-based manner and averaged on the server [27]. Thus, compared to isolated training, recurring communication between a server and devices is required. Over the last few years, FL has been proposed for many use cases in healthcare [30, 31, 32, 33], transportation [34, 35, 36, 37], robotics [38, 39], and smartphone applications [40, 41]; however, only a few real-world deployments are known. A widely used FL system is Google GBoard [40], where a common ML model is collectively trained to predict word suggestions that a smartphone user might want to use when typing on the smartphone's keyboard.

However, deploying FL systems remains challenging. One reason is the variability of data on the devices, i.e., the local private distribution varies among the devices. This is generally caused by the fact that devices generate different kinds of data, influenced by factors like device hardware, user behavior, or location-specific differences. For example, in an image classification system, images on some devices might be brighter or have specific lens distortions compared to others due to differences in the camera system. Another difference might be the quantities of samples per class on devices. Variability like this generally causes slowdowns in training or lower accuracies, as the global training objective and the devices' local objectives are not aligned. This kind of non-independent and identically distributed (non-iid) data is one of the main factors that causes FL to be significantly slower in training convergence and produce lower-accuracy models compared to a setup where all device data is centrally trained.

Another major challenge in FL is the limited resources for training and the heterogeneity of the participating devices [3]. Contrary to a datacenter, where training hardware and software are fully controlled, in FL, devices can have various differing hardware properties as well as other constraints like concurrent tasks, limited communication bandwidth, cooling, or energy that limit their capabilities to perform training. For example, in Google Gboard, devices with less than 2 GB available for training are dropped from the distributed training system [40]. While for large-scale applications like Google Gboard, dropping devices can be acceptable, in many smaller-scale applications, excluding devices results in the loss of valuable data, causing the trained NN model to generalize less effectively and suffer from lower accuracy. Furthermore, excluding specific types of devices due to resource limitations can introduce biased predictions and unfairness, as the data from these devices is no longer represented in the training process [42, 1]. If such a model is later deployed for inference on the excluded devices, the NN model is likely to perform poorly on their typical data, exacerbating disparities in model performance across different

device types. Consequently, data heterogeneity and hardware heterogeneity are often intertwined, as differences in device capabilities frequently correlate with variations in the data they generate or collect.

1.4 Dissertation Contribution

This dissertation makes several contributions tackling these challenges. It addresses device heterogeneity with respect to resources such as communication constraints, computation constraints, memory constraints, and energy constraints. Furthermore, it investigates the interdependencies between heterogeneous resources and non-iid data among such devices. In broader terms, this dissertation enhances the participation of constrained devices in heterogeneous FL through a set of techniques that address the problem at multiple levels of the compute stack, including the abstract algorithm level, the utilization of available arithmetic acceleration, and the design of FL-specific hardware. While previous work has primarily focused on tackling individual constraints in isolation, this dissertation presents techniques that address these constraints jointly, enabling flexible adaptation to multiple constraints while efficiently utilizing available resources. Moreover, whereas prior approaches have allowed for device participation in FL, the techniques developed in this dissertation significantly enhance their effectiveness with respect to accuracy. Consequently, these techniques enable constrained devices to make meaningful contributions within FL, thereby promoting greater fairness.

The main contributions of this dissertation are outlined as follows:

- This dissertation examines in detail the training capabilities of common devices and how these capabilities impact their ability to perform training. Additionally, it investigates the implications of these constraints and the resulting heterogeneity within an FL system, assessing the extent to which existing techniques address these challenges. Furthermore, the interdependency between data heterogeneity and resource heterogeneity is explored through the introduction of a specifically simulated data distribution, labeled resource-correlated non-independent and identically distributed (rc-non-iid).
- A technique is introduced that utilizes partial freezing and quantization of NNs on constrained devices, thereby enabling these devices to participate in FL by independently adapting to FL round-specific communication, computation, and peak memory constraints. This technique facilitates resource adjustments through algorithm-level training modifications, such as freezing and operator fusion, and is further enhanced by hardware-level layer quantization based on integer arithmetic. Finally, the proposed technique significantly improves the ability of constrained devices to contribute to the FL model under an rc-non-iid data regime.
- To specifically address fixed memory constraints, a technique is introduced that enables the training of a large server NN, which no participating device can train

end-to-end due to peak memory limitations. In this setup, memory is treated as a strict constraint, while communication and computation are not considered round-based constraints but should still be utilized efficiently. To achieve this, an algorithm-level technique is proposed that incrementally trains the NN step-by-step, allowing participating devices to handle larger NN models despite memory limitations. This technique, while adhering to memory constraints, significantly accelerates accuracy improvements with respect to computation and communication, substantially outperforming existing methods in both homogeneous and heterogeneous FL settings.

- An extension to the partial freezing scheme is introduced that takes into account the availability of a centralized dataset for pre-training and the optimal selection of NN architectures.
- A hardware-level technique is introduced that, at design time, tackles energy heterogeneity in FL by designing constraint-specific ASIC-based ML training accelerators. The adjustability to device-specific energy constraints is enabled through the use of approximate multiply accumulate (MAC) units in the systolic array (SA), specifically by approximating the mantissa multiplication of floating-point numbers. Additionally, arithmetic units are truncated in memory to reduce the energy consumption of memory operations.

1.5 Dissertation Outline

The remainder of this dissertation is structured as follows: Chapter 2 introduces the main concepts of NN-based machine learning and the basics of FL. Additionally, it formally describes typical data distributions such as independent and identically distributed (iid), non-iid, and rc-non-iid. Further, a detailed analysis of resources and constraints of typical edge devices is provided, along with their implications on FL applications. Existing work and typical baselines are introduced in Chapter 3. Chapter 4 presents the experimental setup, including the simulation environment, used datasets, and NN topologies. Chapter 5 and Chapter 6 present FL techniques that enable devices to participate in and contribute to FL training while adhering to specific device constraints. Chapter 7 extends Chapter 5 to incorporate pre-training and optimal NN architecture selection in FL. Chapter 8 explores hardware design considerations that can mitigate heterogeneity in device capabilities at the design stage. Lastly, Chapter 9 concludes the dissertation and provides an outlook on future research directions.

2 Background

The following chapter introduces ML and FL in general in sections 2.1 and 2.2 and examines in detail the training capabilities of common devices and how these capabilities impact their ability to perform training in section 2.3. Furthermore, it presents the background and notation for the contributions in chapters 5 to 8.

2.1 Machine Learning

Designing classical algorithms usually involves crafting a function F that, given some data input, produces the desired output. However, while it is straightforward to analytically find an algorithm for many applications and tasks, domains such as image or audio processing involve high-dimensional data. As the number of feature dimensions increases, the volume of the space grows exponentially. This causes data points to appear sparsely distributed, reducing the effectiveness of distance-based metrics and making it challenging to identify clear decision boundaries for classification. This effect is known as *curse of dimensionality*. For example, when classifying images, it is infeasible to design a rule-based algorithm to detect high-level concepts like "a dog" or "a cat" in high-dimensional input data.

In the last 15 years, machine learning, i.e., learning the behavior of F using available data $\boldsymbol{x}$, has become very popular and has shown success in many domains, such as image or video processing, audio, and text processing. It has tackled many applications, such as autonomous driving, robotics, and healthcare, thereby slowly replacing analytically derived algorithms.

The following chapter introduces the most important basics of ML that are relevant for this dissertation. To maintain brevity, this introduction focuses on two subsets of ML. Firstly, while many types of learnable structures F, such as decision trees, random forests, regression models, and support vector machines, exist, this introduction focuses on NNs, as NNs represent the most successful and widely used branch of learnable models. Secondly, while many application domains, such as regression tasks, anomaly detection, denoising, and sequence-to-sequence modeling, exist, this introduction focuses on data classification.

The introduction to ML is based on Goodfellow et al. [43]. Sections 2.2 and 2.3 are based on [3, 1].

We assume multidimensional data $\boldsymbol{x}$ and an associated class y that belong to a dataset $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_{|\mathcal{D}|}, y_{|\mathcal{D}|})\}$. In general, in ML, and more specifically in classification tasks, one tries to find a function F such that $F : \mathbb{R}^P \mapsto \mathcal{Y}$, where $\mathcal{Y}$ is the set of possible classes. Consequently, if a P-dimensional input vector $\boldsymbol{x}$ with $\boldsymbol{x} \in \mathbb{R}^P$ is used as input for F, then $\hat{y} = F(\boldsymbol{x})$, where $\hat{y}$ is a prediction of the class associated with the data $\boldsymbol{x}$. A prediction of $F(\boldsymbol{x})$ is correct if $\hat{y} = y$, where y represents the ground-truth label.

2.1.1 Layers

NNs, especially *deep* NNs, comprise many *layers* f_k, i.e., linear or non-linear functions. Therefore, an NN architecture $\hat{y} = F(\boldsymbol{x})$ can be expressed as $\hat{y} = f_K(f_{K-1}(\ldots(f_1(\boldsymbol{x}))))$, or alternatively described by

$$F(\boldsymbol{x}) = \bigcirc_{i=2}^{K} f_i \circ f_1(\boldsymbol{x}). \tag{2.1}$$

Throughout this dissertation, the term *layer* may refer to a single operation (e.g., a convolutional operation or a linear layer), but mostly refers to a group of several consecutive operations.

Linear layer: A linear layer describes the most basic linear relationship that maps an input $\boldsymbol{x}$ from $\mathbb{R}^P$ to $\mathbb{R}^V$ using a parameter matrix $\boldsymbol{\theta} \in \mathbb{R}^{P \times V}$. The linear transformation can be described using

$$\boldsymbol{z} = \boldsymbol{x}\boldsymbol{\theta} + \boldsymbol{b}, \tag{2.2}$$

where $\boldsymbol{z} \in \mathbb{R}^V$ is the layer's output, while $\boldsymbol{b} \in \mathbb{R}^V$ is the bias vector. Through the matrix-vector multiplication, every element in the input vector impacts every element in the output vector. A linear layer is also referred to as a dense layer.

Convolutional layers: For very large input vectors, optimizing the parameters of linear layers can become very inefficient. A highly successful layer that addresses this inefficiency is the convolutional layer, specifically because convolutional layers use a convolution operation, where the convolution kernel or filter serves as trainable parameters. Unlike linear layers, the same filter parameters are convolved across the input, thereby reducing the total number of trainable parameters required. A convolution operation with a kernel $\boldsymbol{\theta} \in \mathbb{R}^U$ can be expressed as

$$\boldsymbol{z} = \boldsymbol{\theta} \star \boldsymbol{x}, \tag{2.3}$$

where U refers to the size of the kernel. Very often, convolutional layers are used in the context of image processing, where the input consists of several 2D input channels, often also referred to as feature maps. In this case, an output feature map is usually the summation of all input feature maps with a specific kernel. The calculation of an

output $\boldsymbol{z}_i \in \mathbb{R}^{V \times V}$, where $i \in [1, \ldots, C_{\text{out}}]$ refers to the index of the output feature map (C_{out} in total), can be expressed as

$$\boldsymbol{z}_i = b_i \cdot \mathbf{1}_{V \times V} + \sum_{j=1}^{C_{\text{in}}} \boldsymbol{\theta}_{ij} \star \boldsymbol{x}_j, \tag{2.4}$$

where $\boldsymbol{\theta}_{ij} \in \mathbb{R}^{U \times U}$ is a 2D kernel, $\boldsymbol{x}_j \in \mathbb{R}^{P \times P}$ is a 2D input feature map, and b_i is the scalar output feature map bias. Differently from linear layers, the kernel dimension of $\boldsymbol{\theta}_{ij}$ is typically significantly smaller than the size of the input feature map (i.e., 3×3). Moreover, there exist several variants of the convolution operation that include strides, specific padding of feature-map boundaries, or dilation. The kernel size and these parameters control the dimension of the output feature map relative to the input, hence they can shrink or expand it.

Non-linearities: In order for the NN to perform a non-linear mapping between the input and the output, non-linear functions are required. A very common layer that enables this is an activation layer, e.g., ReLU. For every input scalar value x_i with $i \in [1, \ldots, P]$, the output z_i is determined as follows:

$$z_i = \max(0, x_i). \tag{2.5}$$

Other commonly used types of layers are normalization layers. In general, normalization layers help stabilize and accelerate training. One example of such a layer is a Batch Normalization (BN) [44] layer. A BN layer takes the input $\boldsymbol{x}$ of a previous layer, subtracts the mean, and scales the input to have unit variance, thereby ensuring that the input for the consecutive layer has a standardized mean and variance. Additionally, BN applies learnable parameters to scale and shift the output. The BN operation that takes $\boldsymbol{x}$ as input and returns $\boldsymbol{z}$ as output can be expressed as follows:

$$\boldsymbol{z} = \frac{\boldsymbol{x} - \boldsymbol{\mu}_{\boldsymbol{x}}}{\sqrt{\boldsymbol{\sigma}_{\boldsymbol{x}}^2 + \epsilon}} \cdot \boldsymbol{\gamma} + \boldsymbol{\beta}, \tag{2.6}$$

where $\boldsymbol{\mu}_{\boldsymbol{x}}$ represents the vector containing the mean of the input $\boldsymbol{x}$ (usually, BN layers are used in conjunction with convolutional layers, and the mean of $\boldsymbol{x}$ is calculated feature-map-wise). The variance of $\boldsymbol{x}$ is represented by $\boldsymbol{\sigma}_{\boldsymbol{x}}^2$. The trainable scale and bias are represented by $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, which both scale and shift the output (usually with a scalar per feature map). Lastly, a small value ϵ is used for numerical stability. Besides BN, there also exist other normalization layers like LayerNorm [45] that are independent of the minibatch.

2.1.2 Training of NNs

In machine learning, the dataset is usually split into two sets: a set generally referred to as the *training* set and a set referred to as the *test* set. Generally, one aims to optimize $\boldsymbol{\theta}$ using

the training set such that F generalizes (i.e., has good performance) on the test set. This is called *empirical risk minimization*. In order to train the function F, or more specifically, the NN, we define a cost function that measures the performance of F as follows:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p}[L(F(\boldsymbol{x},\boldsymbol{\theta}),y)], \tag{2.7}$$

where L is the per-sample loss. As the size of the dataset $\mathcal{D}$ is usually very large, optimizing $\boldsymbol{\theta}$ using $\mathcal{D}$ all at once is computationally infeasible. Hence, the dataset is split into smaller chunks called *minibatches*, and $\boldsymbol{\theta}$ is optimized using many gradient descent steps. Further, we optimize the NN parameters using many data samples (minibatch) within one optimization step. For a minibatch with batch size B, we use

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \frac{1}{B}\sum_{i=1}^{B} L\big(F(\boldsymbol{x}_i,\boldsymbol{\theta}),y_i\big), \tag{2.8}$$

where $F(\boldsymbol{x}_i,\boldsymbol{\theta})$ represents the prediction for the i-th data sample. The gradient of $\boldsymbol{\theta}$ is calculated using

$$\nabla\boldsymbol{\theta} = \frac{\partial L(F(\boldsymbol{x},y))}{\partial\boldsymbol{\theta}}. \tag{2.9}$$

As deep NNs usually comprise many layers, the chain rule can be applied to calculate the gradient $\nabla\boldsymbol{\theta}_k$ for a specific layer k for a single input $\boldsymbol{x}$ using

$$\nabla\boldsymbol{\theta}_k = \frac{\partial L(F(\boldsymbol{x},\boldsymbol{\theta}),y)}{\partial\boldsymbol{\theta}_k} = \frac{\partial L(F(\boldsymbol{x},\boldsymbol{\theta}),y)}{\partial\hat{y}}\left(\prod_{i=k+1}^{K}\frac{\partial f_i(\boldsymbol{x}_i)}{\partial\boldsymbol{x}_i}\right)\frac{\partial f_k(\boldsymbol{x}_k)}{\partial\boldsymbol{\theta}_k}. \tag{2.10}$$

However, usually, gradients for a whole minibatch are calculated per layer in parallel. As the chain rule requires calculating several gradients with respect to the input, i.e., $\frac{\partial f_k(\boldsymbol{x}_k)}{\partial\boldsymbol{x}_k}$, the respective inputs (layer activations) $\boldsymbol{x}_k$ have to be kept in memory during the forward pass. Lastly, the layer's parameter $\boldsymbol{\theta}_k$ is updated using $\boldsymbol{\theta}_k^* = \boldsymbol{\theta}_k - \eta\nabla\boldsymbol{\theta}_k$, where η defines the step size, which is usually referred to as the *learning rate*.

2.2 Federated Learning

Typically, in FL, there are many *devices* $c \in C$ that participate in the training and communicate only with a single *server*. While in centralized training all data $\mathcal{D}$ is centrally available, in FL we assume that each device possesses a local dataset $\mathcal{D}_c$ that is stored on the device and must not be shared.

While there are many different algorithms like FedSGD [27], FedMD [46], or split learning that implement FL behavior, this background section focuses on federated averaging (FedAvg). FedAvg was introduced by McMahan et al. [27] and is generally considered a baseline for FL. In the case of synchronous FL, training is conducted in synchronous

rounds $r \in [1, \ldots, R]$, limited to a total number of rounds R. In each round, participating devices pull the latest NN model from the server, where $\boldsymbol{w}$ represents the weights (i.e., parameters of several layers) of the NN. Following that, each device initializes its local training for a fixed number of minibatches (or epochs) on its data. After training, each participating device uploads its updated model to the server. The server model is updated by averaging the received NN models.

The steps of FedAvg are enumerated in detail as follows and are depicted in fig. 2.1:

Step 1 At the beginning of every training round r, participating devices $C^{(r)}$ pull the latest model $\boldsymbol{w}^{(r)}$ from the server.

Step 2 Devices $c \in C^{(r)}$ perform training, i.e., stochastic gradient descent (SGD), for a fixed number of minibatches (or epochs) on $\boldsymbol{w}_c^{(r)}$ using

$$\boldsymbol{w}_c^{(r+1)} = \boldsymbol{w}_c^{(r)} - \eta \nabla F(\boldsymbol{w}_c^{(r)}), \tag{2.11}$$

where $\boldsymbol{w}_c^{(r+1)}$ is the locally updated model.

Step 3 Following training, devices transfer their model weights $\boldsymbol{w}_c^{(r+1)}$ to the server.

Step 4 After the server has received all models from the participating devices, it aggregates the models into a single model using

$$\boldsymbol{w}^{(r+1)} = \frac{1}{|\mathcal{D}_{C^{(r)}}|} \sum_{c \in C^{(r)}} |\mathcal{D}_c| \boldsymbol{w}_c^{(r+1)}, \tag{2.12}$$

where $\boldsymbol{w}^{(r+1)}$ represents the newly aggregated server model, $|\mathcal{D}_c|$ represents the number of samples on device c, and $|\mathcal{D}_{C^{(r)}}|$ is the total number of processed samples in the round.

Step 5 Lastly, a new round starts with Step 1 ($\boldsymbol{w}^{(r+1)} \rightarrow \boldsymbol{w}^{(r)}$).

This dissertation mainly focuses on *synchronous* FL, as done in [27]. In synchronous FL, the server expects all devices selected for participation in a round to return their trained parameters within a limited time, as the aggregation in eq. (2.12) requires all updates to be present on the server. Hence, if a device takes longer than others, the FL process is bottlenecked by the slow device. Alternatively, if waiting is infeasible, the device must be dropped from the current round, essentially wasting training resources. Devices that slow down the synchronous rounds are labeled *stragglers*. However, there are also asynchronous implementations of FL [47, 48, 49]. In asynchronous implementations, devices can, in principle, submit updates at any time, as each update is aggregated into the server model using a weighted average of the new update and the current server model. However, if a device takes too long to deliver its update, the server model might be updated several times, causing the device to submit an update based on an outdated state of the server model. These kinds of updates are referred to as *stale* updates, which generally reduce the convergence speed and can affect the maximum reachable accuracy [49].

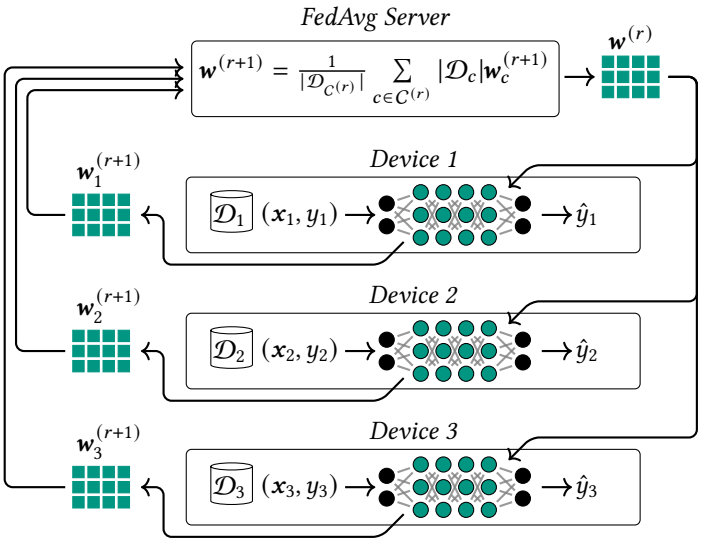


Figure 2.1: Detailed steps of FedAvg with three devices.

2.2.1 Data Distribution in Federated Learning

While in centralized training, data for training is centrally available as introduced in section 2.1, in FL, data is distributed such that each device c has its own dataset $\mathcal{D}_c$, and hence owns its data samples exclusively, such that $\mathcal{D}_i \cap \mathcal{D}_j = \emptyset$ for $i \neq j$ and $i, j \in C$. The following presents three types of distributions:

Iid: In the case of independent and identically distributed (iid) data, it is assumed that all devices sample from the same underlying data-generating distribution P, i.e.,

$$\mathcal{D}_c \sim P(\boldsymbol{x}, y) \quad \forall c, \tag{2.13}$$

thereby ensuring that, while each sample is unique, the underlying distribution of $\boldsymbol{x}$ and y remains the same across all devices. However, typically in FL, this is not the case; rather, data is non-iid or rc-non-iid.

Non-iid: In the non-independent and identically distributed case, the prior assumption does not hold, such that

$$\mathcal{D}_i \sim P_i(\boldsymbol{x}, y) \quad \text{where} \quad P_i(\boldsymbol{x}, y) \neq P_j(\boldsymbol{x}, y) \quad \text{for} \quad i \neq j, \quad i, j \in C. \tag{2.14}$$

As the underlying data-generating distributions $P_i(\boldsymbol{x}, y)$ and $P_j(\boldsymbol{x}, y)$ differ, the distributions of $\boldsymbol{x}$ and y can be different across devices. For brevity, this introduction focuses on two kinds of examples.

Feature distribution skew: In this case, the distributions $P_i(\boldsymbol{x}, y)$ and $P_j(\boldsymbol{x}, y)$ vary in the distribution of the features (i.e., the distribution of $\boldsymbol{x}$), such that $P_i(\boldsymbol{x}) \neq P_j(\boldsymbol{x})$, while the distribution of the labels y remains the same.

Typical examples in an object classification task include devices with differences in data collection conditions, i.e., the data samples (images) collected are brighter, or, in the case of detecting handwritten digits, some devices have samples written with a red marker instead of black.

Label distribution skew: Conversely, in the label distribution skew case, $P_i(y) \neq P_j(y)$, which means that some devices have more examples of specific classes, while others have more samples of different classes. Similarly, such a scenario can manifest if devices have different experiences during data collection. For example, in an object classification task, certain classes may appear significantly more often on some devices.

Lastly, there are mixtures of feature and label distribution skew, and changes in the distributions over time, which are generally referred to as concept drift.

Rc-non-iid: If devices participating in FL, i.e., their capabilities to perform training, vary, their impact on the global model also varies, leading to a specific kind of non-iid distribution called rc-non-iid [1]. In this case, the devices' capabilities (details about varying device capabilities are provided in section 2.3.1) induce differences in their data-generating distributions. Therefore, devices with the same capabilities sample from the same distribution, while these devices do not sample from the same distribution as devices with fewer or greater resources.

We introduce G disjoint subsets of devices $\tilde{C}_g$ with $g \in [1, \ldots, G]$ to represent groups of devices with different training capabilities (i.e., resources). An rc-non-iid distribution can be expressed as follows:

$$\mathcal{D}_i \sim P_g(\boldsymbol{x}, y) \quad \forall i \in \tilde{C}_g, \tag{2.15}$$

$$P_i(\boldsymbol{x}, y) \neq P_j(\boldsymbol{x}, y) \quad \text{for} \quad i \neq j, \quad i, j \in [1, \ldots, G]. \tag{2.16}$$

As a consequence of this definition, if devices are within a subset g, $P_i(\boldsymbol{x}, y) = P_j(\boldsymbol{x}, y)$ if $i \in \tilde{C}_g \wedge j \in \tilde{C}_g$, essentially behaving like iid within their group. However, devices that are in different groups exhibit non-iid behavior, such that $P_i(\boldsymbol{x}, y) \neq P_j(\boldsymbol{x}, y)$ if $i \in \tilde{C}_g \wedge j \notin \tilde{C}_g$.

To highlight the differences between the three types of distributions, we plot the distribution of samples of specific classes over the devices C in fig. 2.2, where a total of 10 classes exists, and $|C| = 15$ devices exist. For non-iid and rc-non-iid, label-skew distributions are generated by sampling from a Dirichlet distribution with a Dirichlet alpha value of $\alpha = 0.5$. A lower α value typically results in a higher degree of label imbalance. Devices are additionally grouped into three disjoint subsets $\tilde{C}_1$, $\tilde{C}_2$, and $\tilde{C}_3$, with $[0, \ldots, 4] \in \tilde{C}_1$,

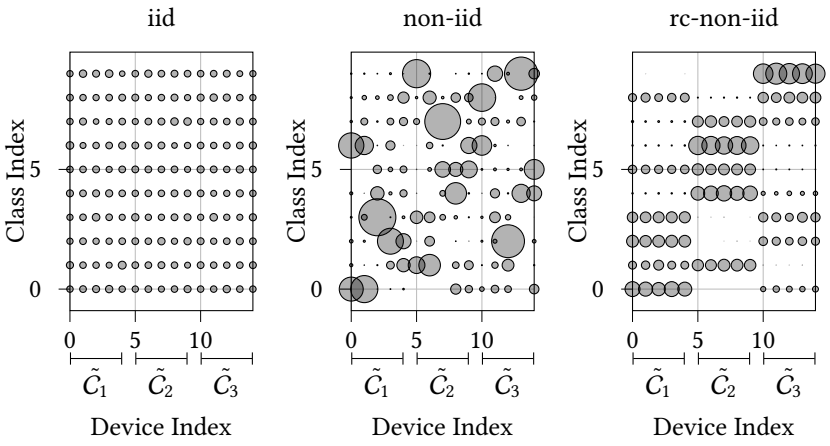


Figure 2.2: Visualization of iid (left), non-iid (center), and rc-non-iid (right) data splits in FL. The horizontal axis represents the device index of $c \in C$. The vertical axis represents the class index (values in $\mathcal{Y}$). The radius of the circles represents the quantity of samples of a specific class present on a specific device.

$[5, \ldots, 9] \in \tilde{C}_2$, and $[10, \ldots, 14] \in \tilde{C}_3$. In fig. 2.2, the horizontal axis refers to the device indices, while the vertical axis refers to the class indices. The radius of the circles refers to the relative number of samples in the device's local dataset $\mathcal{D}_i$. While the distribution of classes can vary, in this example, the total number of samples is the same for all devices (i.e., $|\mathcal{D}_i| = |\mathcal{D}_j|$ for all $i, j \in C$). It can be seen from fig. 2.2 that, in the case of iid, every device has roughly the same number of classes, while in non-iid, the distribution of classes is randomly distributed over all devices. Only in the case of rc-non-iid is a strong correlation between the three groups and the devices' data visible.

2.2.2 Performance Metrics in FL

Throughout this thesis, to evaluate the techniques, different performance metrics are used that typically assess how well the trained (or distributedly trained) NN model generalizes to the unseen test set $\mathcal{D}_{\text{test}}$. To this end, three different metrics are introduced, namely the *accuracy*, the *F1-score*, and the *group-sensitivity*.

Accuracy: The accuracy measure is the most basic performance metric and is primarily used when the classes in the dataset are balanced, i.e., the number of samples for each class is the same. Accuracy can be calculated using

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}, \tag{2.17}$$

where TP, TN, FP, and FN refer to the number of true positive, true negative, false positive, and false negative samples, respectively.

F1-score: In the case of unbalanced data, solely relying on the accuracy measure can be misleading. For example, in a binary classification problem where the test set consists of 95% of samples from class 0 and only 5% from class 1, always predicting class 0 (and ignoring class 1) results in an accuracy of 95%, which is misleading. In cases where data is unbalanced, the F1-score is used, which is the harmonic mean of precision and recall. The F1-score can be calculated as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{F1-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{2.18}$$

Group-sensitivity: In cases where data is rc-non-iid in FL, we assess whether devices within a specific group can impact the server model, i.e., we measure if the server model performs well on the group-specific distribution of classes. We introduce sensitivity_j as the sensitivity (recall) of a model on a specific class j, using

$$\text{sensitivity}_j = \frac{\text{TP}_j}{|\mathcal{D}_{y=j}|}, \tag{2.19}$$

where TP_j represents the number of correctly classified samples with label j, and the number of samples with class j in dataset $\mathcal{D}$ is calculated as $|\mathcal{D}_{y=j}| = |\{(\boldsymbol{x}, y) \in \mathcal{D} \mid y = j\}|$. Lastly, we define the metric *group-sensitivity* of group g as the group-weighted per-class sensitivity, using

$$\text{group-sensitivity}_g = \frac{1}{|\mathcal{D}_g|} \sum_{j=0}^{|\mathcal{Y}|-1} |\mathcal{D}_{g,y=j}| \cdot \text{sensitivity}_j, \tag{2.20}$$

where $|\mathcal{Y}|$ refers to the total number of classes, and $\mathcal{D}_g$ refers to the group dataset, as introduced in eq. (2.15). Throughout this dissertation, a global NN model is considered *fair* when it performs well on all groups' data distributions, specifically when there is group-sensitivity parity among the groups [1]. A global NN is considered unfair if there is a large variance in the group-sensitivities. Besides this parity-based fairness definition, there also exist different notions of fairness [50].

2.3 Computational Resources in FL

2.3.1 Computational Resources in Edge Devices

In the case of *cross-device* FL, most of the data available for training resides on resource-constrained edge devices. Table 2.1 lists the floating-point operations per second (FLOPS) of typical classes of devices alongside the RAM each device is equipped with. Table 2.2 presents the floating-point operations (FLOPs) required for the forward pass during training with a minibatch size of 32. To perform training, a backward pass—requiring

Table 2.1: Computational resources in edge devices [FLOPS and RAM].

Edge-device	FLOPS	RAM
MSP430 Ser.	10^5 - 10^6	0.5 kB - 66 kB
STM32F7 Ser. (ARM Cortex-M7)	$2 \cdot 10^8$	256 kB - 512 MB
Raspberry Pi Ser.	10^8 - 10^{10}	512 MB - 8 GB
Low-End Smartphones	10^{10} - 10^{11}	1 GB - 2 GB
NVIDIA Jetson Nano	10^{11} - 10^{12}	2 GB - 4 GB
High-End Smartphones	10^{11} - 10^{12}	4 GB - 8 GB
Server GPUs (e.g., NVIDIA V100, A100)	10^{13} - 10^{14}	32 GB - 100 GB

approximately 2× the number of FLOPs—is also necessary. Additionally, the peak RAM usage during training, which includes storage of parameters, temporary layer activations, and the optimizer state is reported. Table 2.2 demonstrates that training different NN topologies can demand varying resources from a system.

For example, a low-power MSP430 microcontroller, as equipped in sensor platforms, lacks the processing power and, especially, the memory required to train even low-complexity NN models such as LeNet [9]. Therefore, training on such platforms is considered infeasible. However, many NN models, as listed in Table 2.2, are suitable for training on platforms such as Raspberry Pis, NVIDIA Jetson Nanos, and smartphones.

2.3.2 Computational Heterogeneity in FL

Nevertheless, even though these platforms typically suffice for training, there is a significant degree of heterogeneity among these devices. For example, within the class of smartphones, computational performance (FLOPS) can vary between 10^{10} and 10^{12}, while RAM ranges from 512 MB to 8 GB. Particularly in the context of FL, the heterogeneity of devices may arise from multiple sources:

- Different devices with varying hardware participate in a single FL system. For instance, multiple revisions or generations of devices may be involved. Additionally, some devices might be equipped with custom ML accelerators, such as ASICs or GPUs, while others rely solely on a generic CPU for training. Furthermore, different software revisions can introduce heterogeneity into the FL system.
- Devices may have different power sources, i.e., wall-plugged or battery-powered, which limits the power available for training [8].
- The degradation of components over time can impact a device's training capabilities. A common example is battery fading, which gradually reduces the energy capacity and peak power capabilities of a device [51, 52]. As a consequence, affected devices must operate their processors at lower clock speeds.

Table 2.2: Requirements for NN training [FLOPs and peak RAM].

NN Model	# FLOPs (Forward)	peak RAM (Training)
LeNet [9]	$13.4{\cdot}10^5$	0.5 GB
ResNet18 [18]	$11.2{\cdot}10^8$	0.8 GB
EfficientNet [55]	$6.4{\cdot}10^7$	0.9 GB
MobileNetV2 [56]	$19.2{\cdot}10^7$	1.4 GB
ResNet152 [18]	$7.4{\cdot}10^9$	5.3 GB

- Ambient temperature affects a device's cooling capabilities. The effectiveness of cooling limits thermally safe power dissipation, thereby impacting a device's ability to perform training [53].
- Lastly, devices may have other applications running in parallel with training. In many cases, FL training is considered a secondary task that utilizes excess resources on the platform [54]. Consequently, there is a time-varying degree of contention for shared resources such as CPU time or RAM [6].

2.3.3 Categorization of Constraints

Based on the constraints faced by devices in FL training, as listed in Section 2.3.1, and the participation requirements in FL, constraints can be categorized into two types:

Hard constraints: These constraints completely prevent a device from training a given NN structure. A common hard constraint is the available memory. Despite efforts to shrink the parameters in typical NN structures [56, 55, 18], model architectures still require storing millions of parameters in high precision during training. Furthermore, during backpropagation, many layer activations must be retained in memory, which can easily accumulate to 1 GB, as listed in Table 2.2. In most cases, activation memory constitutes the majority of the memory required for training. Although techniques exist that trade some peak memory for additional computation [57], large peak memory requirements that exceed the device's system memory prevent devices from participating in an FL system. Similarly, severe memory contention that reserves significant portions of the available memory can have the same effect.

Besides RAM, if a synchronous FL server cannot tolerate the slowdown of a straggler device caused by low training throughput, the device must be dropped. Consequently, heterogeneity that causes slowdowns in some devices can result in hard constraints.

Soft constraints: These constraints do not fully prevent the training of a given NN structure, but hinder the overall training throughput, causing slowdowns as the FL server must wait for the device to finish its update. Reasons for such slowdowns can include factors such as the microarchitecture used, component degradation, limited power, or resource contention. Soft constraints in FL produce stragglers, as devices may be unable to complete their training within the given time.

2.3.4 Categorization of Heterogeneity

Soft and hard constraints of devices can cause different types of heterogeneity, namely heterogeneity across devices and heterogeneity across rounds or time.

Heterogeneity across devices: Devices participating in an FL system may have different hard and soft constraints, resulting in heterogeneity across devices. Some of these constraints may be known at design time, such as processor type, accelerator availability, cooling capacity, and peak power considerations. These constraints can be considered static and, therefore, do not change over time. As discussed in Section 2.3.1, devices within the same category can have vastly different capabilities. For example, a low-end smartphone may perform training at only 1/100th of the throughput of a high-end smartphone while relying on just 1/8th of the memory. These differences impact the training capabilities in an FL system.

Heterogeneity across rounds: Many soft constraints are not known at design time, as they are induced by the device's environment and vary across rounds. This is the case for constraints caused by the current battery level, power supply, and ambient temperature. These include all constraints that typically change more slowly than the FL training rounds (which range from minutes to hours) and can be accurately predicted prior to the upcoming round.

Heterogenity across time: Lastly, there are cases where soft constraints change within an FL round and, therefore, are not known prior to the round. An example of such a dynamic constraint is resource contention on a smartphone, which is influenced by the user's usage profile. This profile is typically unpredictable in advance and, therefore, must be modeled as random.

2.3.5 Scale and Granularity of Heterogeneity

Lastly, heterogeneity across devices, rounds, and time in an FL system can result in different scales and granularities within the FL system.

Scale of heterogeneity: Heterogeneity across devices, rounds, and time can manifest at different scales. The scale typically varies according to the constraints. For example, in smartphone applications, peak performance (training throughput) and memory can vary by factors of approximately 100× and 8×, respectively. Conversely, constraints caused by battery fading may result in a peak power reduction of 50%, which roughly corresponds to a 20% decrease in throughput, assuming a cubic relationship between dynamic power and training throughput ($V^2 f$ scaling[1] [58]). This reduction is significantly smaller than the variations observed in memory.

Granularity of heterogenity: Heterogeneity present in an FL system can manifest with different granularity. In a smartphone use case, where devices are equipped with varying memory capacities, the differences often reduce to a small, finite number of device types (e.g., smartphones equipped with 1, 2, and 4 GB of RAM). Conversely, devices experiencing resource contention can exhibit a continuous range of total training throughput within an FL round.

In summary, devices participating in FL can have various capability differences when performing training. Devices are typically constrained in their memory, throughput (FLOPS), energy, and cooling and experience resource contention. These device constraints can be considered hard constraints, i.e., constraints that completely prevent a device from training, or soft constraints that slow down the overall FL training process. Moreover, heterogeneity in devices can be static or dynamic, i.e., constant or changing over rounds or time. Lastly, heterogeneity among devices can manifest at different scales and granularities.

[1] V refers to voltage and f to frequency.

3 Related Work and Baselines

This chapter discusses related work and existing baselines. In section 3.1, existing work with respect to heterogeneous constraints in FL is presented. Techniques that are the closest to chapters 5 to 8 are discussed in more detail, specifically subset-based techniques. Moreover, it defines trivial baselines for comparison in chapters 5 to 8. Lastly, this chapter presents related work relevant to chapters 5 to 8, such as quantization and freezing of layers, memory-constrained training, layer-wise training, energy modeling, and hardware for training acceleration.

3.1 Communication, Computation, and Memory Constraints in FL

Existing works cover certain aspects of communication, computation, and memory constraints in FL; however, all three constraints are not considered jointly and cannot be adapted to independently. A large branch of works focuses on soft constraints, i.e., communication and computation overhead.

Communication overhead in FL is tackled by several works [59, 60], using compression, quantization, and sketched updates to lower the communication burden in FL. However, while communication is saved, computation and memory requirements remain the same. Another branch of works solely focuses on the time aspect of computation, i.e., covering straggler mitigation. Several of these works generally maintain a synchronous aggregation scheme with a tiering of clients [61, 62, 63]. Other works allow variable-length rounds [64, 65].

FedProx: FedProx [66] allows for partial work, i.e., some devices train fewer minibatches to tackle the straggler problem. While this reduces the total computations performed, it does not address communication or memory constraints. Naturally, as a result, devices with fewer updates contribute less to the aggregated model. Besides allowing partial contribution within a round, FedProx introduces a proximal term within the local training, such that instead of minimizing $L(F(\boldsymbol{x}_i, \boldsymbol{w}), y)$, FedProx minimizes the loss L with a proximal term:

$$L(F(\boldsymbol{x}_i, \boldsymbol{w}_c^{(r)}), y) + \frac{\mu}{2}\|\boldsymbol{w}_c^{(r)} - \boldsymbol{w}^{(r)}\|^2, \tag{3.1}$$

where $\boldsymbol{w}_c^{(r)}$ represents the weights of device c in round r, and $\boldsymbol{w}^{(r)}$ represents the weights received from the server. The hyperparameter μ serves as a mixing parameter between the local loss of F and the proximal term. By introducing the proximal term in the optimization target, FedProx aims to maintain similarity between the server model and the locally trained model, thereby tackling the impact of non-iid data.

Yang et al. [67] study the energy trade-off between computation and communication in FL; however, they cannot adjust to per-device communication and computation constraints.

Asynchronous or semi-synchronous FL is also considered to tackle soft constraints, specifically to mitigate the straggler problem [47, 49, 48, 68, 69]. In asynchronous FL, devices cannot become stragglers, as their updates can be incorporated into the server model at any time. However, devices that upload their parameters based on an already outdated server model provide a stale model update, which can lower accuracy and convergence speed [49, 70].

There also exists a branch of works [46, 71, 72, 73, 74, 75] that enable devices to train different NN topologies, which do not derive from FedAvg. Specifically, many techniques utilize knowledge distillation [76]. In these techniques, devices do not share knowledge through the exchange of weights but instead distill their knowledge into soft labels, i.e., output distributions of a shared common dataset. While from a resource perspective, distillation-based techniques decouple the knowledge-sharing mechanism from the NN structure, in general, they cannot keep up with FedAvg-derived techniques in terms of accuracy, convergence speed, and communication overhead [3].

3.1.1 Subset-Based Techniques

Subset-based techniques [77, 78, 79, 80, 70, 6], in principle, allow addressing hard constraints such as communication, computation, and memory constraints; hence, they are explained in more detail. However, all presented techniques share the limitation that the reduction of communication, computation, and memory is coupled, i.e., adjusting one directly affects the others.

Federated Dropout (FD) [77]: Caldas et al. were the first to tackle the high resource requirements imposed on FL devices when performing end-to-end training of typically used NN structures. To lower the resources required for training, they proposed a technique called Federated Dropout (FD). Specifically, FD constructs a subset of the parameters using a scale factor $s \in (0, 1]$, such that instead of training the full set of parameters $\boldsymbol{w}$ of a layer, the parameters are reduced using s

$$\overline{\boldsymbol{w}} \in \mathbb{R}^{\lfloor sP \rfloor \times \lfloor sV \rfloor}, \tag{3.2}$$

where $\overline{\boldsymbol{w}}$ represents a downscaled version of $\boldsymbol{w}$ with size $\lfloor sP \rfloor \times \lfloor sV \rfloor$, and P and V represent the dimensions of the parameters (higher-level dimensionality is omitted for brevity).

Scaling P and V by s means that a linear decrease in s results in a quadratic reduction in parameters. Using s, a device that participates in FL can adjust its computational, communication, and memory overhead by modifying the scaling value s. In FD, the specific output channel indices used to create a subset are selected randomly. The set of output indices that refer to a specific layer k in a specific round r for a specific device c can be labeled as $\mathcal{I}_k^{(r,c)}$. As this set reflects the downscaled output channels, its size is determined by $|\mathcal{I}_k^{(r,c)}| = V_k$. Similar to layer-wise channel pruning, as performed in inference optimization, a layer's input indices are selected to match the previous layer's output indices. The input layer's (f_1) input channels remain unscaled, ensuring that all channels from the input (e.g., the input image) are fed into the downscaled NN.

By using random output indices for each round, each layer, and each device, eventually all parameters on the fully-sized server NN receive training. Therefore, FD allows training larger-scale server NNs, even though no participating device is able to train the NN end-to-end. In case all devices have the full capabilities to train the NN end-to-end, FD becomes algorithmically identical to FedAvg.

While heterogeneous devices are not considered by Caldas et al., scaling s based on individual devices' resources is a natural extension of this technique.

HeteroFL [78]: Diao et al. were the first to support heterogeneous devices in such a subset scheme. Unlike FD, in HeteroFL, a device with a specific resource constraint always trains the same subset of parameters. Consequently, HeteroFL requires that there exists a device capable of training the NN end-to-end; otherwise, a portion of the parameters does not receive any training. In HeteroFL, each device c picks a subset based on a resource-specific scale factor s_c, such that

$$\mathcal{I}_k^{(r,c)} = \{i \mid 0 \leq i < \lfloor s_c V_k \rfloor\}. \tag{3.3}$$

As a consequence of this selection scheme, a portion of the parameters is trained by all devices, while some parameters are only trained by devices that train the NN end-to-end.

FjORD [79]: Horváth et al. proposed a subset scheme similar to HeteroFL, using the same indices as HeteroFL. Unlike HeteroFL, a device switches between constraint-specific subsets that are within its capabilities, rather than training a fixed-size subset. Devices switch randomly between these subsets. Moreover, each subset uses its own subset-specific BN layers. To ensure that all subsets receive sufficient local training, the number of differently sized subsets in a heterogeneous case must be limited.

FedRolex [80]: Alam et al. address the shortcomings of HeteroFL and FjORD, specifically the requirement that a share of devices must train the server NN end-to-end. They achieve this by using a rolling window scheme, i.e., not randomly as done in FD. Additionally,

all devices with the same capabilities train the same channels. The scheme used can be described by

$$\mathcal{I}_k^{(r)} = \begin{cases} \{\hat{r}, \hat{r}+1, \ldots, \hat{r}+\lfloor sV_k \rfloor - 1\} & \text{if } \hat{r}+\lfloor sV_k \rfloor \leq V_k, \\ \{\hat{r}, \hat{r}+1, \ldots, V_k - 1\} \cup \{0, \ldots, \hat{r}+\lfloor sV_k \rfloor - 1 - V_k\} & \text{otherwise,} \end{cases} \tag{3.4}$$

where $\hat{r}$ is the remainder of r mod V_k. The dependence of eq. (3.4) on the FL round r ensures that the rolling window is shifted by 1 each round. Through this shift, (eventually) all parameters receive training, despite the fact that no device can fully train the server NN end-to-end.

DepthFL [81]: In DepthFL, constrained devices use an early exit of the NN with a specific head to reduce the computational complexity of training. More capable devices use deeper exits that incorporate more layers and additionally use the Kullback–Leibler divergence to align the outputs of different heads. Thus, unlike [77, 82, 78, 80, 79], the NN is not scaled in its width but rather in its depth.

FedHM [82]: In FedHM, a lower-complexity submodel is created prior to the round by applying a low-rank operation directly on the NN's parameters. This submodel is created by using singular value decomposition (SVD) to decompose $\boldsymbol{\theta}$ into $\boldsymbol{\theta}_1$, $\boldsymbol{S}$, and $\boldsymbol{\theta}_2$, such that $\boldsymbol{\theta}_1, \boldsymbol{S}, \boldsymbol{\theta}_2 = \text{svd}(\boldsymbol{\theta})$. Using the rank u, a lower-rank version of $\boldsymbol{\theta}$ can be created as $\boldsymbol{\theta}_1 \cdot \text{diag}(\boldsymbol{S}) \cdot \boldsymbol{\theta}_2$, where $\boldsymbol{\theta}_1 \cdot \text{diag}(\boldsymbol{S}) \in \mathbb{R}^{P \times u}$ and $\boldsymbol{\theta}_2 \in \mathbb{R}^{u \times V}$. Participating devices use the lower-rank version of $\boldsymbol{\theta}$ for training. On the server, the parameters are reconstructed as $\boldsymbol{\theta} \sim \boldsymbol{\theta}_1 \cdot \text{diag}(\boldsymbol{S}) \cdot \boldsymbol{\theta}_2$. Heterogeneity in computational complexity is achieved by having a device-specific u. Less constrained devices utilize a larger u, whereas more constrained devices scale down u.

Heterogeneous LoRA [83]: In the context of transformers, especially with Large Language Models (LLMs), LoRA [84] is a popular technique to fine-tune LLMs for specific downstream tasks. In LoRA, the main structure of the NN remains frozen, while low-rank adapters are applied additively to the NN's linear layers. For example, with LoRA, a linear layer of the base NN, defined by its parameters $\boldsymbol{\theta} \in \mathbb{R}^{P \times V}$, is modified to

$$\boldsymbol{z} = \boldsymbol{x}(\boldsymbol{\theta} + \boldsymbol{\theta}_1 \boldsymbol{\theta}_2) + \boldsymbol{b} \tag{3.5}$$

for training, where $\boldsymbol{\theta}_1$ maps from P to a defined lower rank u, and $\boldsymbol{\theta}_2$ maps from u to V.

For fine-tuning, only the additive adapters are trained, while the base NN structure remains frozen. Compared to techniques like HeteroFL or FjORD, it is required that the base NN is pretrained. Recently, a version for heterogeneous FL with LLMs has also been proposed [83]. In this case, the low-rank adapters vary in their rank u, such that the respective constraint is met. The averaging of the differently sized adapters is done similarly to HeteroFL. While LoRA can be very communication efficient, in cases where

activation memory dominates over the weights' contribution to the memory footprint, LoRA is unable to adhere to memory constraints.

3.1.2 Naive Baselines

Besides the existing techniques that tackle communication, computation, and memory constraints to some extent, there also exist several naive baselines that bound the achievable accuracy and fairness properties.

FedAvg (upper bound): FedAvg (upper bound) represents a hypothetical scenario where all participating devices have the full resource requirements to train a given NN end-to-end. Consequently, no heterogeneity with respect to the resources is present among the devices. This serves as an upper bound for any algorithm incorporating resource heterogeneity in FL.

Dropping of devices: Dropping of devices represents a baseline for any algorithm that incorporates heterogeneously constrained devices in FL and is known to be used in production [40]. It is assumed that there are at least two groups of devices: strong and constrained. Strong devices are capable of training the NN end-to-end, whereas constrained devices cannot. In each round, only devices from the strong group are selected for participation; others are ignored and never participate.

Further, it is assumed that training the FL model with the strong devices is sufficient to achieve an acceptable accuracy. However, if data is scarce, i.e., the data present on the strong devices is not sufficient to achieve high accuracy, or especially if the data is rc-non-iid, as presented in section 2.2, dropping devices causes low accuracy and bias towards the strong devices' data. Any technique incorporating constrained devices into a resource-heterogeneous FL system must improve accuracy in general or fairness relative to this lower bound.

Small (homogeneous) model: A small homogeneous model serves as a lower bound for any algorithm that incorporates heterogeneously constrained devices in FL. Based on the available resources, a small model is selected such that even the most constrained device can perform training. As a consequence, stronger devices are not fully using their capabilities, as they are limited by the most constrained device. A small model can be constructed similarly to scaling down the NN, as done in HeteroFL or FjORD. The selected indices $\mathcal{I}_k^{(r,c)}$ of the subset stay the same throughout the training, independent of the device capabilities, such that

$$\mathcal{I}_k^{(r,c)} = \mathcal{I}_k = \{i \mid 0 \leq i < \lfloor sV_k \rfloor\}. \tag{3.6}$$

Any technique that incorporates constrained devices into a resource-heterogeneous FL system should enable higher accuracies than selecting a model that can be homogeneously trained by all devices.

3.2 Quantization and Freezing in ML Training

Quantization of NNs has most prominently been used for inference. Including quantization in training has only recently gained popularity, as it cannot be as straightforwardly applied. Naive training with quantized parameters leads to stagnation in training, as gradients are rounded to zero [85]. There are works that apply stochastic rounding to prevent this [86], however, the final accuracy is still reduced. Stochastically applied freezing [87] has been used to speed up the training of NNs, however, not systematically to lower computational complexity. Chen et al. [88] detect specific parameters that have stabilized in FL training, freeze them, and exclude them from being uploaded to the server to reduce communication. However, this scheme does not lower the computation or memory footprint in training. Recently, Yang et al. [89] proposed freezing layers to achieve communication and memory savings; however, they do not exploit the potential of quantization in the frozen layers, nor do they focus on computation.

In summary, quantization is mostly applied in inference and has not been used in combination with freezing to lower computational complexity. Specifically, the combination of both has not been exploited in distributed scenarios, such as in FL.

3.3 Memory-Constrained Training

In centralized training, several techniques can be used to lower memory usage. Kirisame et al. [90] present Dynamic Tensor Rematerialization, which enables recomputing activation maps on-the-fly, so some activation maps do not have to be stored throughout the forward and backward passes. Encoding [91, 92] and compression [93] have been proposed to lower the memory footprint during training. All these techniques trade memory for computation or accuracy. In the case of FL, such techniques can be applied orthogonally to resource-specific FL algorithms.

3.4 Layer-Wise Training

Layer-wise NN training has been proposed for the training of convolutional NNs as an alternative to end-to-end error backpropagation. Hettinger et al. [94] propose adding convolutional layers one by one to the training using auxiliary heads and freezing early layers to save memory by reducing the need to store layer activations. Specifically, in

unsupervised learning, layer-wise learning with contrastive loss has been explored [95, 96]. Recently, progressive model growth [97] has also been explored in the context of FL, mainly to lower communication overhead. However, eventually, devices in FL must train the full model, thus, no memory reduction is achieved. Similarly, Kundu et al. [98] grow the trained NN during FL training, based on the data classification complexity.

All these techniques do not enable heterogeneously memory-constrained devices in FL to participate, and most require that, eventually, devices in FL train the full NN end-to-end. Hence, they would exclude memory-constrained devices.

3.5 Energy Modeling in FL

Existing works that study the energy consumption of FL training mostly focus on the communication aspect. Tran et al. [99] study the energy trade-offs between computation and communication in FL; however, only CPU cycles and frequency are considered, neglecting the costs of memory accesses. Prior works that tackle soft and hard constraints [78, 79, 80] model heterogeneity solely based on the number of MACs, FLOPs, and the number of parameters in an NN, but do not consider energy.

3.6 Accelerators for On-Device Training

Most research focusing on accelerators for edge devices concentrates on inference, usually employing `int8` arithmetic, which allows for low accuracy degradation in inference compared to full precision (`float32`). In recent years, there has also been a focus on on-device training on edge devices [100, 101, 102]. However, while `int8` is sufficient for most inference applications, the calculation of gradients usually requires higher precision, and training cannot be achieved with `int8` without many modifications. Apart from GPUs, which utilize the single instruction, multiple data (SIMD) design, systolic array-based ASICs have been proposed [101, 103]. SAs can outperform typical GPUs with respect to energy, as they are more purposefully built for matrix operations and have a more efficient data flow, reducing memory read and write operations [103]. Existing ASICs for training NN models have not been designed with consideration of FL.

4 Experimental Setup

4.1 Simulation of FL Training

Cross-device FL is usually conducted at a large scale, ranging from hundreds to millions of participating devices. These devices are data-generating, i.e., the data used for training in FL is created on the devices themselves. Evaluating FL techniques requires simulating the distributed process, as evaluation on a large number of deployed, data-generating devices is, in most cases, infeasible.

For that reason, the training data $\mathcal{D}_{\text{training}}$ of standard benchmark datasets, as presented in section 4.2, is separated into device-exclusive chunks $\mathcal{D}_c$ that represent the data generated on the FL device. The training data $\mathcal{D}_{\text{training}}$ can, thereby, be distributed in several ways as outlined in section 2.2. Prior to the split of $\mathcal{D}_{\text{training}}$, a chunk of the overall data $\mathcal{D}$ is separated to serve as a test set $\mathcal{D}_{\text{test}}$. The performance of the aggregated NN model is evaluated using a separate test set $\mathcal{D}_{\text{test}}$ on the server. Figure 4.1 depicts the individual parts of the simulation setup. Throughout this dissertation, the accuracy (eq. (2.17)) and group-sensitivity (eq. (2.20)) refer to the test set accuracy and group-sensitivity. PyTorch [104] and TensorFlow [105] are used to perform the training, where device concurrency is simulated by sequentially performing device training one by one, queueing the trained device models, and performing aggregation after all devices' training updates are provided. Usually, each individual FL experiment is repeated with several random seeds to ensure deterministic data distributions and repeatable random device selection. This allows for a fair comparison of techniques and ensures that the overall setup is deterministic and experiments are repeatable. Metrics such as accuracy, group-sensitivity, performed FLOPs, and communication footprint are evaluated on a round basis and logged into JSON files. All simulations are conducted using GPU clusters equipped with NVIDIA Tesla V100 and A6000 GPUs.

4.2 Datasets

The following vision datasets are used:

- CIFAR10 [106] is an image classification dataset that consists of 10 classes, where all classes have exactly the same number of samples. The data is split into a training

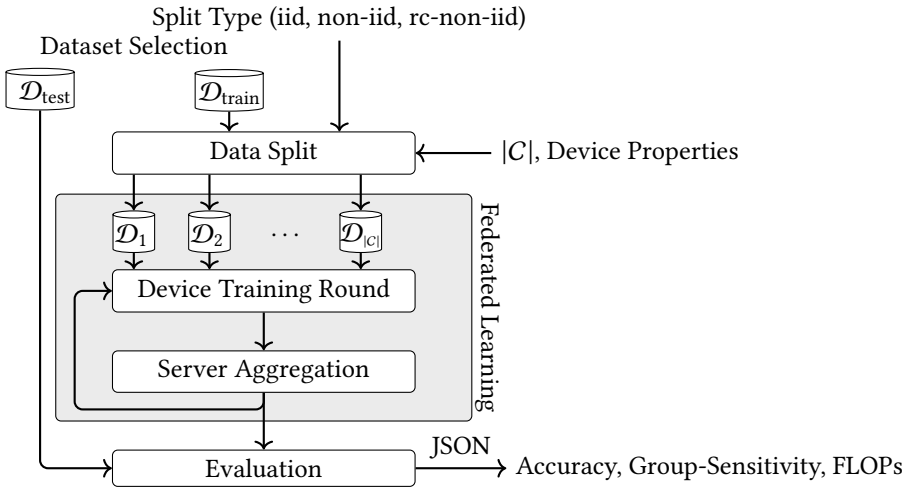


Figure 4.1: Simulation setup for the evaluation of FL-derived algorithms.

set containing 50 K images and a test set of 10 K images. Each data sample is an RGB image with a resolution of $3 \times 32 \times 32$.

- CIFAR100 [106] is a dataset identical to CIFAR10, with the only difference being the increased number of classes (i.e., 100 classes instead of 10). Therefore, the complexity of the dataset is higher, and the number of samples per class is lower.
- CINIC10 [107] is a dataset similar to CIFAR10; however, the classes are more complex, making it a more challenging training problem.
- FEMNIST [108] is a dataset that is part of the Leaf benchmark [108], specifically designed for the evaluation of FL algorithms. It consists of handwritten digits from `0-9`, handwritten uppercase letters from `A-Z`, and lowercase letters from `a-z`. However, compared to CIFAR, the classes are not balanced. Throughout this dissertation, a version with $640,500$ training samples and $160,129$ test samples is used. For better compatibility with CIFAR datasets, the grayscale images with a size of 28×28 are converted to RGB images with a resolution of $3 \times 28 \times 28$. FEMNIST provides a specific data split that can be used in the context of FL. This split partitions the data based on writers (e.g., data producers), such that each subset representing an FL device contains data associated with a specific writer of digits and letters. This split represents a feature-skew distribution, as presented in section 2.2.1.
- XChest [109] is a classification dataset consisting of chest X-ray images. Throughout this dissertation, a binary decision version is used where images are classified as either `healthy` or `unhealthy`. A version with $12,700$ training samples and $2,250$ test samples, with a resolution of $3 \times 256 \times 256$, is used.

- ImageNet [11] is a large-scale labeled dataset that consists of 1,000 classes and 1.4 M images for training and 50 K for testing. To make FL training feasible with such a large-scale dataset, images are downscaled to $3 \times 64 \times 64$.
- TinyImageNet [110] is a smaller-scale version of ImageNet, containing only 200 classes and 100 K samples for training and 1 K for testing. TinyImageNet uses a resolution of $3 \times 64 \times 64$.

The following natural language processing (NLP) problems are studied:

- IMDB [111] is a binary text classification dataset that contains movie reviews labeling a movie as either `good` or `bad`. To enable training on these text reviews, the individual reviews are capped at a sequence length of 512. The dataset used in this dissertation consists of 40 K training samples and 10 K test samples. The text is tokenized using a SentencePiece [112] tokenizer with a vocabulary size of 16,000.
- Shakespeare [108] is part of the FL benchmark Leaf and contains the written works of Shakespeare's plays. The data is used to perform next-character prediction (i.e., predict a character from `A-Z, a-z`). The text is split into 209,552 individual data-character target tuples for training and 21,399 for testing. The text is tokenized based on individual characters and capped at a sequence length of 80. Similar to FEMNIST, the Leaf benchmark provides a feature-skew distribution for Shakespeare, where data is split based on individual plays (a total of 25).
- OpenWebText [113] contains about 40 GB of text scraped from the internet. The text is gathered by extracting post URLs from Reddit. Text from the respective links is collected, and non-English text is filtered out.

4.3 NN Architectures

Throughout this dissertation, several common NN architectures are used for evaluation:

- ResNets [18] are a collection of image classification NNs that overcome the vanishing gradient problem of deep NN structures by adding shortcuts alongside blocks of many stacked NN layers. In ResNet, every convolutional layer is followed by a BN layer and a nonlinearity (ReLU). Between each block, the output of the previous block and the shortcut connection are added together. The default configuration of ResNet consists of four sections, where each section doubles the number of convolutional filters and, at the same time, reduces the feature map width and height by 2×. Each section can have a variable number of convolutional layers; for example, ResNet50 describes a ResNet with 50 layers, while ResNet18 and ResNet34 describe ResNets with 18 and 34 layers, respectively. There also exists an adjusted version for smaller image input sizes, such as CIFAR10, which features only three sections instead of four, hence having lower capacity (referred to as ResNet20,

ResNet44, and ResNet56). However, in principle, both versions can be adjusted for different image resolutions with small modifications.

- DenseNets [114] are similar to ResNets; however, instead of adding shortcuts between the sections, the shortcut path is concatenated to the section's output.
- MobileNet [56] provides an NN structure that is specifically optimized for efficient inference (latency and size) by utilizing depthwise separable convolutional layers.
- Transformers [12] utilize linear layers instead of convolutional layers as building blocks, making them more suitable for dense information input, such as tokenized text sequences. A key component of transformers is the self-attention mechanism, which allows each token to dynamically consider all other tokens in the sequence when producing its output representation. This enables the transformer to capture long-range dependencies between tokens. Recently, transformers have also been adapted to process images [115].

5 Communication-, Computation-, and Memory-Aware FL

5.1 Scope and Contribution

Devices in real-world systems are typically constrained in their ability to perform training, as outlined in section 1.1. These constraints manifest themselves in limited computational resources, memory, and communication capabilities, which hinder participation in FL. Moreover, these constraints often vary among devices. For instance, in a smartphone use case, many different types of smartphones can participate (i.e., devices from different vendors or devices from different hardware generations). Smartphones usually utilize a wireless connection for data transmission, which is prone to varying throughput due to effects such as channel fading [116].

In order to enable efficient learning on such devices and facilitate participation in FL systems, FL needs to adapt to these constraints, i.e., be *constraint-aware*. Although several techniques have been proposed to tackle this problem [78, 79, 77], the common approach in these works is to reduce the complexity by training a subset of the NN model on less capable devices, as outlined in chapter 3, thereby matching the resources required for training with the actual available resources. However, these techniques suffer from two weaknesses:

- Lowering the subset complexity by scaling s as done in eq. (3.2) tightly couples the communication overhead with the computation (FLOPs) and the peak memory requirements. Therefore, one cannot adjust to one constraint without adjusting the other. This coupling causes underutilization of resources, as a device has to choose s such that all constraints are satisfied. For instance, a device with a very good communication channel but limited computational resources has to pick a low s (so that it can deliver the update on time), thereby wasting available communication capabilities.
- While these techniques enable constrained devices to participate in the training, they do not allow devices to contribute effectively. Especially when fairness is evaluated, it is evident that subset-based techniques do not preserve fairness (i.e., group-sensitivity parity [50]) when evaluated with rc-non-iid data.

This chapter is mainly based on [1].

In this chapter, we propose a new technique, CoCoFL, that allows constrained devices to calculate gradients based on the full model, irrespective of their capabilities. This is enabled by the novel combination of partial freezing and quantization of the model on constrained devices. We show that quantizing frozen layers while keeping the trained layers at full precision enables a significant reduction in resource requirements, while still enabling effective learning on devices. Freezing layers reduces the need for gradient calculation for those layers and removes the need to store intermediate activations. Further, the quantization of frozen layers speeds up computation. Partial freezing and quantization open up a large design space, where each layer can be frozen or trained on a specific device, allowing for adjustment to communication, computation, and memory constraints. In summary, this chapter makes the following novel contributions:

- We empirically show that state-of-the-art subset techniques do not enable better accuracies than straightforward baselines (as introduced in chapter 3), i.e., simply excluding constrained devices from the training. This can be observed across various benchmarks, such as CIFAR10, XChest, and Shakespeare, and across various NN topologies, such as ResNet, DenseNet, and transformers.
- Partial freezing and quantization enable a large design space of training configurations to choose from, allowing devices to better adjust to individual communication, computation, and memory constraints, thereby utilizing available resources more efficiently.
- Compared to the state of the art, CoCoFL enables higher fairness of contribution (i.e., better group-sensitivity parity) among the devices.

5.2 Problem Statement

System model: The technique described in this chapter targets a synchronous cross-device FL setting, as outlined in section 2.2. It is assumed that there is a predefined, limited *round time* T, within which devices must upload their model updates to the server. All devices are assumed to have an equal number of data samples, and each participating device must iterate over its full local dataset within one round. Furthermore, the NN structure is assumed to be given and fixed. The server discards updates that arrive late (i.e., stragglers); therefore, devices must complete their training on time.

Device model: Participating devices are assumed to be heterogeneous with respect to their computation (i.e., FLOPS, throughput), available memory for training, and communication capabilities. The training throughput depends on various factors, such as the number of cores, the microarchitecture used, memory bandwidth, etc., as outlined in section 2.3. Furthermore, it depends on the chosen NN topology and the number of data samples processed per round (training configuration). Similarly, the available memory M_c of a device c depends on hardware properties, while the memory required for training

depends on the software and training configuration. While some training configurations are fixed (e.g., the NN topology), others, denoted as $A_c^{(r)}$, can be adjusted per device and per round. In this case, $A_c^{(r)} \in \mathcal{A}$ describes the subset of all NN layers trained in round r by device c, where $\mathcal{A}$ represents the set of all possible configurations. The time required to train the NN with a configuration $A \in \mathcal{A}$ is represented by $t_c : \mathcal{A} \to \mathbb{R}$. Similarly, the memory required for training is represented by $m_c : \mathcal{A} \to \mathbb{R}$. For the evaluation of this technique in section 5.5, m_c and t_c are determined experimentally through measurements (profiling); however, an analytical derivation is also feasible.

The communication between the server and devices is commonly asymmetric. Consequently, the download link from the server to the device can be neglected due to the high transmission power of base stations [67]. However, the upload link from devices to the server is usually subject to heterogeneity, as the channel quality between the base station and devices varies. The communication constraint $Q_c^{(r)}$ of a device c in round r is defined as a limited number of bits that can be uploaded to the server at the end of the round. All layers that are not in a specific $A_c^{(r)}$ remain frozen (and quantized); hence, they do not change and do not need to be uploaded to the server. Therefore, for any $A \in \mathcal{A}$, the function $q : \mathcal{A} \to \mathbb{N}$ can be derived based on the NN topology by counting the parameters per trained layer. Note that while q is device-independent, t_c and m_c may depend on device-specific hardware and software.

Objective: The main objective is to maximize the *final* accuracy (i.e., the accuracy of the server model w^R after R rounds) while adhering to round-varying communication, computation, and memory constraints by selecting, per device and per round, the set of trained layers $A_c^{(r)}$:

$$\begin{aligned} \text{maximize accuracy}(w^{(R)}) \quad & \forall c \in C, \quad \forall\, 1 \le r \le R \\ & \text{s.t. } t_c(A_c^{(r)}) \le T \wedge m_c(A_c^{(r)}) \le M_c \wedge q(A_c^{(r)}) \le Q_c^{(r)} \end{aligned} \tag{5.1}$$

As a secondary metric, group-sensitivity, as defined in eq. (2.20), is evaluated after R rounds.

5.3 Partial Freezing and Quantization

5.3.1 NN Structure and Training

Commonly used NN structures, such as ResNet [18], DenseNet [114], and MobileNet [56], follow similar repeating patterns, where a convolutional layer is followed by a BN layer and then by ReLU. While other types of layers also exist, this structure accounts for most of the training time. We define this repeating structure as a *block*, which we treat as the

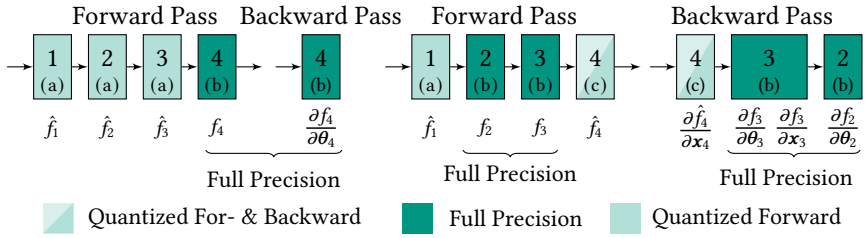


Figure 5.1: Visualization of blocks (a), (b), and (c) in CoCoFL. The left illustrates a configuration where only block 4 is trained, while blocks 1-3 ($\hat{f}_1, \hat{f}_2, \hat{f}_3$) remain frozen and quantized. The right shows a configuration where blocks 2 and 3 are trained, while blocks 1 and 4 remain frozen and quantized. Since block 4 remains frozen, a quantized backward pass is required.

smallest entity in the NN that is either trained or frozen. We denote by K the number of blocks in a given NN, such that the forward pass is computed as

$$\hat{y} = f_K(f_{K-1}(...(f_1(\boldsymbol{x})))), \tag{5.2}$$

where $\hat{y}$ represents the output, and $\boldsymbol{x}$ the input. The backward pass is computed accordingly using the chain rule eq. (2.10). For simplicity, this description follows the structure of ResNet; however, multiple skip connections or transformer-based architectures are also compatible with the proposed technique (see section 5.5).

5.3.2 Freezing, Fusion, and Quantization of Blocks

Freezing: Freezing a parameter removes the need to calculate a gradient with respect to that parameter. By the chain rule (eq. (2.10)), the number of required intermediate gradients (i.e., gradients with respect to inputs) depends on a block's index. For example, calculating $\frac{\partial L}{\partial \boldsymbol{\theta}_K}$ requires no intermediate gradients, whereas $\frac{\partial L}{\partial \boldsymbol{\theta}_1}$ requires intermediate gradients from all other blocks. Consequently, computing gradients for early blocks in an NN is computationally expensive, while training later blocks is relatively cheap. Based on per-block operations, a distinction can be made between three types of blocks, as illustrated in fig. 5.1.

(a) *Frozen block:* Without any preceding trained block, a frozen block requires only a forward pass $f_i(\boldsymbol{x})$.

(b) *Trained block:* Trained blocks require a forward pass $f_i(\boldsymbol{x})$, the calculation of gradients with respect to their parameters $\frac{\partial f_i(\boldsymbol{x}_i)}{\partial \boldsymbol{\theta}_i}$, and gradients with respect to the input $\frac{\partial f_i(\boldsymbol{x}_i)}{\partial \boldsymbol{x}_i}$.

(c) *Frozen block with backward pass:* If preceded by trained blocks, a frozen block requires a forward pass $f_i(\boldsymbol{x}_i)$ and gradients with respect to their input $\frac{\partial f_i(\boldsymbol{x}_i)}{\partial \boldsymbol{x}_i}$.

As a consequence, freezing blocks reduces the number of per-block operations for frozen blocks (from 3 to 2 or 1), thereby lowering the number of MACs required in the backward pass. Additionally, freezing blocks also impacts peak memory: If a block is not trained, the activation values $\boldsymbol{x}_i$ can be released from memory during the forward pass, as they are not required for training. Lastly, frozen layers do not need to be uploaded to the server, thereby reducing communication costs.

Fusion: As described earlier, common structures such as ResNet, DenseNet, and MobileNet use BN operations following the convolutional layer to act as a regularization layer. Since BN is a linear operation when input statistics remain constant, it can be fused with the preceding convolutional layer. This fusion is typically done as an optimization for inference, i.e., it is implemented in inference frameworks like ONNX [1]; however, its application for training is novel. In the case where the block is frozen (type (a)), statistics remain constant over time. Hence, the BN operation can be expressed as a linear operation, with $\boldsymbol{y}_{i_{BN}}$ representing the channel-wise output of the BN operation, $\boldsymbol{y}_{i_{\mathrm{CL}}}$ the convolutional layer's output, and ϵ a small constant for numerical stability

$$\boldsymbol{y}_{i_{BN}} = \frac{\gamma_i}{\sqrt{\sigma_i^2 + \epsilon}} \boldsymbol{y}_{i_{CL}} + \left(\frac{-\mu_i \gamma_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta_i \right) \cdot \mathbf{1}, \tag{5.3}$$

where the coefficient of $\boldsymbol{y}_{i_{CL}}$ represents a new combined scaling factor, and the second summation term represents a new combined bias. To fuse the two linear operators, the preceding convolutional layer (eq. (2.4)) can be expressed as

$$\boldsymbol{y}_{i_{BN}} = \left(\frac{\gamma_i (b_i - \mu_i)}{\sqrt{\sigma_i^2 + \epsilon}} + \beta_i \right) \cdot \mathbf{1} + \sum_{j=1}^{C_{\text{in}}} \left(\frac{\gamma_i}{\sqrt{\sigma_i^2 + \epsilon}} \cdot \boldsymbol{\theta}_{ij} \right) \star \boldsymbol{x}_j = \tilde{b}_i \cdot \mathbf{1} + \sum_{j=1}^{C_{\text{in}}} \tilde{\boldsymbol{\theta}}_{ij} \star \boldsymbol{x}_j, \tag{5.4}$$

where $\tilde{b}_i$ represents the new fused bias and $\tilde{\boldsymbol{\theta}}_{ij}$ represents the fused 2D kernel.

The same fusion can be applied to type (c) layers, with the main difference that the BN input statistics, μ and σ^2, are only valid for a limited number of training steps, as they are influenced by the preceding layers that receive gradient updates.

Using BN fusion during training, the forward pass of type (a) and type (c) blocks is simplified by fusing three operators, thereby reducing the number of operations.

Quantization: Quantization of operators in NN is a commonly used technique for inference to reduce computational resource requirements. In contrast, here, quantization is applied during training; however, only the frozen part of the NN is additionally quantized.

[1] `https://onnxruntime.ai`

This differs from quantization-aware training [86], as all trained parameters remain in full precision. Therefore, only type (a) and type (c) blocks receive quantization. Specifically, the previously fused convolution operation is performed in `int8` instead of the commonly used `float32`. Additionally, in type (c) blocks, the calculation of intermediate gradients in the backward pass is also quantized (e.g., the transposed convolution operation). By applying quantization in both the forward and backward pass, frozen and quantized blocks require less time for execution. However, while quantization reduces computational complexity, low-precision arithmetic introduces quantization noise in both the forward and backward passes of frozen layers, thereby affecting training. Additionally, updating the statistics of fused layers only at the beginning of the FL round introduces errors. The experimental evaluation, however, demonstrates that the benefits of increased efficiency with respect to training time outweigh the added noise. In summary, through freezing, fusion, and quantization of selected blocks, the computational complexity is significantly reduced, allowing less capable devices to compute parameter gradients of trained blocks in full precision based on the full NN structure. This novel combination has not yet been exploited for training.

5.3.3 Implementation in PyTorch

The proposed training scheme is implemented in PyTorch 1.10 [104], which supports `int8` quantization. However, conceptually, the proposed technique is not limited to a specific quantization scheme and could also be implemented with `int4` or `float16`. As of now, PyTorch provides the necessary operators only for `int8` in `FBGEMM` and `QNNPACK`.

Contrary to quantization in inference, when used in training, a layer's input scale, as well as BN statistics, change throughout training. The proposed implementation enables real-world training time reduction through a combination of on-the-fly scale calculation and statistics provided by the server.

Quantization in the forward pass: To optimally utilize the `int8` range and thereby preserve high accuracy, a scale $d_{\boldsymbol{x}}$ is required. The scale can be calculated using

$$d_{\boldsymbol{x}} = 2 \cdot \max\big(|\max(\boldsymbol{x})|, |\min(\boldsymbol{x})|\big)/127.0. \tag{5.5}$$

Quantized operators (e.g., `linear`, `conv2d`, or `add`) take a quantized tensor $\hat{\boldsymbol{x}}$ (and quantized weights) as input. Calculations are performed using `int8` arithmetic, but the accumulation of results is done at a higher precision (i.e., `int16/32`). Therefore, each result must be mapped back from `int16/32` to `int8` using an output scale $d_{\boldsymbol{z}}$. The output scale can be calculated similarly to eq. (5.5).

For blocks of type (a), the input scale of the input tensor $d_{\boldsymbol{x}}$ is calculated on the fly at the beginning of the first type (a) block for each minibatch. The input is then quantized and remains in its quantized representation throughout the forward propagation through type (a) blocks. For input scales $d_{\boldsymbol{x}}$, the scale calculation introduces negligible overhead,

as $\boldsymbol{x}$ is already available in its `float32` representation. Moreover, the output scales $d_{\boldsymbol{z}}$ depend on the output of an operation (e.g., `linear`, `conv2d`, `add`) and therefore cannot be calculated a priori without performing the operation in full precision, which would render quantization ineffective. Due to this PyTorch limitation, the output scale $d_{\boldsymbol{z}}$ and the BN statistics are obtained from the server and set only once per FL round.

Quantization in the backward pass: PyTorch's Autograd system does not natively support a quantized backward pass but instead expects `float32` values for each calculated gradient. To enable quantization in the backward pass, we implement a custom PyTorch *Module* for blocks of type (c), based on a custom Autograd *Function* that encapsulates all quantized operations within a single PyTorch *backward* call. Due to this limitation, quantization and dequantization are required for each quantized block in the backward pass. The scale $d_{\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}}$ of the intermediate gradients (with respect to the input) is calculated on the fly. These implementation limitations are inherently accounted for in the experimental evaluation through profiling and measurements on real hardware (i.e., addressing these limitations would further improve the efficacy of the proposed technique). Lastly, the output scales of operators in the backward pass (e.g., transposed convolution) are obtained from the server.

In total, these overheads are minor compared to the speedups gained through quantization, as large convolutional operations dominate the training time. For example, in MobileNet, the overhead from scale calculation, quantization, and dequantization of type (c) blocks is 6%, while the reduction in computation time due to quantization results in a speedup factor of 1.3.

Type (b) blocks (trained blocks) require only minor modifications to obtain the scales used by frozen blocks. To acquire the scales $d_{\boldsymbol{z}}$, PyTorch forward and backward *hooks* are used. Calculating these scales introduces negligible overhead, as all tensors in trained blocks are available in `float32`. Moreover, it is sufficient to calculate the scales in the last minibatch of a local round. After acquiring these values locally on the devices, they are uploaded to the server alongside the NN parameters and averaged together with the NN parameters.

Similarly, for the BN statistics, devices that train a specific block track the statistics μ and σ^2 during training and upload the respective estimates to the server. To perform fusion, as presented previously, devices download the statistics from the server prior to each training round and apply eq. (5.4).

Transformer-specific implementation details: For transformer NNs, encoder layers are treated as blocks. In type (a) blocks, linear, layer normalization, and ReLU operations are quantized. Due to limitations in PyTorch, the attention mechanism must remain in full precision. In type (c) blocks, linear layers and their intermediate gradient calculations are quantized.

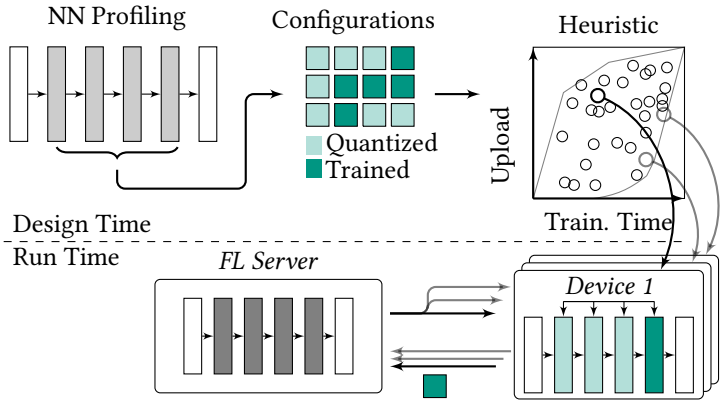


Figure 5.2: Overview of all components of CoCoFL. Profiling of configurations with respect to communication, computation, and memory is performed at design time. At runtime, a heuristic selects configurations per device and per round.

5.4 Overall CoCoFL algorithm

Partial freezing, combined with operator fusion and quantization, enables resource-specific adjustment of the communication, memory, and computational resources required in an an FL round. We present a communication- and computation-aware FL technique named CoCoFL, which allows constrained devices to select trained and frozen blocks based on their available resources. Furthermore, CoCoFL requires modifications to the server-side aggregation procedure. The runtime and design-time components of CoCoFL are depicted in fig. 5.2.

5.4.1 Heuristic Configuration Selection

A selection of trained blocks is labeled a training *configuration* $A \in \mathcal{A}$. The set of all configurations for an NN with K blocks therefore consists of $|\mathcal{A}| = 2^K$ configurations, as each block can be either trained or frozen. Furthermore, in each round, a device can select a separate configuration. This increases the total search space to $2^{K \cdot |C|}$, which is infeasible to explore in its entirety and impractical as it depends on the NN structure. Simplifying the search space by assigning a separate quality measure per configuration is also infeasible, as many devices may train different configurations. As a consequence of these observations, heuristic optimization is required. CoCoFL selects a random configuration on each device based on its available resources. Thus, the probability that all devices select the same configuration is minimal (as collective training of the same configuration has shown inferior results), while ensuring that all blocks within a device's capabilities are eventually trained. Additionally, no signaling between devices and the server, nor

Algorithm 1: CoCoFL Device

Requires: $q, t_c, m_c, \mathcal{D}_c, T, Q_c, M_c$
receive $\boldsymbol{w}^{(r)}$ from the server
$\mathcal{A}_f \leftarrow \{A \in \hat{\mathcal{A}} : t_c(A) \leq T \ \wedge \ m_c(A) \leq M_c \ \wedge \ q(A) \leq Q_c\}$ `// feasible configurations (eq. (5.6))`
$\mathcal{A}_{\max} \leftarrow \{A_j \in \mathcal{A}_f : \forall A_k \in \mathcal{A}_f : A_j \not\subset A_k\}$ `// discard non-maximal configurations (eq. (5.7)`
$A_c^{(r)} \leftarrow$ random-choice$(\mathcal{A}_{\max})$ `// random choice`
$(\boldsymbol{w}_{\text{train}}, \boldsymbol{w}_{\text{quant}}) \leftarrow \text{apply}(A_c^{(r)}, \boldsymbol{w}^{(r)})$ `// freeze/quantize blocks`
$\tilde{\boldsymbol{w}}_{\text{train}} \leftarrow$ train model $(\boldsymbol{w}_{\text{train}}, \boldsymbol{w}_{\text{quant}})$ with local data $\mathcal{D}_c$ `// local training`
send $\tilde{\boldsymbol{w}}_{\text{train}}$ to the server `// parameter update`

additional synchronization between devices, is required. A detailed ablation study of the heuristic is provided in section 5.5.

To quantify which configurations are feasible for a device to train in a given round, design-time profiling of a real implementation of each configuration is performed. Alternatively, this could also be derived using an analytical model. Profiling a single configuration takes several seconds. Therefore, profiling all 2^K configurations for a typical NN structure is infeasible. We therefore only consider configurations labeled $\hat{\mathcal{A}} \subseteq \mathcal{A}$ that train a single continuous range of blocks. This reduces the configuration space to $|\hat{\mathcal{A}}| = \frac{K(K+1)}{2}$. The runtime algorithm for devices is outlined in algorithm 1. At the beginning of a round, each participating device determines the set of feasible configurations $\mathcal{A}_f$, given by

$$\mathcal{A}_f = \{A \in \hat{\mathcal{A}} : t_c(A) \leq T \ \wedge \ m_c(A) \leq M_c \ \wedge \ q(A) \leq Q_c\}. \tag{5.6}$$

Furthermore, we discard all configurations that train a subset of blocks if there exists a configuration that includes these blocks. That is, only maximal configurations are retained, using

$$\mathcal{A}_{\max} = \{A_j \in \mathcal{A}_f : \forall A_k \in \mathcal{A}_f, A_j \not\subset A_k\}. \tag{5.7}$$

From the set of remaining configurations in $\mathcal{A}_{\max}$, a device randomly selects a configuration from $\mathcal{A}_{\max}$.

5.4.2 Aggregation of Partial Results

Each device updates only the parameters of the blocks that receive training ($\tilde{\boldsymbol{w}}_{\text{train},c}$), thereby saving communication by omitting the transfer of frozen parameters. The server (outlined in algorithm 2) weights the updates based on each device's contribution to a block

Algorithm 2: CoCoFL Server (Synchronization and Aggregation)

$\boldsymbol{w}^{(1)} \leftarrow$ random initialization
foreach round $r = 1, 2, \ldots, R$ **do**
  $C^{(r)} \leftarrow$ select devices
  broadcast $\boldsymbol{w}^{(r)}$ to selected devices $C^{(r)}$
  foreach $c \in C^{(r)}$ *in parallel* **do**
    receive $\tilde{\boldsymbol{w}}_{\text{train},c}$ from device c
  foreach *block* i **do**
    $C_i \leftarrow \{c : \tilde{\boldsymbol{w}}_{\text{train},c}$ contains block $i\}$ `// devices that trained block i`
    $\boldsymbol{w}_i^{(r+1)} \leftarrow \left(1 - \frac{|C_i|}{|C^{(r)}|}\right) \cdot \boldsymbol{w}_i^{(r)} + \frac{1}{|C^{(r)}|} \sum_{c \in C_i} \tilde{\boldsymbol{w}}_{\text{train},c}(i)$

in the round to account for partial training. For each block i in the NN, the parameters on the server are updated using

$$\boldsymbol{w}_i^{(r+1)} = \left(1 - \frac{|C_i|}{|C^{(r)}|}\right) \cdot \boldsymbol{w}_i^{(r)} + \frac{1}{|C^{(r)}|} \sum_{c \in C_i} \tilde{\boldsymbol{w}}_{\text{train},c}(i), \tag{5.8}$$

where C_i represents the set of participating devices that trained block i in the current round.

5.5 Experimental Evaluation

Partial quantization of NN models leads to hardware-specific improvements in execution time and memory consumption. Therefore, the evaluation of the proposed technique follows a hybrid approach, where on-device training loops are profiled on real hardware, and the profiling information is used to simulate large distributed systems. This hybrid approach enables the evaluation of large-scale systems with hundreds or thousands of FL devices.

5.5.1 Experimental Setup

To factor out any potential microarchitecture-dependent peculiarities with respect to quantization, the proposed technique and the baselines are evaluated on two different platforms: an x64 AMD Ryzen 7 with 64 GB RAM and a Raspberry Pi with an ARMv8 CPU (Cortex-A72 64-bit) with 8 GB RAM. For each configuration in $\hat{\mathcal{A}}$, the execution time, maximum memory usage, and upload volume are measured. The collected measurements are stored in a lookup table for the FL simulations. NN-specific details about K, $|\hat{\mathcal{A}}|$, and the hardware platform used are provided in table 5.1. For simplicity in the implementation, one *skip connection block* in ResNet, MobileNet, and DenseNet is chosen as the smallest entity

Table 5.1: Hyperparameters of FL experiments.

Setting	R	$\lvert C \rvert$	$\lvert C^{(r)} \rvert$	η	η-decay (0.1)	θ-decay	$\lvert \hat{\mathcal{A}} \rvert$	K	Platform
DenseNet	800	100	10	0.1	[750]	0.001	253	23	ARM
MobileNet	1000	100	10	0.1	[600,800]	0.01	210	21	x64
MobileNet/ GroupNorm	1000	100	10	0.1	[800]	0.001	210	21	x64
MobileNet/ XChest	1000	100	10	0.1	[600]	0.01	210	21	x64
ResNet50	800	100	10	0.1	[750]	0.01	171	19	x64
ResNet18	600	3500	35	0.1	[400]	0.01	55	11	ARM
Transformer	1000	100	10	0.1	[800]	–	35	8	x64

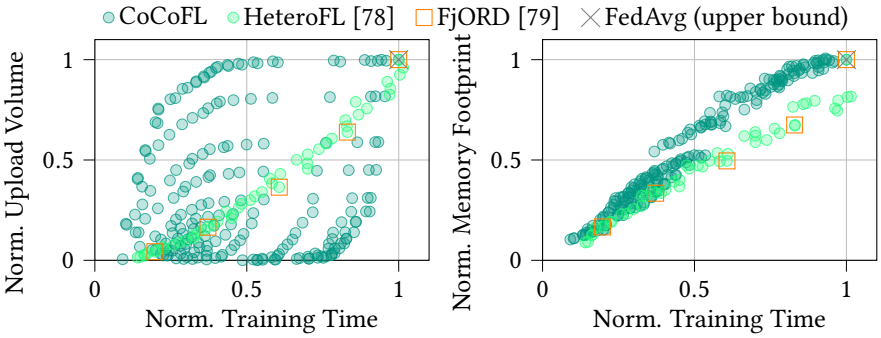


Figure 5.3: Exemplary profiling of MobileNet with respect to communication (upload volume), computation (training time), and memory, normalized to end-to-end training. Each marker in the plot represents a training configuration $A \in \hat{\mathcal{A}}$.

that is either trained or frozen. Profiling results—specifically, training time over upload volume and training time over memory footprint—of MobileNet on x64 are visualized in fig. 5.3. All quantities are normalized to 1, where 1 represents end-to-end training of the NN.

Profiling setup details: On average, profiling takes 17 min for MobileNet on x64 and 1 h on ARM. The peak memory reached during training of a specific NN architecture can be measured using operating system tools. To obtain a representative measurement of peak memory usage, the following training-related procedures are executed:

1. The NN model parameters are loaded from disk into memory.
2. A training batch (i.e., data with a batch size of 32) is loaded from disk into memory.
3. The optimizer is initialized.

4. Training (i.e., forward pass, backward pass, and parameter update) is performed for 16 minibatches.

The maximum memory consumed during this minimal training procedure is measured using the Linux syscall `getrusage()` with the flag `RUSAGE_SELF`. This call returns the maximum amount of memory consumed by the respective process in the form of a struct containing a variable called `ru_maxrss`. For all measurements, the overhead caused by the required Python packages (e.g., PyTorch, NumPy) is measured before the training procedure and subtracted from the final value.

The same training procedure is used for measuring the training time. The time measurements include the NN's forward pass execution, setting gradients to zero, backpropagation, and optimizer steps. The upload volume can be directly acquired by calculating the size of the NN's `state_dict`, filtering for parameters that required gradient calculations during training. Using the same setup, training with subsets of filters, as used in HeteroFL and FjORD, is also evaluated.

Profiling of baselines: Similar to the profiling of $\hat{\mathcal{A}}$, the baselines HeteroFL and FjORD are also profiled. In both cases, a configuration refers to a specific ratio of filters that are retained while others are dropped. As a result, the resource footprints of CoCoFL and HeteroFL/FjORD differ. The profiling results show that the combination of freezing, fusion, and quantization enables training configurations that reduce execution time by up to 90% and memory footprint by up to 98% relative to end-to-end training of the NN. Furthermore, the results indicate that CoCoFL's configurations provide a larger, independent space of configurations, where communication and memory are more decoupled. This results in greater flexibility when selecting a suitable configuration. Consequently, available resources can be utilized more efficiently. In contrast, HeteroFL/FjORD configurations exhibit tightly coupled communication and memory/computation reduction, meaning one cannot be adjusted independently of the other.

5.5.2 FL Setup and Hyperparameters

Using the gathered profiling measurements, CoCoFL and the baselines are evaluated with data from the CIFAR10/100, FEMNIST, CINIC10, XChest, and Shakespeare datasets. The respective training set is split and distributed to C as described in chapter 4. NN models ResNet, DenseNet, MobileNet, and transformers are used for evaluation. In each round, a subset $C^{(r)}$ is randomly selected for participation in round r. Additionally, before training, devices are randomly grouped into three equally sized sets, $\tilde{C}_1$, $\tilde{C}_2$, and $\tilde{C}_3$, referred to as the sets of *strong*, *medium*, and *weak* devices. The set of strong devices is capable of training the NN end-to-end and, therefore, has no limiting computation, memory, or communication constraints. The FL round time T is set to the time required for a strong device to complete a local training round. Furthermore, a set of medium devices is introduced, each having 2/3 of the computational and memory resources of a strong

device (i.e., relative to training the NN end-to-end). To match the round time T, medium devices must select configurations that reduce the required computations to 2/3 of those of strong devices. Lastly, the set of weak devices has 1/3 of the computational and memory capabilities of strong devices. The communication budget for medium and weak devices is modeled randomly over rounds to simulate an environment with varying channel quality. Specifically, the communication budget of a device in $\tilde{C}_2$ or $\tilde{C}_3$ in round r is sampled using a uniform distribution, such that

$$Q_i^{(r)} \sim \mathcal{U}\left(\frac{Q_{\text{strong}}}{2}, Q_{\text{strong}}\right) \quad \forall i \in \tilde{C}_2 \cup \tilde{C}_3. \tag{5.9}$$

For all federated training, the SGD optimizer with an initial learning rate of $\eta = 0.1$ is used. To ensure a fair comparison with the baselines, momentum is disabled for SGD, as it is incompatible with FjORD. The remaining hyperparameters are provided in table 5.1. For each experiment, the average accuracy and the standard deviation of three independent seeds are reported after R rounds of training. Several data split scenarios are studied: an iid case, where data is randomly distributed to devices, and the quantity of samples per class is equal across all devices. Furthermore, we examine a non-iid case, where the quantity of samples per class varies randomly among devices. Lastly, we study a rc-non-iid case, as described in section 2.2, where the quantity of samples per class correlates with the device group, i.e., some class information is available only on weak or medium devices.

5.5.3 Experimental Results

Vision models: When data is iid, CoCoFL performs close to the full-resource upper bound FedAvg, improving the final accuracy over the baselines by 5.5 p.p. for DenseNet (CIFAR10) and by 3.7 p.p. and 6.6 p.p. for MobileNet (CIFAR10) and ResNet50 (CIFAR100), respectively. Similar trends are observed for CINIC10, clearly indicating the effective resource utilization of CoCoFL compared to the baselines. An outlier to this trend is the FEMNIST dataset, where FedAvg with device dropping achieves the highest accuracy, despite the fact that 2/3 of the devices are dropped. This can be attributed to the high number of redundant samples available on the devices (10 K per class, compared to 5 K and 500 in the cases of CIFAR10 and CIFAR100, respectively). However, it can be observed that when the sample count is very limited, as is the case with XChest (only 12 K samples in total), the advantage of CoCoFL over the baselines increases.

The general necessity of including constrained devices in FL training becomes even more evident with non-iid data. As shown in table 5.2, using a Dirichlet distribution with $\alpha = 0.1$ results in a larger gap between the upper bound and the naive baseline, demonstrating the importance of constrained devices. CoCoFL enables constrained devices to make a more valuable contribution to the global model, achieving up to 20 p.p. higher accuracy compared to the state of the art. In the case of DenseNet and MobileNet, FjORD and

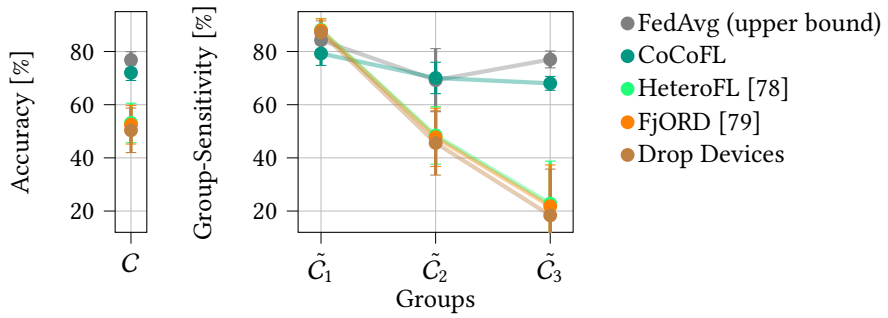


Figure 5.4: Accuracy and group-sensitivity of MobileNet on CIFAR10 with rc-non-iid data. The results show that CoCoFL enables fairness for constrained devices ($c \in \tilde{C}_2$ and $c \in \tilde{C}_3$), achieving performance close to the upper bound. The baselines HeteroFL and FjORD perform similarly to device dropping.

HeteroFL exhibit inferior performance, even when compared to the naive baseline. Similar conclusions can be drawn from experiments with ResNet18/50, although FjORD and HeteroFL perform slightly better compared to device dropping.

Fairness in rc-non-iid: To quantify the contribution of devices from the medium ($\tilde{C}_2$) and weak ($\tilde{C}_3$) groups, group-sensitivity is calculated based on eq. (2.20). The results for MobileNet (rc-non-iid) are visualized in fig. 5.4, where CoCoFL achieves group-sensitivities of 79%, 70%, and 68% for devices in $\tilde{C}_1$, $\tilde{C}_2$, and $\tilde{C}_3$, respectively. These results are close to the group-sensitivity parity provided by the upper bound FedAvg. In contrast, the baselines HeteroFL and FjORD reach 88%, 48%, and 23% for $\tilde{C}_1$, $\tilde{C}_2$, and $\tilde{C}_3$, highlighting that strong devices that train the NN end-to-end dominate the global model, while medium and weak devices struggle to embed their class knowledge into the global model. Both baselines perform similarly to dropping medium and weak devices completely, failing to ensure fairness among the groups.

Other normalization techniques: To evaluate the impact of other common normalization techniques, such as group normalization [117], we study a modified version of MobileNet where batch normalization is replaced with group normalization. For CIFAR10, it can be observed that, with the same hyperparameters, the accuracy of all algorithms is lower; however, the general trends remain similar to MobileNet with BN. In rc-non-iid scenarios, CoCoFL achieves group-sensitivities of 63.7% ± 3.5%, 60.7% ± 5.6%, and 58.3% ± 9.8% for $\tilde{C}_1$, $\tilde{C}_2$, and $\tilde{C}_3$, respectively. FjORD and HeteroFL achieve 80.7% ± 7.2%, 42.8% ± 8.5%, and 18.3% ± 14.4%, and 81.3% ± 7.1%, 42.8% ± 9.0%, and 19.0% ± 13.5%, respectively. Overall, the results indicate that CoCoFL improves both accuracy and fairness compared to the baselines, regardless of the normalization technique used.

Table 5.2: Final accuracy for CoCoFL and baselines. Accuracy in % for DenseNet, MobileNet, ResNet18, ResNet50, and transformer. For results of XChest with MobileNet, the F1 score is given, as the dataset is unbalanced. For each experiment, the average and standard deviation of three independent seeds is given. For transformer (TF) with Shakespeare, data is distributed according to the Leaf [108] benchmark.

NN	DenseNet			MobileNet					ResNet18	
Setting	CIFAR10			CIFAR10		CIFAR10 (w. GroupNorm)			FEMNIST	
Dirichlet α	– (iid)	n.-iid@0.1	rc@0.1	– (iid)	rc@0.1	- (iid)	n.-iid@0.1	rc@0.1	– (iid)	rc@0.1
FedAvg (upper b.)	84.3±0.1	75.6±1.5	74.9±3.3	84.9±0.2	77.4±2.7	76.1±0.1	69.4±0.5	72.9±1.4	86.2±0.1	82.9±1.2
CoCoFL	**82.0±0.2**	**71.9±1.8**	**68.8±4.6**	**83.2±0.3**	**72.4±2.9**	**71.3±0.1**	**61.2±1.4**	**63.6±3.5**	85.0±0.1	**81.5±0.6**
FjORD [79]	73.7±0.1	60.4±2.1	48.8±6.8	79.1±0.3	51.9±7.3	64.4±0.6	42.9±1.7	47.0±5.0	85.5±0.0	69.3±8.3
HeteroFL [78]	76.4±0.3	64.0±2.4	51.2±7.4	79.5±0.2	53.0±7.6	64.8±0.1	55.2±0.3	47.5±5.0	85.9±0.0	70.9±5.8
Drop Devices	76.5±0.1	60.4±4.2	50.9±7.5	78.1±0.4	49.9±8.7	56.2±0.8	52.8±1.1	45.9±4.7	**86.1±0.1**	64.9±7.8

NN	ResNet50		DenseNet			MobileNet(large)			TF	TF-S2S	
Setting	CIFAR100		CINIC10			XChest			IMDB	Shakespeare	
Dirichlet α	– (iid)	rc@0.1	– (iid)	n.-iid@0.1	rc@0.1	– (iid)	n.-iid@0.5	rc@0.5	– (iid)	– (iid)	–rc (Leaf)
FedAvg (upper b.)	57.0±0.3	53.0±0.6	77.2±0.1	53.9±2.3	65.1±1.1	94.1±0.3	85.9±1.8	93.2±0.2	82.6±0.4	49.1±0.1	49.4±0.1
CoCoFL	**52.5±0.2**	**41.8±2.5**	**73.6±0.1**	**53.5±4.3**	**52.4±7.3**	**91.3±0.3**	**73.0±6.4**	**87.3±3.8**	**82.5±0.5**	**49.3±0.3**	**49.1±0.3**
FjORD [79]	43.6±0.8	29.6±4.3	65.1±0.7	49.2±2.2	41.1±6.9	66.3±0.9	52.7±3.9	62.4±0.8	78.5±0.7[1]	42.9±0.5	43.0±0.3
HeteroFL [78]	45.9±0.7	31.0±2.3	69.4±0.2	50.4±2.4	43.4±7.2	69.4±1.0	65.0±0.9	65.4±1.6	79.2±0.3[1]	44.1±0.2	44.1±0.2
Drop Devices	35.2±0.2	23.7±0.4	67.7±0.4	48.3±2.5	42.4±7.2	68.2±1.0	67.0±0.6	66.8±1.4	78.5±0.6	40.5±0.3	40.3±0.1

[1]No configuration for weak devices available, therefore weak devices are dropped.

NLP models: To demonstrate the applicability of CoCoFL to NLP problems, the freezing and quantization scheme is adapted for transformers. Two tasks are studied: text classification with IMDB and next-character prediction with the Shakespeare dataset. The studied transformer consists of 6 encoder layers with an embedding size of 128, a hidden size of 128, 2 attention heads, and a single linear output layer. For IMDB, the sequence length is set to 512, while for Shakespeare, it is set to 80. For IMDB, a SentencePiece tokenizer with a vocabulary size of 16 K is used, and the final layer maps to a binary output for classifying a review as positive or negative. For Shakespeare, tokenization is performed at the character level. Similar to convolutional NNs, HeteroFL and FjORD scale down the feature embedding and hidden dimensions. To adjust resource ratios, the values [100%, 81.25%, 62.5%, 40%, 12.5%] are used. However, for IMDB, a large memory overhead remains, preventing weak devices from having suitable configurations to participate in training, leading to their exclusion. For the Shakespeare rc-non-iid experiment, data is distributed as proposed in the Leaf benchmark [108], where different Shakespeare plays (a total of 25) are assigned to different device groups. The results in table 5.2 confirm that CoCoFL is also effective for NLP tasks with transformers, significantly outperforming the studied baselines.

5.5.4 Ablation Study

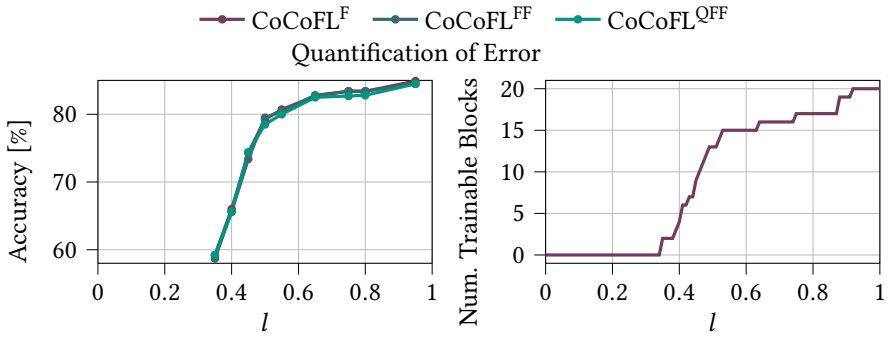
Several ablation studies are conducted to highlight the impact of individual components of CoCoFL.

Quantification of the error: To quantify the gains and the quantization error introduced through operator fusion of frozen blocks in CoCoFL, the MobileNet/CIFAR100 iid experiment is modified such that, instead of three, two groups are used. The first group consists of 10% strong devices, i.e., devices capable of training the NN end-to-end. The remaining 90% of the devices belong to the second group, labeled *limited* devices. Each of the limited devices has an adjustable relative computation and memory limit parameterized by l, such that

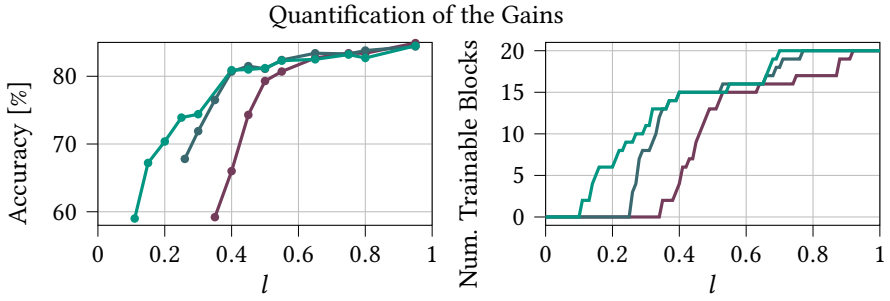
$$t_{\text{limited}}(A) = \frac{1}{l} \cdot t_{\text{strong}}(A) \qquad M_{\text{limited}} = l \cdot M_{\text{strong}}. \tag{5.10}$$

No communication constraint is applied, so $Q_{\text{limited}} = Q_{\text{strong}}$. As a result, a limited device, depending on l, must select a configuration A that satisfies both the memory and computation constraints. Several experiments are conducted where l is varied within $l \in (0, 1]$. All remaining hyperparameters are identical to those in the mainline experiments (table 5.1). To better highlight the individual contributions of the various components of CoCoFL, three variants of CoCoFL are introduced. For each variant, the respective memory footprint and computational complexity are profiled to assess the configurations.

- CoCoFL$^{\text{F}}$is a variant where only freezing is applied, without quantization or operator fusion. The set of feasible configurations is labeled $\hat{\mathcal{A}}^F$.



(a) The left plot shows the final accuracy without utilizing the gains of quantization while still using quantized arithmetic. The right plot shows the cumulative number of trainable blocks, which is the same for all three variants since gains are ignored. The error analysis indicates that the error introduced through quantization and fusion is negligible.



(b) The left plot shows the final accuracy utilizing the gains. The right plot shows the cumulative number of trainable blocks, utilizing the gains of quantization and fusion. When gains are utilized, limited devices can train more blocks and achieve higher accuracies, significantly outweighing the error.

Figure 5.5: Ablation study of the error and gains introduced by quantization and fusion in CoCoFL.

- CoCoFL$^{\text{FF}}$is a variant where freezing and operator fusion are applied, but without quantization. Configurations are labeled $\hat{\mathcal{A}}^{FF}$.
- CoCoFL$^{\text{QFF}}$is the mainline variant. The configurations are equivalent to $\hat{\mathcal{A}}$.

To quantify the error introduced through fusion, estimation of statistics, and quantization noise, all three variants CoCoFL$^{\text{F}}$, CoCoFL$^{\text{FF}}$, and CoCoFL$^{\text{QFF}}$are evaluated for different values of l. Each variant uses the configurations in $\hat{\mathcal{A}}^F$ for the limited devices, ignoring all gains from quantization and fusion while incorporating the introduced error. The right part of fig. 5.5a displays the cumulative number of trainable blocks for a given l in $\hat{\mathcal{A}}^F$. Since all three variants use $\hat{\mathcal{A}}^F$, only a single line is shown in the plot. To participate in training, limited devices require a minimum $l = 0.35$, which allows them to train a single block. The accuracy over l for the three variants is visualized on the left side of fig. 5.5a. It can be seen that, overall, the error is mostly below 1 p.p., with a maximum of 2.3 p.p. for CoCoFL$^{\text{QFF}}$and 1.7 p.p. for CoCoFL$^{\text{FF}}$.

Quantification of the gains: To quantify the gains, each introduced variant uses its own profiling results, such that CoCoFL$^{\text{FF}}$and CoCoFL$^{\text{F}}$use $\hat{\mathcal{A}}^{FF}$ and $\hat{\mathcal{A}}^{F}$, respectively. CoCoFL$^{\text{QFF}}$enables a maximum reduction of 75% in computation time and 60% in memory compared to CoCoFL$^{\text{F}}$(45% and 28% compared to CoCoFL$^{\text{FF}}$). Consequently, at a constraint parameterized by l, CoCoFL$^{\text{QFF}}$and CoCoFL$^{\text{FF}}$can train more configurations, and more importantly, configurations with more blocks. This results in higher accuracy in FL, as shown in the left of fig. 5.5b, where with the same $l = 0.25$, CoCoFL$^{\text{QFF}}$achieves an increase of 6.5 p.p. in final accuracy over CoCoFL$^{\text{FF}}$, while CoCoFL$^{\text{F}}$ has no configuration that satisfies the constraints of limited devices. At a parameterized constraint with $l = 0.4$, CoCoFL$^{\text{QFF}}$achieves a final accuracy increase of 14.3 p.p. over CoCoFL$^{\text{F}}$. The results suggest that the more blocks limited devices can train, the higher the final accuracy. This is visualized on the right of fig. 5.5b, where the total number of trainable blocks is cumulatively plotted over l. In the case of l approaching 1, the advantage of quantization and fusion vanishes and can even result in small accuracy losses due to the introduced error.

Configuration selection: Lastly, to verify the robustness of the proposed selection scheme for feasible configurations, specifically the per-device, per-round random selection of a configuration from $\mathcal{A}_{\text{max}}$, several experiments are conducted. For these experiments, the same setup as in the error and gains ablation study is used, i.e., a share of 10% strong devices and 90% limited devices. The constraints are parameterized using $l \in [0.2, 0.3, 0.4, 0.5]$ to verify that the proposed heuristic remains robust across a wide range of possible constraints. Firstly, the configuration reduction mechanism is studied in more detail, specifically, we compare:

- **max**: Keeping only maximum configurations $\mathcal{A}_{\text{max}} \subseteq \mathcal{A}_f$, i.e., configurations that are not a subset of any other feasible configuration. This setting is equivalent to the mainline CoCoFL approach.
- **all**: Keeping all feasible configurations, i.e., $\mathcal{A}_f \subseteq \hat{\mathcal{A}}$.

Secondly, orthogonal to the configuration reduction, the configuration selection in CoCoFL is compared against:

- **max**: Using the configuration that trains the maximum number of blocks within a device's capabilities. The combination of this selection mechanism with both reduction mechanisms (i.e., **max+max** and **max+all**) results in the same configurations being selected, so it is only evaluated once.
- **min**: Training the configuration with the minimum number of blocks.
- **round-robin**: Switching between feasible configurations in a round-robin manner (all limited devices pick the same configuration in a round).
- **random**: Randomly switching between configurations, as used in mainline CoCoFL.

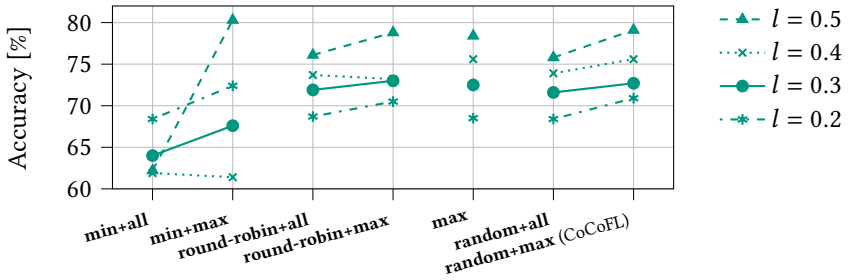


Figure 5.6: Ablation study of heuristic configuration selection in CoCoFL and its impact on final accuracy for a range of $l \in [0.2, 0.3, 0.4, 0.5]$.

The final accuracy is presented in fig. 5.6. It can be observed that, in almost all cases, using only maximal configurations (i.e., configurations that are not a subset of another feasible configuration) increases the final accuracy, independent of the configuration selection strategy. Most importantly, we observe that **random+max** (as employed in CoCoFL), independently of l, is among the highest-performing selection strategies. Furthermore, it can be observed that training the same blocks on all devices does not improve upon random selection, as **round-robin+max** performs worse than **random+max**. Depending on l, **max** and **min+max** can outperform **random+max**; however, not consistently across different values of l.

5.6 Summary

This chapter proposed CoCoFL, an FL technique that allows participating devices to adjust to individual communication, computation, and memory constraints by allowing constrained devices to freeze, fuse, and quantize specific layers. The evaluation shows that CoCoFL better incorporates knowledge from constrained devices into the server model, thus preserving fairness among participants. The ablation study shows that the gains from quantization largely outweigh the introduced quantization noise. Furthermore, the ablation study shows that a random selection of maximal configurations enables the highest accuracy without requiring additional signaling and synchronization overhead.

Compared to the state of the art, which utilizes subsets of the NN structure, in CoCoFL, layer gradients are calculated based on the full NN structure. While quantized layers are processed on CPUs, potential computation gains could be even higher by incorporating integer inference accelerators that are already embedded in smartphone SoCs. Lastly, CoCoFL remains compatible with techniques such as client selection, communication compression, and differential privacy techniques.

6 Memory-Constrained FL

6.1 Scope and Contribution

While in chapter 5, strict per-round communication, computation, and memory constraints are enforced, this chapter relaxes the assumption that communication and computation limits are enforced on a round level; rather, both should be efficiently used throughout the training. However, peak memory requirements for training remain a main obstacle, especially in FL. In real-world deployments, when memory capacity is not large enough to keep weights, activations, and optimizer states in memory, devices are simply dropped from the training [40].

In chapter 5, prior art assumes that there must be a share of devices capable of training the NN end-to-end; otherwise, a share of parameters never receives any training. This chapter loosens this assumption and aims to train an NN with devices in FL that are all incapable of training the NN end-to-end due to limited memory.

6.1.1 Motivational Example

Several techniques are proposed to solve this task given these assumptions, mainly by building a lower-complexity submodel on the devices and embedding it into a large server NN. Submodels are usually constructed by scaling the width of the NN; for example, in the case of convolutional NNs, reducing the number of filters per layer. In particular, Caldas et al. propose FD [77], which randomly selects, per round and per device, a subset of the large NN server model for on-device training. Alam et al. propose FedRolex [80], where the subset selection for all devices resembles a sliding window approach in which, each round, the filter indices are shifted by one. Both techniques compared to other subset techniques [79, 78] allow a set of constrained devices to train a larger server NN without any device being capable of training it fully end-to-end. The goal is thereby to make use of the larger capacity the server NN provides.

We find, however, that this is not the case; rather, applying [77, 80] deteriorates the accuracy when compared to a small model that can be fully trained end-to-end. To find experimental evidence, we compare both techniques in an FL scenario with CIFAR10, FEMNIST, and

This chapter is mainly based on [2].

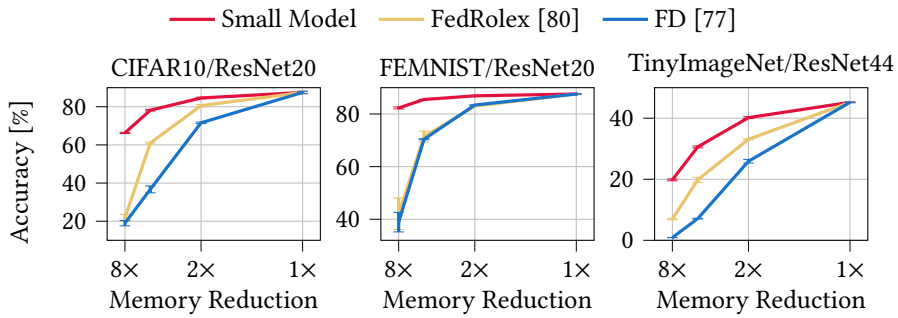


Figure 6.1: Final accuracy over memory reduction of FedRolex and FD compared to a small model baseline using different NN topologies and datasets. In the case of 1×, both techniques are equivalent to FedAvg with end-to-end training (upper bound).

TinyImageNet. The device NNs are scaled down to enable memory reductions of 2×, 4×, and 8×. Similarly, we construct a small model baseline by scaling the NN but using the same smaller-scale NN for aggregation on the server, i.e., performing baseline FedAvg with a smaller-scale model. In fig. 6.1, it can be seen that, in general, with the presented combination of datasets and NN topologies, reducing the capacity, i.e., scaling down the NN, decreases the reachable accuracy. For example, for CIFAR10, the accuracy of a full-scale NN at 87.5% drops to 66.3% when a downscaled NN is used to satisfy an 8× memory constraint. Consequently, the trained task benefits from more NN capacity, especially with TinyImageNet, which is a more complex dataset compared to the others. Further, it can be seen that, especially for 4× and 8× enforced memory reduction, the accuracy of FD and FedRolex significantly drops compared to a small model. For example, for CIFAR10, FedRolex accuracy decreases to 21.7% for 8×.

A potential reason for the observed large drop in accuracy of FD and FedRolex is that a large portion of filters is dropped every round from the local training, extremely limiting the co-adaptation of these parameters. The gradients for a currently trained subset on a device are calculated without consideration of the already trained parameters that reside on the server. This causes the server NN to contain redundant features that degrade the overall accuracy, as FD shares methodological similarity with regular dropout [118]. Regular dropout is commonly used as a regularization technique that limits the NN's dependence on specific neurons by randomly dropping neurons during training, therefore purposefully limiting the co-adaptation between parameters. While a limited amount of dropout of parameters can help with regularization and prevent overfitting, FD and FedRolex adopt the mechanism but exercise dropout to its extreme, dropping large portions of the parameters such that the co-adaptation between parameters is insufficient to achieve high accuracy. As mentioned before, in FD, the gradients for the subset of parameters on a device are calculated without consideration of the error of the remaining parameters that reside on the server. Given the random nature of the subset selection, as subsets are selected randomly and change per device and per round, the chance that a specific subset

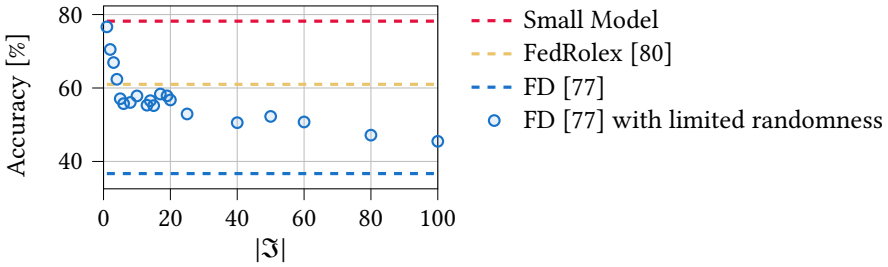


Figure 6.2: Final accuracy of FD with limited randomness on CIFAR10 (4× memory reduction) and ResNet20 over the cardinality of $\mathfrak{I}$.

is trained over a sizeable portion of the data (such that its parameters can co-adapt) is relatively low.

To further study the effects of co-adaptation on the reachable accuracy and the differences between FD and FedRolex, the following experiment is conducted using CIFAR10 and a 4× memory reduction scheme:

- To control the randomness, i.e., the likelihood that parameters are trained in conjunction with each other, FD is modified such that all devices train the identical subset indices in a round, i.e., $\mathcal{I}^{(r,c)} = \mathcal{I}^{(r)}$ (layer index k is omitted for simplicity).
- The randomness is further controlled by sampling specific subsets prior to the training, which are then chosen randomly each round. Therefore, $\mathcal{I}^{(r)}$ is not constructed randomly each round but is picked from a fixed set $\mathfrak{I} = \{\mathcal{I}_1, \dots, \mathcal{I}_{|\mathfrak{I}|}\}$. Further, $\mathcal{I}^{(r)}$ is uniformly sampled each round from $\mathfrak{I}$, i.e., $\mathcal{I}^{(r)} \sim \mathcal{U}(\mathfrak{I})$. Thereby, the cardinality of $\mathfrak{I}$ impacts the likelihood of two parameters being trained together in a round. The aggregation of subsets at the server is kept unchanged.

To highlight the dependence of the accuracy on the co-adaptation of parameters, the cardinality of $\mathfrak{I}$ is varied, such that $|\mathfrak{I}| \in [1, 100]$. Thereby, the probability that a specific subset is selected is $p = \frac{1}{|\mathfrak{I}|}$. The final accuracy after 2500 rounds of training is visualized in fig. 6.2.

Several effects can be observed:

1. In the case where $|\mathfrak{I}| = 1$, FD achieves the highest accuracy and resembles a small model, even though some filters remain in their initialized state on the server and never receive training.
2. The final accuracy of the modified FD algorithm drops proportionally to p with increased $|\mathfrak{I}|$. As more randomness is introduced, co-adaptation is reduced, and accuracy decreases. Since FD randomly picks subsets per round per device, its co-adaptation—and hence its accuracy—is even lower than training with 100 distinct subsets.

3. FedRolex, specifically its sliding window approach, is a special case of FD with limited randomness. Its accuracy is equivalent to $|\mathfrak{I}| = 5$ in this experiment.

6.1.2 Contribution

The experiments shown above suggest that applying FD and FedRolex is rather harmful and that they cannot improve the accuracy beyond an end-to-end trained small model. Thus, they cannot make use of the increased capacity of a larger server model. Based on these observations, this chapter proposes a novel technique that enables successive freezing and training of parameters of the FL models on devices. This technique, therefore, reduces the training resource requirements while allowing greater co-adaptation between parameters. In doing so, the technique can utilize the larger capacity NN at the server, increasing the accuracy beyond an end-to-end trained small model baseline. Instead of switching between random subsets of the server NN on a round-by-round basis, the same set of parameters is trained for several rounds, and successively, the subset size is increased by adding untrained parameters. To comply with the same memory constraints as in [80, 77], the early layers of the NN are trained at full width, while a scaled-down NN head is used.

Then, throughout the FL rounds, the early layers are frozen, and the NN's head is expanded. By freezing the early layers, no activations need to be kept in memory; hence, the memory footprint is reduced. However, unlike standard FL and FedRolex, the error of the frozen layers is incorporated into the gradient calculation throughout the training, allowing for co-adaptation between the parameters. This scheme is applied successively throughout the training until all parameters of the large-capacity server NN are trained. In summary, this chapter introduces the following novel contributions:

- We empirically show that employing current state-of-the-art techniques, FD and FedRolex, can actually hurt performance in memory-constrained systems.
- We propose a novel training scheme labeled *Successive Layer Training (SLT)*, which addresses the shortcomings of state-of-the-art methods by successively adding more parameters to the training, successively freezing early layers, and reusing a scaled-down head throughout the training.
- The evaluation of common NN topologies, such as ResNet and DenseNet, shows that SLT achieves significantly higher accuracies in iid and non-iid CIFAR, FEMNIST, TinyImageNet, and ImageNet training compared to state-of-the-art methods and end-to-end training of a small model. Further, SLT provides faster convergence, reducing communication overhead required to reach a certain accuracy level by over 10× compared with FedRolex and FD. The same observation is made for computational effort (i.e., FLOPs).
- Lastly, SLT is capable of incorporating devices with heterogeneous memory constraints, outperforming FD and FedRolex, as well as HeteroFL [78] and FjORD [79].

6.2 Problem Statement

This work, similarly to the previous chapter, targets a synchronous cross-device FL setting, where $c \in C$ devices act as FL participants, and a single server is responsible for aggregation. Further, there exists a specific NN topology F that is intended to be trained by the devices in a distributed manner over R rounds. The objective is to maximize the global model accuracy. In any round $r \leq R$, it is assumed that a fixed number of devices $|C^{(r)}|$ participate. Devices are assumed to have limited memory M_c, specifically in such a way that no device can train the NN end-to-end. Communication and computation on devices are considered weak constraints, i.e., there is no fixed per-round constraint; however, these resources should be used efficiently.

6.3 Methodology

The following section describes the proposed technique called *SLT* for convolutional NNs. The given NN architecture F can be described as the consecutive execution of K operations or layers, where each layer is defined as f_k, with $k \in [1, \ldots, K]$. We label a convolution followed by batch normalization and an activation function as a layer. Further, in line with [77, 80], a subset $\overline{\boldsymbol{w}_k}$ of the layer parameters (on the server) $\boldsymbol{w}_k$, scaled down using s, is described as

$$\overline{\boldsymbol{w}_k} = \boldsymbol{w}_k^{s,s} \qquad \overline{\boldsymbol{w}_k} \in \mathbb{R}^{\lfloor sP_k \rfloor \times \lfloor sV_k \rfloor} \qquad \boldsymbol{w}_k \in \mathbb{R}^{P_k \times V_k}, \tag{6.1}$$

where P_k denotes the layer's input dimension, V_k denotes the output dimension of the fully-sized parameters, and $s \in (0, 1]$ represents a scaling factor. The same concept applies to four-dimensional parameters, such as conv2d kernels, where the kernel dimensions remain unscaled. Thus, for brevity, these indices are omitted. To obey memory constraints M_c on the participating devices, the NN layers are split into three consecutive parts. The first part of the NN contains layers that have already received training and remain frozen on the devices. The second part of the NN contains layers that are being fully trained on the devices, while the last part represents the NN's *head*. To train the remaining layers within the memory budget M_c, the head's parameters are scaled down as described in eq. (6.1). Throughout the FL training, the *training configuration* is successively switched, i.e., the part of the NN that is being fully trained ($s = 1$) moves from the first layer to the last layer. Thereby, the remaining untrained parameters from the down-scaled head are successively added. In addition to switching training configurations, more early layers are successively frozen, starting with the first layer to comply with the memory budget. The parts of the NN that are frozen, trained, and the down-scaled head are labeled F_F, F_T, and F_H, respectively, such that the full model can be described as $F = F_F \circ F_T \circ F_H$. The switching from one configuration to the next is visualized in fig. 6.3.

- The first part, F_F, labels the portion of the NN that is frozen, where the server parameters $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{K_F}$ are not updated by the devices. Freezing these layers removes the need to store activations for backpropagation, thus lowering the memory overhead during training. The frozen part of the NN, where layers $1, \ldots, K_F$ remain frozen, is defined as
$$F_F = \bigcirc_{k \in \{k:0<k\leq K_F\}} f_k. \tag{6.2}$$

- The second part, F_T, labels the portion of the NN that is fully trained by the devices. The parameters $\boldsymbol{w}_{K_F+1}, \ldots, \boldsymbol{w}_{K_T}$ are updated during training. This part of the NN is defined as
$$F_T = \bigcirc_{k \in \{k:K_F<k\leq K_T\}} f_k. \tag{6.3}$$

- The last part, F_H, describes the NN's head, which is scaled down using s, such that
$$F_H = \bigcirc_{k \in \{k:K_T<k\leq K\}} f_k, \tag{6.4}$$
where the down-scaled layers $K_T + 1, \ldots, K$ receive training. Layer $K_T + 1$, i.e., the first layer of F_H, scales down the parameters to the width of the head such that $\overline{\boldsymbol{w}}_{K_T+1} = \boldsymbol{w}^{1,s}_{K_T+1}$, where $\boldsymbol{w}^{1,s}_{K_T+1} \in \mathbb{R}^{P_{K_T+1} \times \lfloor sV_{K_T+1} \rfloor}$. Consecutive layers are scaled down using s, such that $\overline{\boldsymbol{w}}_{K_T+j} = \boldsymbol{w}^{s,s}_{K_T+j} \quad \forall j \in [2, \ldots, K - K_T]$.

We define A as a training configuration that complies with the given memory constraint, such that $A = \{K_F, K_T, s\}$ fully describes the mapping of layers f_k, where $k \in [1, \ldots, K]$, into F_F, F_T, and F_H, along with the head's scaling factor s. Each A has a respective memory footprint that can be assessed using the function $m : \mathcal{A} \mapsto \mathbb{R}$, such that $M = m(A)$, where $\mathcal{A}$ is the set of all possible configurations. The mapping m, i.e., the peak memory footprint of a configuration A, can either be measured empirically or determined analytically by calculating the size of the weights and the activations that must be kept in memory.

6.3.1 Configuration Selection

For each selected configuration A, we aim to fully utilize the available memory. We define n as a *configuration step*, where $n \in [1, \ldots, N]$, with a total of N steps. At each step, parameters are added to the training (i.e., filling up the remaining parameters of a head's layer). Steps are distributed over the total training rounds R (refer to fig. 6.4). K_T is set to $K_F + 1$, such that at every configuration step, exactly one layer is fully trained (filled up). The process starts with $K_F = K_T = 0$ (consequently, $F = F_H$) to pre-train the head for a certain number of rounds. After pre-training, K_F is incremented one by one, and $K_T = K_F + 1$ is applied (in the first configuration, no layers are frozen, thus $F = F_T \circ F_H$). The transition to the consecutive configuration is achieved by increasing K_F by one. Hence,

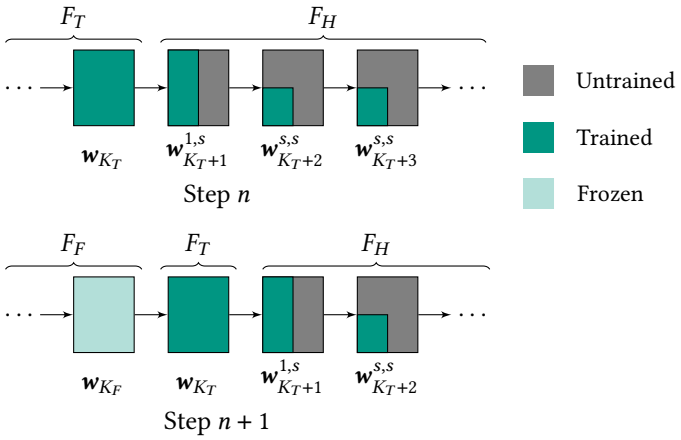


Figure 6.3: Visualization of SLT. With every step n, K_F and K_T are incremented by 1. $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{K_F}$ denote the parameters that remain frozen during training, while $\boldsymbol{w}_{K_T}$ denotes the parameters of layer K_T that are fully trained. $\boldsymbol{w}_{K_T+1}, \ldots, \boldsymbol{w}^{s,s}_K$ denote the parameters of the scaled-down head using the scaling factor s.

for the training configuration at step n, we have $K_F = n - 1$ and $K_T = n$. The scaling factor of the head, s_n, at step n is determined using

$$\max\ s_n \text{ s.t. } m(A_n) \leq M\ \wedge\ s_n \leq s_j\ \forall j \in [n+1, \ldots, N]. \tag{6.5}$$

Equation (6.5) ensures that each configuration complies with the memory limit M. Secondly, it enforces that s_n can only increase with n, ensuring that parameters are only added to the head throughout the training and never (not even temporarily, as in prior work) removed. A justification for maximizing s instead of F_T is presented in the form of an ablation study in section 6.4. The configuration selection can be performed independently of the data in an offline manner. Lastly, the final step N is defined as the step where all remaining parameters (if permitted by the memory constraint) can be added to the training, resulting in $s = 1$. If this step is reached, A_N is used for training in the remaining FL rounds. The training process is visualized in fig. 6.4, while the full algorithm is outlined in algorithm 3.

6.3.2 Mapping from Steps N to Rounds R

All required steps N to eventually train all parameters in the server NN are distributed over the total number of FL rounds R. Specifically, the rounds are allocated based on the share of parameters added to the training within a configuration A. The total number

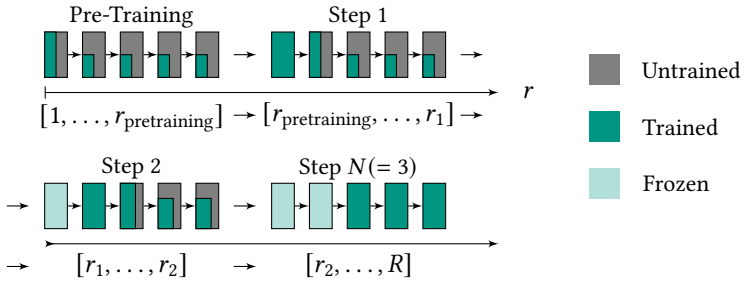


Figure 6.4: Visualization of the SLT training scheme with an exemplary 5-layer NN. Training is initiated with pre-training for $r_{\text{pretraining}}$ rounds, followed by training for $r_1 - r_{\text{pretraining}}$, $r_2 - r_1$, and $R - r_2$ rounds in configurations 1, 2, and N $(= 3)$, respectively.

of trained parameters (i.e., parameters that are uploaded) can be computed using $q(A)$ (indices for four-dimensional parameters are omitted):

$$q(A) = q(\{K_F, K_T, s\}) = \left(\sum_{k \in \{k : K_F < k \leq K_T\}} P_k V_k \right) + P_{K_T+1} \lfloor s V_{K_T+1} \rfloor + \sum_{k \in \{k : K_T+1 < k \leq K\}} \lfloor s P_k \rfloor \lfloor s V_k \rfloor. \tag{6.6}$$

Using q, the share of rounds R_n for a step n can be determined. For pretraining, i.e., step 1,

$$R_1 = \frac{q(\{0, 1, s\})}{q(\{0, 0, 1\})} - R_{\text{pretraining}}. \tag{6.7}$$

For all steps $n > 1$, the step-specific rounds R_n are calculated using:

$$R_n = R \cdot \frac{q(\{n-1, n, s_n\}) - q(\{n-2, n-1, s_{n-1}\})}{q(\{0, 0, 1\})}. \tag{6.8}$$

Lastly, the switching point for pretraining is defined as $r_{\text{pretraining}} = R_{\text{pretraining}}$, and for all steps n,

$$r_n = R \cdot \frac{q(\{n-1, n, s_n\})}{q(\{0, 0, 1\})}. \tag{6.9}$$

6.3.3 Extension to Heterogeneous Memory Constraints

As the memory available for training on devices can be heterogeneous, we extend SLT to accommodate heterogeneous memory constraints. Specifically, we support heterogeneity through the following mechanism: Devices with the most restrictive constraints perform training as in the homogeneous case, using s as described in eq. (6.5). Devices with fewer constraints use the same scaling factor s_n per configuration to ensure that all devices train

Algorithm 3: Successive Layer Training: $\overline{\boldsymbol{w}}$ and $\boldsymbol{w}$ label the sets of all layer parameters.

Requires: Number of rounds R, devices C, number of devices per round $|C^{(r)}|$, configurations $A_1, \ldots, A_N$, where $n \in [1, \ldots, N]$, satisfying the memory constraint M, and initialized parameters $\boldsymbol{w}^{(1)}$

Server:

- **foreach** round $r = 1, 2, \ldots, R$ **do**
  - $C^{(r)} \leftarrow$ select $|C^{(r)}|$ random devices out of C
  - $\overline{\boldsymbol{w}}^{(r)}, A^{(r)} \leftarrow \text{ConfigurationSelection}(\boldsymbol{w}^{(r)}, r)$
  - **foreach** *device* $c \in C^{(r)}$ *in parallel* **do**
    - $\overline{\boldsymbol{w}}^{(r)}, A^{(r)}$ receive from server
    - $\overline{\boldsymbol{w}}^{(r,c)} \leftarrow \text{DeviceTraining}(\overline{\boldsymbol{w}}^{(r)}, A^{(r)})$
    - upload $\overline{\boldsymbol{w}}_j^{(r,c)}$ to server $\forall j \in \{j : K_F < j \leq K\}$
  - $\overline{\boldsymbol{w}_j}^{(r+1)} \leftarrow \frac{1}{|C^{(r)}|} \sum_{c \in C^{(r)}} \overline{\boldsymbol{w}_j}^{(r,c)}$ `// averaging of trained layers`
  - $\boldsymbol{w}^{(r+1)} \leftarrow \overline{\boldsymbol{w}}^{(r+1)}$ `// layers get embedded into server model`

ConfigurationSelection $(\boldsymbol{w}, r)$**:**

- $A \leftarrow \text{LookupTable}(r, \boldsymbol{w})$
- $\overline{\boldsymbol{w}} \leftarrow \boldsymbol{w}$ `// scaling down NN head based on configuration`
- Return $\overline{\boldsymbol{w}}, A$

DeviceTraining $(\overline{\boldsymbol{w}}, A)$**:**

- freeze $\overline{\boldsymbol{w}_j}$ for $j \in \{0 < j \leq K_F\}$ according to A
- **foreach** local minibatch b **do**
  - $\overline{\boldsymbol{w}} \leftarrow \overline{\boldsymbol{w}} - \eta \nabla L(\overline{\boldsymbol{w}}, b)$
- Return $\overline{\boldsymbol{w}}$

the same number of parameters within a layer. This ensures that devices with tighter constraints still train all parameters of the head, without leaving some of the head's parameters unconsidered on the server. However, this results in underutilized memory on the less constrained devices. To address this, less constrained devices freeze fewer layers, allowing them to train more layers at full width. For a given K_T, s_n, and A_n—determined by the device with the most restrictive constraint—the remaining memory of a less constrained device c is utilized by minimizing K_F, such that

$$\min \; K_F \;\; \text{s.t.} \;\; m(A_n) \leq M_c, \tag{6.10}$$

where M_c is the memory budget of the less constrained device. In summary, less constrained devices only adapt K_F while keeping s_n the same as that of the most constrained device.

6.4 Experimental Evaluation

6.4.1 Experimental Setup

The proposed technique, SLT, is evaluated in an FL setting using PyTorch [104], where data from CIFAR10, CIFAR100, FEMNIST from the Leaf benchmark [108], and TinyImageNet are assigned to each device $c \in C$, such that each device c has its own private dataset $\mathcal{D}_c$ of equal size. In each round r, a subset of devices $C^{(r)}$ is picked randomly. For training, SGD with a momentum of 0.9 and an initial learning rate of $\eta = 0.1$ is used, with cosine decay applied to reduce η to 0.01, and a weight decay of 1.0×10^{-5}. The vision models ResNet20, ResNet44, and DenseNet40 are selected for evaluation. Each experiment is repeated, such that the presented results are the average of 3 independent seeds, along with the standard deviation after R rounds of training. For CIFAR10, CIFAR100, and TinyImageNet, a scenario is evaluated with $|C| = 100$ devices, where in each round $|C^{(r)}| = 10$ devices are actively participating. For FEMNIST, $|C| = 3550$, and $|C^{(r)}| = 25$ devices are picked. FL training is performed for $R = 2500$ rounds for CIFAR10, CIFAR100, and TinyImageNet. For FEMNIST, $R = 1000$ rounds are conducted. Lastly, for ImageNet, due to its higher complexity and size, $|C| = 500$, and $R = 4000$ rounds are selected. In each round, each device iterates over its local dataset once. Standard data augmentation techniques, such as random cropping and horizontal and vertical flips, are applied (flips are omitted for FEMNIST).

6.4.2 Memory Footprint during Training

In general, the high memory requirements in training NN can be attributed to three groups: Firstly, the NN's weights have to be stored in memory during both the forward and the backward pass. Additionally, stateful optimizers require additional memory. Secondly, in order to calculate the gradients in the backward pass, activation maps of the layers have to be kept in memory. Lastly, the calculated gradients have to be kept in memory before applying the gradient step. In state-of-the-art convolutional NNs, the activations account for most of the memory requirements, while gradients and weights represent a minority of the memory footprint. For example, for ResNet44 and DenseNet40, it can be measured that activations make up $\sim$ 99% of the required memory, while the remaining 1% can be attributed to weights and gradients.

Existing baselines, such as FD and FedRolex, reduce the weights by scaling down the channels. While reducing s results in a quadratic reduction of parameters, it only results in a linear decrease in activations, thus scaling s only linearly reduces peak memory. For SLT and the baselines, the maximum amount of memory required during training is determined by counting the size of the activation maps, as well as gradients and weights. In the case that layers are frozen (i.e., F_F), it is only required to load the parameters into memory, while the memory of the activation maps in the forward pass can immediately be released. For the fully trained part of the NN (i.e., F_T), parameters, gradients, and

activation maps have to be kept in memory. Lastly, for the downscaled head (i.e., F_H), memory is almost linearly reduced with the head's scale s.

Measurement of peak memory in PyTorch: Measurements of peak memory can be done offline, prior to training, and do not require any private data from the devices. In more detail, the measurements are performed using the following PyTorch mechanisms:

- Measurement of weights: The size of the weights can be measured by summing up all tensors that are present in the `state_dict`.
- Measurement of activations and gradients: To measure the size of the activations that have to be kept in memory, `backward_hooks` are used. These hooks are attached to all relevant PyTorch modules in the NN, such as `Conv2d`, `BatchNorm2d`, `ReLU`, `Linear`, and `Add` operations. If a hook attached to a module is called, the respective size of the activation map and gradient is added to a global variable. The total contribution of the activations and gradients is the sum of the individual values.

We evaluate memory constraints by scaling down s in FedRolex and FD using $s_{\text{FD/FedRolex}} \in [0.125, 0.25, 0.5, 1.0]$ for experiments with ResNet, and $[0.33, 0.66, 1.0]$ for DenseNet. To that end, similar to SLT, we introduce a function $m_{\text{subset}}(s_{\text{FD/FedRolex}})$, which maps the scaling factor $s_{\text{FD/FedRolex}}$ to a memory measurement. We observe that, for DenseNet, SLT only enables a reduction of 3×, as some layers in DenseNet have significantly larger activation maps than others, which limits SLT's effectiveness. For SLT, suitable configurations A_n are acquired, specifically by maximizing s_n to use the same memory as the baselines, using

$$\max\ s_n\ \text{ s.t. }\ m(A_n) \leq m_{\text{subset}}(s_{\text{FD/FedRolex}})\ \wedge\ s_n \leq s_j\ \ \forall j \in [n+1, \ldots, N]. \tag{6.11}$$

For heterogeneous memory constraints, we introduce several levels of memory constraints. For example, $\boldsymbol{s}_{\text{FD/FedRolex}} = [0.125, 0.25]$ describes a setup where 50% of the devices train with $s_{\text{FD/FedRolex}} = 0.125$, while the remaining 50% train with $s_{\text{FD/FedRolex}} = 0.25$. Similarly, we evaluate different granularities of constraints by using $\boldsymbol{s}_{\text{FD/FedRolex}} = [0.125, 0.25, 0.5]$ and $\boldsymbol{s}_{\text{FD/FedRolex}} = [0.125, 0.25, 0.5, 1.0]$, where each memory constraint is applied to 33% and 25% of the devices, respectively.

In addition to FD, FedRolex, and a small model baseline, HeteroFL [78] and FjORD [79] are evaluated. Both methods require that a share of devices is capable of training the server NN end-to-end. Therefore, we reduce the size of the server NN by scaling down its layers, allowing these techniques to fulfill this requirement.

6.4.3 Experimental Results

Iid results: In the iid case, results are presented in table 6.1. It can be observed that SLT reaches significantly higher accuracy for ResNet20 and CIFAR10 under all evaluated constraints, outperforming the small model baseline by up to 7.8 p.p. and the state of the art by 52.4 p.p.. In the case of FEMNIST, it can be seen that a small model already

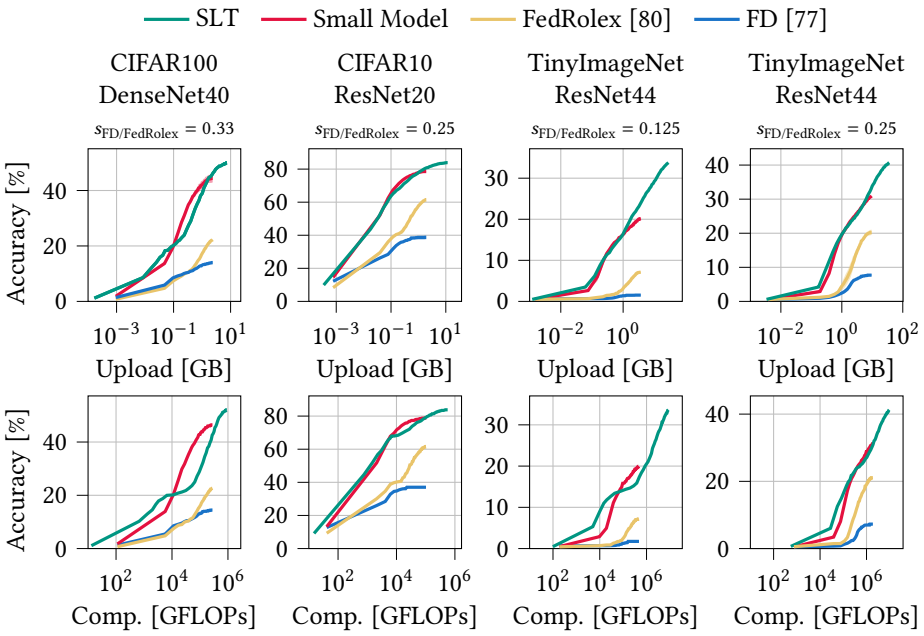


Figure 6.5: Highest accuracy reached in training in % over resources spent, specifically upload and computations performed (FLOPs), for CIFAR10, CIFAR100, and TinyImageNet with DenseNet, ResNet20, and ResNet44 using iid data.

has sufficient capacity for the complexity of the problem. Thus, only a few percentage points separate $s_{FD/FedRolex} = 0.125$ from the full model (i.e., where $s_{FD/FedRolex} = 1.0$). As a consequence of this small accuracy gap between these models, SLT provides only a minor benefit over the small model baseline. The opposite effect can be observed with CIFAR100 and TinyImageNet, where the small model baseline loses up to 25.4 p.p. compared to the full unscaled model. Additionally, it can be observed that for low values of $s_{FD/FedRolex}$, FD and FedRolex fail to learn useful representations at all. In these cases, SLT improves upon the small model by 13.7 p.p. and outperforms the state of the art by 26.6 p.p..

non-iid results: Typically, data in FL is not iid but rather non-iid. Therefore, the same experiments are repeated with non-iid data and presented in table 6.1. Similar to [119], the rate of non-iid-ness is controlled by a Dirichlet distribution with hyperparameter α. For all experiments, α is set to 0.1. From table 6.1, it can be observed that the small model baseline, in the case of CIFAR10 and FEMNIST, loses accuracy compared to the full-scale model. As this occurs, the gain of SLT increases. CIFAR100 and TinyImageNet show a similar, proportional drop in accuracy across all evaluated algorithms, where SLT outperforms other techniques by a significant margin.

Table 6.1: Results for SLT and baselines with iid and non-iid experiments for ResNet20, ResNet44, ResNet56, and DenseNet on CIFAR10, CIFAR100, FEMNIST, TinyImageNet, and ImageNet. Accuracy is given in % with the mean and standard deviation over three independent seeds after R rounds of training.

$s_{\text{FD/FedRolex}}$	0.125	0.25	0.5	1.0	0.125	0.25	0.5	1.0			
iid	ResNet20/CIFAR10				ResNet20/FEMNIST						
SLT	**74.1±0.8**	**83.4±0.2**	**85.2±0.6**		**84.4±0.3**	**85.8±0.1**	**86.9±0.0**				
Small Model	66.3±0.3	78.2±0.4	84.6±0.4	87.5±0.6	82.3±0.4	85.5±0.1	86.9±0.0	87.6±0.0			
FedRolex [80]	21.7±1.9	61.0±0.5	80.6±0.6		42.1±6.0	71.4±2.1	83.0±0.1				
FD [77]	19.0±1.4	36.7±1.8	71.6±0.4		38.9±3.7	70.4±0.2	83.4±0.3				
non-iid	ResNet20/CIFAR10				ResNet20/FEMNIST						
SLT	**52.4±0.9**	**69.6±0.6**	**75.5±1.3**		**81.2±1.6**	**83.0±2.0**	**83.8±1.9**				
Small Model	44.7±1.2	63.1±0.7	73.6±0.6	80.5±1.3	79.6±0.8	82.9±1.1	83.3±2.4	84.0±1.9			
FedRolex [80]	15.0±3.7	29.8±1.7	48.3±2.9		39.4±2.0	59.3±2.1	78.5±0.5				
FD [77]	11.3±0.9	10.7±0.6	34.9±5.7		15.9±8.2	51.0±1.2	79.7±1.1				
$s_{\text{FD/FedRolex}}$	0.125	0.25	0.5	1.0	0.125	0.25	0.5	1.0	0.33	0.66	1.0
iid	ResNet44/TinyImageNet				ResNet56/ImageNet				DenseNet40/CIFAR100		
SLT	**33.5±0.1**	**40.3±0.5**	**42.3±0.2**		**24.2±0.2**	**31.2±0.3**	**34.6±0.1**		**51.1±0.0**	53.3±0.6	
Small Model	19.8±0.3	30.6±0.3	40.2±0.3	45.2±0.1	8.9±0.2	18.4±0.0	30.3±0.1	41.6±0.5	43.9±1.5	**55.9±0.1**	60.2±0.5
FedRolex [80]	6.9±0.2	19.8±0.8	33.1±0.2		3.4±0.2	11.4±0.3	21.3±0.5		22.2±0.3	46.7±0.1	
FD [77]	0.9±0.0	7.1±0.1	25.9±0.6		0.3±0.2	6.2±0.2	16.8±0.5		13.5±0.5	41.9±1.5	
non-iid	ResNet44/TinyImageNet				ResNet56/ImageNet				DenseNet40/CIFAR100		
SLT	**28.5±1.2**	**35.1±1.1**	**36.1±0.2**		**21.7±0.9**	**29.7±0.3**	**31.8±0.4**		**45.9±1.4**	48.4±0.5	
Small Model	16.9±0.2	25.3±0.5	34.2±0.4	39.0±0.8	8.4±0.3	16.2±0.2	27.3±0.3	38.7±0.3	40.5±1.2	**51.8±0.2**	55.8±0.5
FedRolex [80]	1.5±0.5	12.7±1.3	26.1±0.5		2.8±1.1	10.2±0.9	18.4±0.4		20.0±0.3	42.9±0.4	
FD [77]	0.4±0.1	0.6±0.0	20.0±1.3		0.1±0.0	0.1±0.0	15.7±0.4		7.6±0.1	36.9±1.0	

Table 6.2: Results for SLT and baselines with heterogeneous constraints and non-iid data. Accuracy in % after R rounds of training is reported.

Setting	DenseNet40/CIFAR100		ResNet44/TinyImageNet		
$\boldsymbol{s}_{\text{FD/FedRolex}}$	[0.33, 0.66]	[0.33 - 1.0]	[0.125, 0.25]	[0.125 - 0.5]	[0.125 - 1.0]
SLT	**46.4±2.0**	**49.3±1.8**	**30.3±1.2**	**33.0±0.5**	**35.9±0.4**
Small Model	40.5±1.2	40.5±1.2	16.9±0.2	16.9±0.2	16.9±0.2
FedRolex [80]	33.2±0.4	43.9±1.3	5.4±0.2	13.8±1.3	23.6±0.7
FD [77]	21.2±0.6	38.1±0.4	0.5±0.1	0.6±0.1	20.6±1.7
HeteroFL [78]	42.2±1.3	42.8±0.5	20.7±0.7	24.1±0.2	23.3±0.5
FjORD [79]	38.7±0.4	36.9±0.4	22.4±1.0	25.3±0.3	27.5±0.8

Communication, computation, and convergence speed: We evaluate SLT as well as the baseline with respect to convergence speed, i.e., the gain in accuracy relative to the communication spent and computation performed. In fig. 6.5, it can be seen that SLT converges as fast as a small model while reaching higher final accuracies. Compared to FD and FedRolex, SLT requires significantly less communication and computation to reach a specific accuracy level.

Heterogeneous memory constraints: To evaluate SLT with heterogeneous memory constraints, experiments with TinyImageNet and CIFAR100 are repeated and presented in table 6.2. It can be observed that SLT outperforms all other evaluated techniques in all scenarios. Contrary to the homogeneous case, FedRolex improves upon a small model in some cases; however, this is not the case with FD. For FjORD and HeteroFL, both techniques outperform the small model baseline. However, higher granularity of resources lowers the accuracy of both. For example, in the case of HeteroFL with ResNet44, using 4 constraint levels results in lower accuracy than using 3 levels, despite the fact that the average resources are higher with 4 levels (i.e., $\mathbb{E}[\boldsymbol{s}_{\text{FD/FedRolex}}] \approx 0.47$) compared to the case with 3 levels (≈ 0.29). A similar behavior can be observed with FjORD and DenseNet40. As both techniques, in principle, share the subset mechanism of FD in the heterogeneous case, they suffer from supporting more constraint levels, which causes less co-adaptation between the parameters.

6.4.4 Ablation Study of Maximizing s_n over F_T

To justify the design choice to maximize s_n for all training steps n over F_T, we conduct an ablation study to investigate the trade-off between s_n and F_T. In these experiments, fractions of the maximized s_n are used, labeled as s_{ablation}. The FL training is evaluated for different values of s_{ablation}, specifically using $\frac{s_{\text{ablation}}}{s_n} \in (0, 1]$. If only a fraction of the maximized s_n is used, the available memory is naturally underutilized. To highlight the trade-off between s_n and F_T, the remaining available memory is utilized by increasing F_T. The

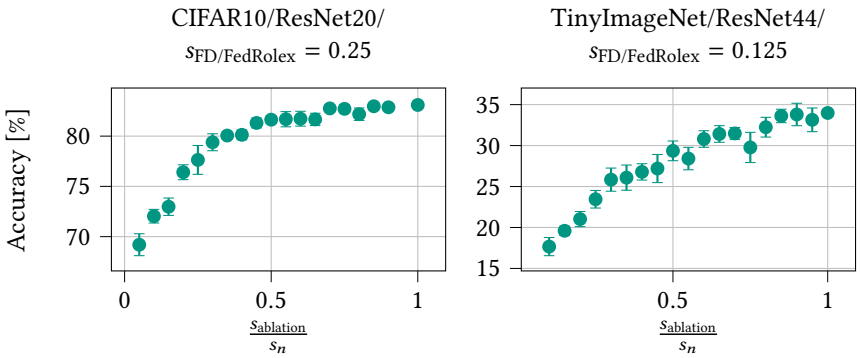


Figure 6.6: Visualization of accuracy in % of the trade-off between maximizing s over F_T for CIFAR10 and TinyImageNet with ResNet20 and ResNet44, with memory reduction of 4× and 8×.

Table 6.3: Steps N in SLT for various constraints $s_{\text{FD/FedRolex}}$ and NN models.

Constraint $s_{\text{FD/FedRolex}}$	ResNet20	ResNet44	DenseNet40
0.66	-	-	10
0.5	8	14	-
0.33	-	-	15
0.25	14	26	-
0.125	16	14	-

results of experiments conducted with CIFAR10/ResNet20 and TinyImageNet/ResNet44 are visualized in fig. 6.6. Apart from the selection of s and F_T, all hyperparameters are kept the same. The results show that increasing s_{ablation} results in an almost monotonically increasing accuracy for both CIFAR10 and TinyImageNet. This increase in accuracy suggests that maximizing s_n over F_T is preferable, meaning that training a downscaled head with a larger width is more effective than training a larger number of layers at full width with an extremely downscaled head. In both studied cases (CIFAR10 and TinyImageNet), maximum accuracy is reached when $s_{\text{ablation}} = s_n$, which is the configuration used in SLT throughout the mainline experiments in section 6.4.3. As a consequence of maximizing s_n, the NN's layers have a more uniform width, which appears to be favorable in terms of achievable accuracy.

6.4.5 Ablation Study of Mapping of Steps N to Rounds R

In general, any layer trained in SLT should receive a sufficient amount of training, such that it can extract useful features for downstream layers. At the same time, early layers should not overfit in the current configuration, as more capacity in the form of parameters is added throughout the training. Since the number of rounds R is limited, the transition

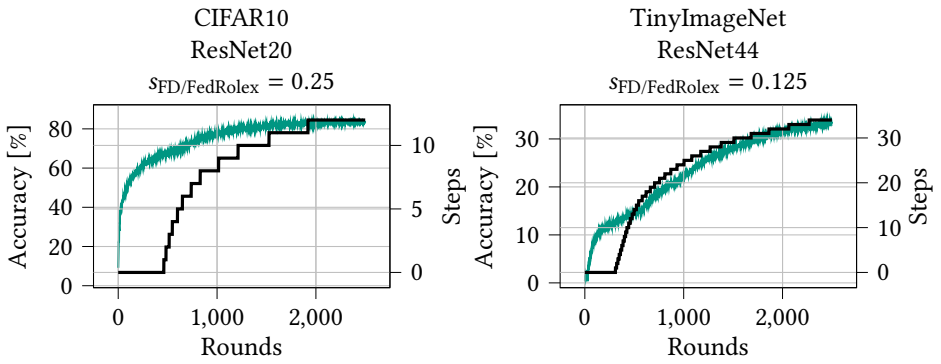


Figure 6.7: Accuracy in % over rounds (green) and steps over rounds (black) with SLT for ResNet20, ResNet44, and DenseNet40 using CIFAR10, CIFAR100, and TinyImageNet.

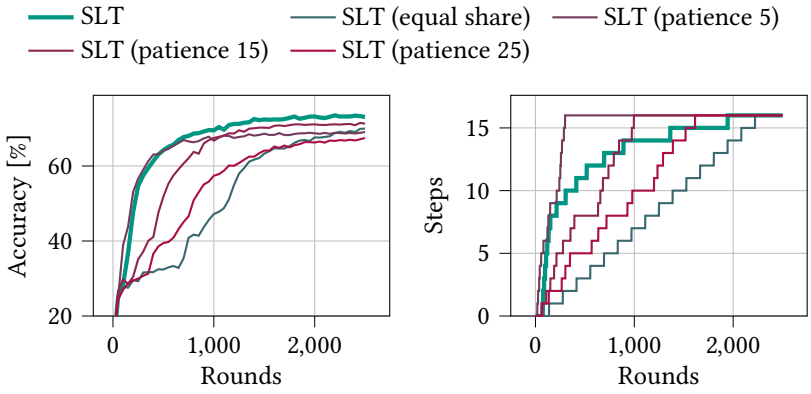


Figure 6.8: Visualization of different strategies to map N to R using ResNet20 with CIFAR10. The left plot shows accuracy over rounds in %, while the right plot shows steps over rounds.

from one configuration (a total of N steps) to the next has to be mapped to the available rounds R. If too many rounds are spent on the first configurations (i.e., the early layers), these layers might overfit to the problem and fail to provide useful upstream features. Furthermore, the number of rounds spent on the later layers will be limited. If too few rounds are spent on the early layers, they might also fail to provide useful upstream features, i.e., their capacity is inefficiently used as they have not received enough training. Therefore, a trade-off is required to optimally train each layer. In SLT, steps N are mapped proportionally to the amount of added parameters (that were previously untrained) relative to N. Since N depends on how many steps are required to reach $s = 1$, the number of steps depends on the constraint. Table 6.3 lists the number of steps N for different ResNets and DenseNet models at various constraint levels. The mapping of steps N to rounds R for CIFAR10, CIFAR100, and TinyImageNet is visualized in fig. 6.7, where accuracy over

rounds and steps over rounds (black) are displayed. This proportional mapping has several advantages over other techniques. Most importantly, it depends only on the total number of rounds R and the NN structure, but not on run-time factors such as the current accuracy. Hence, it can be calculated prior to training. We compare SLT's mapping scheme against two other mapping schemes:

1. Equal share: This mapping scheme equally distributes all rounds across all steps. The scheme can be calculated offline.
2. Early stopping: In this scheme, the decision of when to switch is made during training, thus requiring device measurements or a suitable server-side test set. If the test accuracy on the server does not improve after a specific number of FL rounds, the mapping switches to the next configuration (i.e., n is increased by one). The number of rounds without improvement is usually referred to as *patience*. We evaluate with patience values of 5, 15, and 25.

To compare the mapping schemes, experiments with CIFAR10/ResNet20 are conducted, where everything except for the mapping scheme is kept consistent with the mainline experiments. It can be seen from fig. 6.8 that SLT's mapping scheme outperforms both the equal share and early stopping approaches, reaching higher final accuracy as well as faster convergence throughout the training. With equal share, it can be observed that the accuracy saturates quite early in the training, as too many rounds are spent on the early layers. Conversely, the last layer receives too little training, resulting in inferior final accuracy compared to SLT's mapping. Early stopping, when carefully tuned, can converge as fast as SLT's mapping but falls behind in terms of final accuracy. Additionally, it requires test data accuracy during training to decide when to switch. In summary, the proportional mapping of SLT achieves the highest final accuracy while converging the fastest compared to other mapping schemes. Moreover, it can be calculated offline.

6.5 Summary

This chapter proposed SLT, a technique that allows heterogeneous memory-constrained devices to train a specific NN model that no participating device can train end-to-end. The presented experiments show that existing techniques addressing this problem perform worse than an end-to-end smaller model. One reason existing techniques fail to improve upon this baseline is that they train subsets of the full-scale NN within a round, thereby optimizing a share of parameters without consideration of already trained parameters on the server, resembling regular dropout techniques. As a consequence, the co-adaptation among these trained subsets is limited, causing redundantly trained features and hurting the global accuracy of the NN model. Based on this observation, the proposed SLT technique is designed to avoid this behavior by successively adding parameters throughout the training without removing already trained parameters locally. To simultaneously adhere to memory constraints, early layers are successively frozen, eliminating the need to

hold their activation maps in memory. This procedure allows for enhanced co-adaptation among parameters while respecting memory constraints and enables the FL training process to significantly outperform both the state of the art and the small model baseline.

7 FL for Fine-Tuning NNs with Constrained Devices

7.1 Scope and Contribution

In the previous chapters 5 and 6, it is assumed that devices in FL have to train the given NN structure from scratch (i.e., with parameters in a randomly initialized state) on resource-constrained devices. As a consequence, every layer eventually has to be trained by some device so that it contributes to the learning goal and its capacity is not wasted. In CoCoFL (chapter 5), a share of devices has to be able to train the NN end-to-end, or at least be capable enough to propagate the error back to the first layer of the NN, which is computationally expensive. In chapter 6, the NN layers are successively trained to ensure that all layers are eventually trained.

However, for some FL tasks, it can be assumed that a public dataset is available for centralized training, which shares some commonality with the FL task. By pretraining an NN on such a dataset, the convergence speed of the FL task can be significantly increased. This is particularly true in the text domain, where tasks like text classification benefit greatly from pretraining on large quantities of generic text. Language structures learned during pretraining, such as grammatical patterns and common semantic relationships, generalize well to downstream tasks, as these structures remain consistent across different text domain problems. Similarly, in object classification, generic vision datasets enable layers in the NN to detect general features learned from broader data distributions, such as edges and corners. The pretrained states of the NN layers provide useful features for downstream layers in the FL task. Consequently, the early layers of the NN do not necessarily need to be trained during FL training.

This chapter presents an extension of the work in chapter 5. Specifically, this chapter studies how pre-trained transformer architectures can be fine-tuned using FL with resource-constrained devices. To adapt large NNs, e.g., LLMs, several techniques have been proposed, such as Adapter [120] and LoRA [84], which mainly allow fine-tuning a large model to a downstream task in a parameter-efficient way. However, while being parameter-efficient, it can be observed that techniques like LoRA primarily reduce the memory overhead of gradients and optimizer states but do not address activation memory.

This chapter is mainly based on [1, 5].

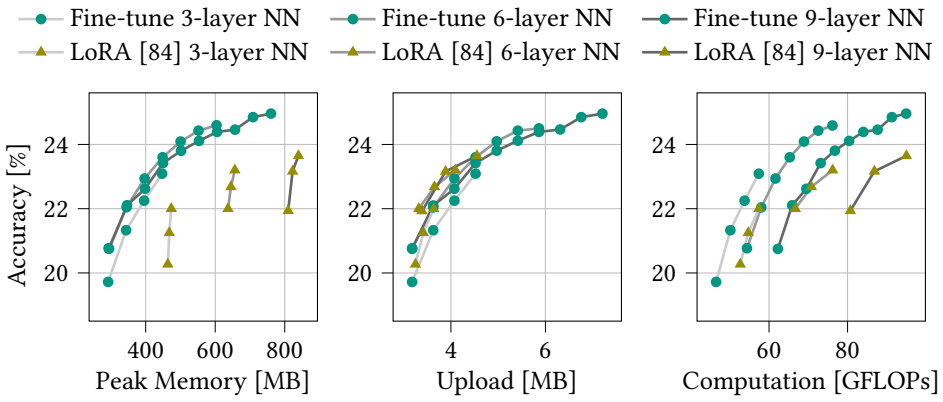


Figure 7.1: Comparison of downstream accuracy in % after 75 rounds on Shakespeare using layer fine-tuning (layers frozen from first to last) and LoRA with different ranks (24, 12, 3). The figure shows that, in most cases, layer fine-tuning provides better accuracy-resource trade-offs than LoRA.

This is particularly relevant for significantly smaller models, where activation memory dominates the total memory usage, and weights and optimizer states contribute comparatively little. The reason for this is that while weight memory increases quadratically with the embedding size, activation memory increases only linearly with the embedding size. In fig. 7.1, we compare tiny transformers ($3-12$ layers, 3 heads, with an embedding size of 92) that were pre-trained on OpenWebText [113] and trained in a federated manner on Shakespeare [108] to perform next-token prediction. The plots suggest the following conclusion: LoRA can achieve similar communication savings compared to fine-tuning (i.e., training only the last layers of the NN, a special case of CoCoFL (chapter 5)). However, LoRA requires significantly more memory to reach the same accuracy compared to fine-tuning the last layers. Similarly, with respect to computation, LoRA requires more FLOPs as a complete forward and backward pass is required.

Based on these observations, this chapter proposes an extension of CoCoFL, where we assume that NNs are pre-trained with a suitable dataset. Further, we assume that the NN structure is not fixed but can be selected from a set of reasonable choices. Based on the observations of this scenario in fig. 7.1, a strategy is proposed that, through optimal NN architecture selection and layer fine-tuning, allows achieving the highest accuracy while adhering to given device constraints in training. In summary, this chapter introduces the following novel contributions:

- The work in this chapter is the first to study resource-constrained cross-device FL with the availability of pre-trained models. We observe that layer fine-tuning surpasses existing vanilla LoRA [84] strategies as well as recent adaptations for FL with heterogeneity [83]. Furthermore, with pre-trained NNs, layer fine-tuning

also surpasses existing FL techniques [78, 79, 82, 81, 80] that focus on resource-constrained devices.

- In existing works, it is always assumed that the NN structure is given. By contrast, in this chapter, it is assumed that a set of feasible pre-trained structures exists. We propose *Constraint-Aware Federated Finetuning (CAFF)*, an architecture selection technique that achieves the highest accuracy given a device constraint. Existing state-of-the-art methods and CAFF are evaluated in various settings. The results show that, in most of these settings, CAFF outperforms existing techniques, especially when fairness is evaluated in rc-non-iid scenarios.

7.2 Problem Statement

System model: The technique described in this chapter, similar to chapter 5, focuses on a synchronous cross-device FL setting. Differently from chapter 5, where it is assumed that there is a given, fixed NN structure, here it is assumed that there exists a set of NN structures $\mathcal{F}$ from which the FL system can choose. Further, it is assumed that the NNs are not trained from scratch in the FL training but are already pre-trained on suitable data that is centrally available.

Device model: Participating devices are assumed to have heterogeneous resources and are specifically constrained in their FLOPs per FL round, their memory available for training, and the per-round communication. Without loss of generality, we assume that these constraints are static and known to the server prior to the FL training. Similarly to chapter 5, the available memory of a device c is characterized by M_c, the communication by Q_c, and the number of FLOPs per round by H_c. An NN architecture $F \in \mathcal{F}$ is parametrized by the number of stacked layers K. In contrast to chapter 5, the training configuration A now specifically incorporates the training NN topology, which is parametrized by K. The number of FLOPs of a configuration can be calculated using $h : \mathcal{A} \mapsto \mathbb{R}$, where $\mathcal{A}$ represents the set of all configurations A.

Objective: Given the set of pre-trained NN structures, the main objective is to select an NN model (from the set $\mathcal{F}$) that maximizes the final accuracy of the server model $\boldsymbol{w}^{(R)}$ after R rounds of training, while adhering to per-round fixed FLOP, memory, and communication constraints:

$$\begin{aligned} &\text{maximize accuracy}(\boldsymbol{w}^{(R)}) \quad && \forall c \in C \\ &&& \text{s.t. } h(A_c) \leq H_c \wedge m(A_c) \leq M_c \wedge q(A_c) \leq Q_c \end{aligned} \tag{7.1}$$

As a secondary metric, group-sensitivity, as defined in eq. (2.20), is evaluated after R rounds.

7.3 Methodology

Heterogeneous freezing of pre-trained NNs: Each NN architecture $F \in \mathcal{F}$ is parametrized by K, which represents the number of stacked layers of the NN architecture. This chapter considers the NNs to be generally pre-trained with a suitable server dataset; hence, early layers provide suitable upstream features without being adjusted by the FL process. The number of layers trained by a device c is referred to as K_T, such that the NN architecture F_K, parametrized by K, can be described as $F_K = F_F \circ F_T$,

$$F_F = \bigcirc_{k \in \{0<k<K-K_T\}} f_k \qquad F_T = \bigcirc_{k \in \{K-K_T \leq k \leq K\}} f_k, \tag{7.2}$$

where F_F represents the frozen part of the NN, and F_T, parametrized by K_T, represents the trained part of the head that receives fine-tuning through the FL process. A training configuration $A = \{K, K_T\}$ is therefore parameterized by the NN architecture, parameterized by K, and the number of trained layers K_T. Using this definition, eq. (7.1) can be adjusted s.t. each device c obeys the constraints through a common selection of K and a per-device selection of K_T

$$h(K, K_T) \leq H_c \wedge m(K, K_T) \leq M_c \wedge q(K, K_T) \leq Q_c. \tag{7.3}$$

NN-architecture selection with heterogeneous devices: Since it is assumed that each device $c \in C$ is required to participate in the FL, a feasible NN structure has to be picked that allows all devices to train at least a single layer of the NN. Based on the constraints of the individual devices, such a subset of $\mathcal{F}$ can be acquired by applying

$$\begin{aligned} \mathcal{F}_{\text{feasible}} \leftarrow \{F_K \in \mathcal{F} \mid \forall c \in C (\exists K_T) \\ [h(K, K_T) \leq H_c \wedge m(K, K_T) \leq M_c \wedge q(K, K_T) \leq Q_c]\}. \end{aligned} \tag{7.4}$$

Based on observations from preliminary experiments in fig. 7.1, we propose the following architecture selection scheme: The NN architecture is selected such that it maximizes the average number of layers being trained, given the heterogeneous device capabilities, using

$$\hat{F}_K = \text{argmax}_{F_K} \left\{ \frac{1}{|C|} \sum_{c \in C} K_T \;\middle|\; F_K \in \mathcal{F}_{\text{feasible}} \right\}. \tag{7.5}$$

In case two NN architectures allow for the same average number of trained layers, the NN architecture that incorporates more total layers is chosen. This conditional choice can be incorporated through the introduction of a tie-breaking term:

$$\hat{F}_K = \text{argmax}_{F_K} \left\{ \epsilon K + \frac{1}{|C|} \sum_{c \in C} K_T \;\middle|\; F_K \in \mathcal{F}_{\text{feasible}} \right\}, \tag{7.6}$$

where ϵ is a small value (i.e., 1×10^{-6}) that influences the decision when there is a tie in the number of trained layers. The proposed selection scheme has specific implications

Algorithm 4: Constraint-Aware Federated Finetuning.

Requires: Total rounds R, devices C, number of devices per round $|C^{(r)}|$, device constraints M_c, H_c, Q_c, and set of pretrained NNs $\mathcal{F}$

Server:

$\mathcal{F}_{\text{feasible}} \leftarrow \{F_K \in \mathcal{F} \mid \forall c \in C(\exists K_T)[h(K, K_T) \leq H_c \wedge m(K, K_T) \leq M_c \wedge q(K, K_T) \leq Q_c]\}$

$\hat{F}_K = \text{argmax}_{F_K} \left\{ \epsilon K + \frac{1}{|C|} \sum_{c \in C} K_T \;\middle|\; \forall F_K \in \mathcal{F}_{\text{feasible}} \right\}$

$\hat{K} \leftarrow \hat{F}_K$ `// get optimal` $\hat{K}$

foreach round $r = 1, 2, \ldots, R$ **do**

  foreach *device* $c \in C^{(r)}$ *in parallel* **do**

    $\boldsymbol{w}^{(r)}$ receive from server

    $\boldsymbol{w}^{(r,c)} \leftarrow \text{DeviceTraining}(\boldsymbol{w}, \hat{K})$

    upload $\boldsymbol{w}^{(r,c)}_{\text{trained}} \leftarrow [\boldsymbol{w}^{(r,c)}_{K_T - \hat{K}}, \ldots, \boldsymbol{w}^{(r,c)}_{K}]$ to server

  foreach *layer* i **do**

    $C_i \leftarrow \bigcup_{c \in C} \{c \mid \boldsymbol{w}_i^{(r,c)} \in \boldsymbol{w}^{(r,c)}_{\text{trained}}\}$

    $\boldsymbol{w}_i^{(r+1)} \leftarrow \left(1 - \frac{|C_i|}{|C^{(r)}|}\right) \cdot \boldsymbol{w}_i^{(r)} + \frac{1}{|C^{(r)}|} \sum_{c \in C_i} \boldsymbol{w}_i^{(r,c)}$

DeviceTraining $(\boldsymbol{w}, K)$**:**

pick K_T s.t. $h(K, K_T) \leq H \wedge m(K, K_T) \leq M \wedge q(K, K_T) \leq Q$

foreach local minibatch b **do**

  $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla L(\boldsymbol{w}, b)$

Return $\boldsymbol{w}$

depending on the dominant constraint type. In cases where only memory constraints are present, it can be observed that the memory required for training is mostly independent of the total number of layers (since no activations need to be stored for frozen layers, and in smaller-scale transformers, activation memory dominates). Thus, eq. (7.6) selects the NN with the most layers. Similarly, the communication overhead (the upload) is completely independent of the total number of layers K. However, the number of computations depends on both the number of trained layers K_T and the total layers K, as the frozen forward pass induces computational overhead. Generally, for a homogeneously computationally constrained setup, it can be observed (fig. 7.1) that training more layers of a shallower NN is favorable over training fewer layers of a deeper NN structure. An ablation study of a heterogeneous setup is provided in section 7.4.2. The server-side averaging is done as described in chapter 5. Finally, the full algorithm is detailed in algorithm 4.

7.4 Experimental Evaluation

7.4.1 Experimental Setup

Pretraining: The proposed technique and the state of the art are evaluated with a set of pre-trained models $\mathcal{F}$, specifically transformer models with layers $K \in [3, 6, 9, 12]$, where each layer represents a multihead attention block with feedforward linear layers. The general structure of all F in $\mathcal{F}$ is the same; only the specific number of layers differs. The transformer NNs are parameterized with an embedding dimension of 96 and 3 heads per attention block for language modeling tasks, and 192 and 6 heads for vision tasks. For language tasks, a SentencePiece tokenizer with a vocabulary size of 8192 is used. Each architecture intended for language modeling is pre-trained for 750 K steps on OpenWebText, using a context length of 256 and a batch size of 128.

For vision tasks, the transformer NN is pretrained on ImageNet for 500 K steps with a batch size of 256. Data augmentation techniques such as random flipping, rotation, and random cropping are applied. For the NNs in the text domain, a starting learning rate of $\eta = 5 \times 10^{-4}$ is used. The learning rate is decayed over training using a cosine decay to $\eta = 5 \times 10^{-5}$. For vision tasks, $\eta = 1 \times 10^{-4}$ and $\eta = 1 \times 10^{-5}$ are used, respectively. For pre-training, the AdamW [121] optimizer ($\beta_1 = 0.9$, $\beta_2 = 0.95$) is used with a weight decay of 0.1 for linear layers. Lastly, a dropout rate of 0.05 is applied during all pre-training.

Federated training: Based on the pre-trained NNs, a downstream task is fine-tuned in a federated manner, where data from Shakespeare (using next-token prediction, based on the SentencePiece tokenization used in pre-training), CIFAR10, and CIFAR100 is distributed in equal shares to devices $c \in C$. In both cases, a device count of $|C| = 100$ is used, with $|C^{(r)}| = 10$ devices actively participating in each round. Overall, $R = 75$ rounds of training are performed. In the case of fine-tuning with Shakespeare, each device randomly samples a sentence using a context length of 256 from its local text data and trains for 8 batches with a batch size of 32. For CIFAR, each device iterates once over its local data using a batch size of 32. To maintain the same resolution with CIFAR as in pre-training, the CIFAR images are upsampled to $3 \times 64 \times 64$. CIFAR10 and CIFAR100 are distributed in an non-iid and rc-non-iid fashion using $\alpha = 1.0$ and $\alpha = 0.1$, respectively. The learning rate is decayed from $\eta = 1 \times 10^{-4}$ to $\eta = 1 \times 10^{-5}$ ($\eta = 1 \times 10^{-4}$ to $\eta = 1 \times 10^{-6}$ for vision models), and dropout is set to 0. The remaining hyperparameters of the training remain the same as in the pre-training setup. For each experiment, the mean and standard deviation of three independent seeds after R rounds of training are presented. In all experiments (including the baselines), the embedding layer remains frozen during the FL training, while the final output layer (i.e., LayerNorm and the linear layer) always receives training.

Resource constraints: Throughout this chapter's evaluation, peak memory, communication (specifically, upload) constraints, and computation constraints are considered. To

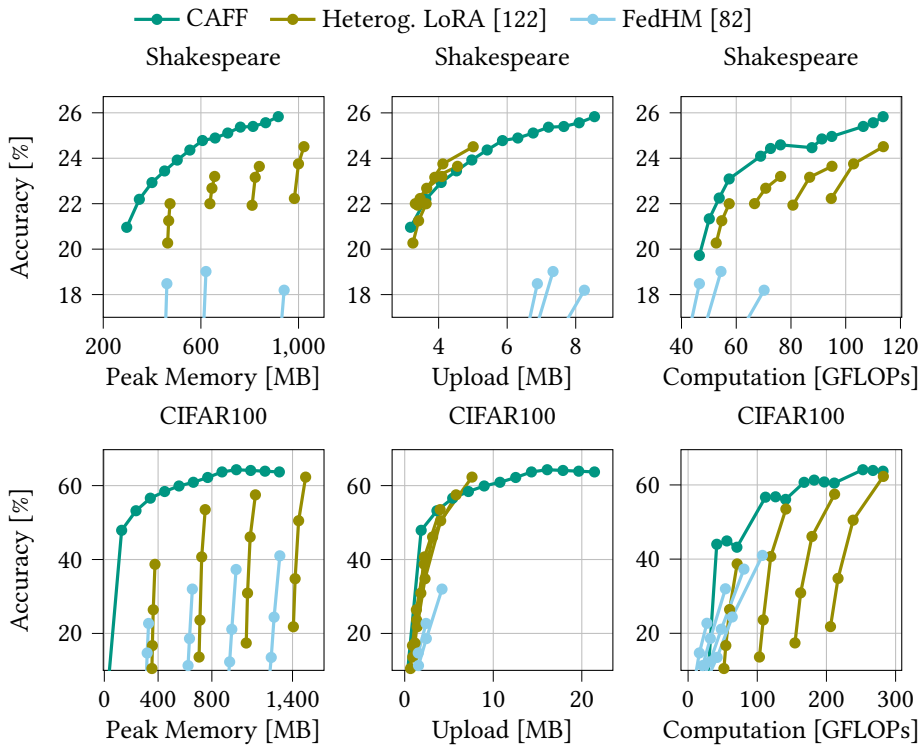


Figure 7.2: Visualization of the accuracy-resource Pareto front of CAFF, heterog. LoRA, and FedHM for Shakespeare and CIFAR100 training. For CAFF, the selection of K and K_T is performed using eq. (7.6). For the baselines, all NN topologies are evaluated.

highlight the peculiarity of individual techniques with respect to specific constraints, only one constraint type is applied in each experiment. Note that this is not a design limitation of CAFF or the baselines. The respective resource footprints are gathered similarly to chapter 6, and the computation constraint refers to a single minibatch training step.

7.4.2 Experimental Results

Homogeneous results: For FL fine-tuning, the proposed technique as well as the state of the art (Heterog. LoRA, FedHM, and FedRolex) are evaluated in a homogeneous setting (i.e., all devices have the same constraint) using Shakespeare and CIFAR100 as fine-tuning datasets. Specifically, we conduct several experiments with memory constraints of 500 MB, 700 MB, and 900 MB (400 MB, 600 MB, and 800 MB for vision models), upload constraints of 4 MB, 6 MB, and 8 MB (8 MB, 12 MB, and 16 MB for vision models), and computation constraints of 60 GFLOPs, 80 GFLOPs, and 100 GFLOPs (75 GFLOPs, 150 GFLOPs,

and 200 GFLOPs for vision models). The FLOP constraint refers to a single minibatch training step. Results for HeteroFL, FjORD, and DepthFL are omitted, as they all require a share of devices to train the NN end-to-end and would therefore degrade to vanilla FedAvg. For completeness, the range of supported constraints for each algorithm is visualized in fig. 7.2. To ensure a fair state-of-the-art comparison, state-of-the-art techniques are evaluated with each architecture $K \in [3, 6, 9, 12]$, and the NN architecture with the highest achieved accuracy is chosen for comparison in table 7.1.

In general, we observe that CAFF enables higher accuracies in most of the considered constraint settings, particularly outperforming existing techniques under memory constraints. To reach the same accuracy as CAFF, LoRA requires $2-3\times$ more memory. However, LoRA shows superior results in certain cases when it comes to communication constraints. For FedRolex, it can be observed that accuracy deteriorates when training a larger pre-trained model (i.e., not end-to-end). Figure 7.2 shows that both heterog. LoRA and FedHM, by lowering the rank for a specific architecture, affect the required communication and computation but have only a minimal impact on memory. In contrast, CAFF supports a larger range of constraints across all three types of studied resources.

Heterogeneous results: To explore how CAFF performs in an environment with heterogeneous constraints, scenarios are evaluated where devices are grouped in a 50%/50% ratio into two groups. Memory constraints are evaluated with $[400, 600]$, $[600, 800]$, and $[400, 800]$ MB ($[200, 400]$, $[400, 600]$, and $[200, 800]$ MB for vision models). Similarly, communication constraints are evaluated as $[2, 8]$ and $[4, 8]$ MB ($[2, 4]$ and $[2, 8]$ MB for vision models). Lastly, computation constraints are evaluated as $[55, 75]$ and $[57, 100]$ GFLOPs ($[75, 100]$ and $[125, 200]$ GFLOPs for vision models). Tables 7.2 and 7.3 present results for Shakespeare, CIFAR10, and CIFAR100.

We observe that across most evaluated constraint scenarios, CAFF results in higher final accuracies than the state of the art. With CAFF, most importantly, higher average resource availability leads to higher final accuracies. This is not always the case with the state of the art, as seen with Shakespeare: even though higher average resources can be utilized, many baselines actually perform worse or show no improvement. For DepthFL, it can be observed that the additional exits, especially in language modeling, significantly increase the communication and memory overhead. Hence, for many evaluated scenarios, no suitable configuration is available.

Rc-non-iid experiments: To evaluate whether the weaker group within the two groups of devices can impact the global model, the data distribution is changed from non-iid to rc-non-iid (using $\alpha = 0.1$). Specifically, the data of CIFAR100 is distributed in an rc-non-iid fashion among the two groups $\tilde{C}_1$ and $\tilde{C}_2$. To assess the impact, the group-sensitivity of the weaker group is evaluated and presented in table 7.3. It can be observed that, using CAFF, the specific classes of the weaker group are better represented in the global model, as the group-sensitivity is significantly higher than with the state of the art. However,

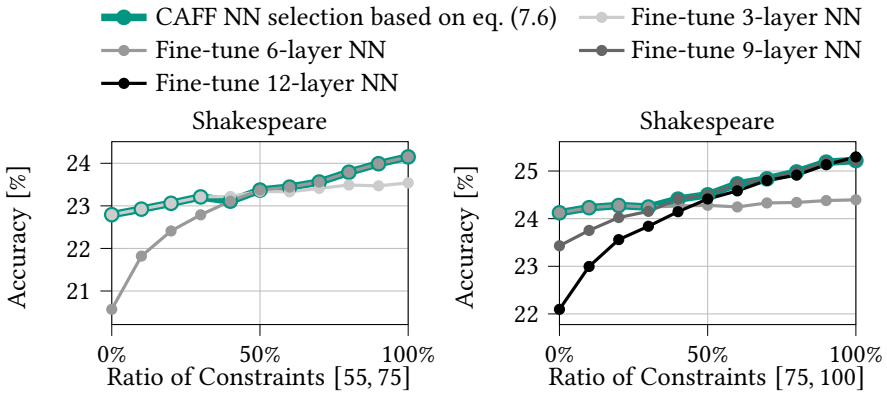


Figure 7.3: Ablation study of the NN selection scheme (eq. (7.6)). The plot shows the optimality of CAFF across different mixtures of constraints.

it is also evident that when the variance of resources increases, weaker devices are less well represented. This can be seen with memory constraints of $[200, 400]$, which perform better than $[200, 800]$ MB, as in the latter case, stronger devices have an increased impact on the model. In contrast, we observe that the state of the art—especially the shared aggregation mechanisms of heterog. LoRA, FedRolex, HeteroFL, and FjORD—is ineffective when such a distribution is present, as group-sensitivity drops by up to 20 p.p. to 30 p.p. compared to CAFF. The only exception to this is when no suitable configuration for the baselines exists (labeled with $*$, †, and ‡ in table 7.3). In these cases, the state of the art degrades to the homogeneous case. Lastly, for DepthFL, similar behavior can be observed. A potential reason for this is that weaker devices only train the early exits and have no impact on the last exit of the NN, i.e., they have little influence on the head's decision boundary.

Ablation study: To validate the assumptions made in the NN selection in eq. (7.6), the following experiment is conducted: two groups of devices with computation constraints of $[55, 75]$ GFLOPs and $[75, 100]$ GFLOPs are evaluated. Within each experiment, the ratio of devices in each group is varied. For example, a ratio of 10% refers to 10% of devices belonging to the group with the 55 GFLOPs constraint, while 90% belong to the group with the 75 GFLOPs constraint. The results of the architecture selection based on eq. (7.6) are presented in fig. 7.3, where the ratio is varied from 0% to 100%. Additionally, the accuracy of the non-selected architectures is depicted. The results show that maximizing the average of K_T across all devices maximizes the accuracy, independent of the mixing ratio. Depending on the average, eq. (7.6) switches from a 3- to a 6-, and from a 6- to a 9-layered NN, thereby maximizing the accuracy.

Table 7.1: Accuracy and standard deviation of three independent seeds in % for Shakespeare and CIFAR100 with homogeneous memory, upload, and computation constraints after R rounds of training. If no configuration is available for a given constraint, – is reported.

Setting	Peak Memory [MB]			Upload [MB]			Computation [GFLOPs]		
Shakespeare	500	700	900	4	6	8	60	80	100
CAFF	**23.4±0.1**	**24.9±0.3**	**25.6±0.3**	22.2±0.2	**24.8±0.2**	**25.4±0.3**	**23.1±0.1**	**24.6±0.3**	**25.0±0.2**
Heterog. LoRA [122]	22.0±0.3	23.2±0.5	23.6±0.3	**23.2±0.2**	24.5±0.1	24.5±0.1	22.0±0.3	23.2±0.5	23.6±0.3
FedHM [82]	18.5±0.3	19.0±0.2	19.0±0.2	–	–	19.0±0.2	19.0±0.2	19.0±0.2	19.0±0.2
FedRolex [80]	1.3±0.0	4.5±0.3	4.5±0.3	3.1±0.1	4.5±0.3	4.5±0.3	4.5±0.3	4.5±0.3	4.5±0.3
CIFAR100	400	800	1200	8	12	16	100	150	200
CAFF	**56.6±0.2**	**62.2±0.5**	**63.9±0.6**	58.4±0.2	60.9±0.1	**63.7±0.4**	**43.2±1.3**	**56.1±1.2**	**60.8±1.0**
Heterog. LoRA [122]	38.7±0.4	53.5±0.9	57.5±0.4	**62.3±0.1**	**62.3±0.1**	62.3±0.1	38.7±0.4	53.5±0.9	46.1±0.8
FedHM [82]	22.7±0.4	32.0±0.6	37.3±1.3	41.0±0.2	41.0±0.2	41.0±0.2	37.3±1.3	41.0±0.2	41.0±0.2
FedRolex [80]	1.2±0.1	1.2±0.1	1.1±0.1	1.1±0.1	1.1±0.1	1.1±0.1	1.1±0.1	1.1±0.1	1.1±0.1

– Indicates no suitable K.

Table 7.2: Accuracy and standard deviation of three independent seeds in % for Shakespeare and non-iid CIFAR100 with heterogeneous constraints after R rounds of training.

	Peak Memory [MB]			Upload [MB]		Computation [GFLOPs]	
Shakespeare	[400,600]	[600,800]	[400,800]	[2,8]	[4,8]	[55,75]	[75,100]
CAFF	**23.9±0.5**	**25.1±0.4**	**24.8±0.3**	**24.6±0.4**	**25.3±0.5**	**23.3±0.4**	**24.6±0.4**
Heterogenous LoRA [122]	–	21.9±0.3‡	–	24.4±0.3	24.7±0.3	21.9±0.2	23.3±0.2
FedHM [82]	–	18.8±0.0†	–	–	–	18.9±0.4	19.3±0.2
FedRolex [80]	16.2±0.2	15.7±0.1	16.2±0.2	–	16.3±0.3	16.2±0.2	15.7±0.1
HeteroFL [78]	22.1±0.4	23.0±0.2	22.1±0.4	–	22.0±0.5	22.1±0.4	23.0±0.2
FjORD [79]	19.8±0.3	21.5±0.1	19.8±0.3	–	19.8±0.2	19.8±0.3	21.5±0.1
DepthFL [81]	–	–	–	–	–	–	21.7±0.4
CIFAR100	[200,400]	[400,600]	[200,800]	[2,4]	[2,8]	[75,100]	[125,200]
CAFF	**52.7±0.1**	**58.8±0.2**	**57.2±0.5**	**50.6±0.2**	54.1±0.4	**42.3±1.7**	**60.2±1.1**
Heterogenous LoRA [122]	–	38.9±0.9‡	–	45.0±1.1	**55.2±0.5**	38.9±0.9‡	49.2±0.9
FedHM [82]	–	23.5±1.6†	–	–	18.0±0.6	40.1±0.6†	31.6±1.3
FedRolex [80]	33.9±1.0	40.7±0.8*	33.9±1.0	–	32.7±0.6	40.7±0.8 *	44.7±2.6
HeteroFL [78]	33.3±1.0	41.0±0.9*	33.3±1.0	–	33.4±0.5	41.0±0.9*	45.6±1.9
FjORD [79]	26.3±0.3	40.1±0.9*	26.3±0.3	–	24.2±0.5	40.1±0.9*	37.0±1.2
DepthFL [81]	26.8±1.2	41.0±0.9*	26.8±1.2	–	–	41.0±0.9*	28.7±1.3

* Degrades to vanilla FedAvg.
† Degrades to homogeneous FedHM.
‡ Degrades to vanilla LoRA.
– Indicates no suitable K.

Table 7.3: Accuracy and standard deviation of three independent seeds in % for rc-non-iid CIFAR100 and CIFAR10 with heterogeneous constraints after R rounds of training.

	Peak Memory [MB]			Upload [MB]		Computation [GFLOPs]	
CIFAR100 (Group-Sensitivity)	[200,400]	[400,600]	[200,800]	[2,4]	[2,8]	[75,100]	[125,200]
CAFF	**28.8±5.3**	**52.3±8.2**	**23.4±5.3**	**37.8±5.7**	**25.4±4.7**	**46.7± 3.7**	**40.7± 9.9**
Heterogenous LoRA [122]	–	41.3±7.5‡	–	10.5±3.2	13.5±3.8	41.3± 7.5‡	11.5± 3.5
FedHM [82]	–	26.2±6.9†	–	–	16.7±6.9	43.2± 5.7†	28.3±10.4
FedRolex [80]	8.4±2.4	44.7±8.5*	8.4±2.4	–	7.9±2.2	44.7± 8.5*	11.5± 3.9
HeteroFL [78]	8.3±2.5	44.6±8.3*	8.3±2.5	–	8.1±2.3	44.6± 8.3*	11.3± 3.9
FjORD [79]	5.8±1.5	40.7±10.2*	5.8±1.5	–	5.3±1.4	40.7±10.2*	8.2± 2.8
DepthFL [81]	5.5±1.2	44.6±8.3*	5.5±1.2	–	–	44.6± 8.3*	6.9± 1.7
CIFAR10	[200,400]	[400,600]	[200,800]	[2,4]	[2,8]	[75,100]	[125,200]
CAFF	**86.9±0.1**	**89.3±0.1**	**89.8±0.1**	85.5±0.3	87.8±0.1	**82.4± 0.2**	**91.1± 0.2**
Heterogenous LoRA [122]	–	80.1±0.1‡	–	**89.4±0.6**	**90.4±0.5**	80.1± 0.1‡	87.3± 0.5
FedHM [82]	–	71.3±1.3†	–	–	64.8±1.0	81.9± 0.2†	78.5± 0.6
FedRolex [80]	76.8±1.1	80.5±1.2*	76.8±1.1	–	77.2±1.0	80.5± 1.2*	86.8± 0.4
HeteroFL [78]	76.9±1.1	80.6±1.2*	76.9±1.1	–	77.5±1.1	80.6± 1.2*	87.1± 0.3
FjORD [79]	74.0±1.3	80.5±0.2*	74.0±1.3	–	72.6±2.0	80.5± 0.2*	83.9± 0.0
DepthFL [81]	74.3±1.1	80.5±0.2*	74.3±1.1	–	–	80.5± 0.2*	85.9± 0.2

* Degrades to vanilla FedAvg.
† Degrades to homogeneous FedHM.
‡ Degrades to vanilla LoRA.
– Indicates no suitable K.

7.5 Summary

This chapter proposed CAFF, an extension to chapter 5, which assumes that a suitable centralized dataset for pre-training is available prior to the FL process. Furthermore, this chapter loosens the assumption that the given NN structure is fixed, instead making it an optimization parameter of the resource-constrained FL process. The evaluation shows that, in such a scenario, it is sufficient to train only the last layers of the NN in the federated process while keeping the early layers in their pre-trained state. Further, the results suggest that in settings with heterogeneous constraints, choosing an architecture that maximizes the average number of trained layers across devices also maximizes the final accuracy of the FL process. Lastly, the proposed technique, CAFF, remains compatible with the quantization in the forward pass proposed in CoCoFL, as well as with orthogonal optimizations in communication (i.e., compression) and differential privacy techniques.

8 FL-Aware Hardware Design

8.1 Scope and Contribution

Previous chapters 5 to 7 inherently make the assumption that a predefined set of devices, specifically the hardware, is given, and FL has to adapt to the given hardware. For instance, in chapter 5, training is assumed to be performed on available CPUs, which are already deployed on the devices and are also used to perform non-ML-related tasks, such as user tasks or running an operating system, besides training.

Contrarily, this chapter assumes that the FL system—specifically, the hardware—can be designed prior to deployment, and that specific resources provided by the environment can be accounted for through *resource-aware FL-hardware design*. While the previous chapters examine lower-level hardware constraints—namely, FLOPs and peak memory requirements—this chapter focuses on the underlying constraint of devices: energy consumption during training. This chapter, therefore, introduces a novel, unexplored approach: the design of ML accelerators for FL. To adjust to the heterogeneous energy constraints of devices, this work exploits two mechanisms: First, an approximate MAC unit that, depending on the constraint, applies approximate computing principles [123] to the mantissa multiplication of floating-point operations, minimally affecting the learning process. Second, each accelerator, by design, uses a compressed arithmetic format that lowers the energy costs of loading and storing bytes from main memory into static random access memory (SRAM) and the processing elements (PEs), thereby significantly reducing energy consumption while only minimally impacting accuracy. We assume that devices have specific energy constraints for training, which are known at design time and can be adjusted for through specific accelerator designs. Hence, these constraints must remain static over time. Two examples of such a deployment scenario could be:

- It is known that some devices are deployed in an environment where energy is more limited than in other devices. For instance, a group of devices could be required to perform a secondary task that draws energy from a shared battery. Depending on the use case, some devices might rely on energy harvesting [124], e.g., via a solar cell, causing the energy to vary depending on the environment. Similarly, the cooling capabilities might limit the energy that can be used for training.

This chapter is mainly based on [4].

- Another source of heterogeneity could be a scenario where groups of devices have to be manufactured at different costs. For example, some devices are manufactured using the latest fabrication technology (i.e., having the most energy-efficient technology node). To cut costs, a group of devices is manufactured at a cheaper technology node that is less energy-efficient. To compensate for the inherently higher energy demand, the accelerator is designed to be approximate and use compressed memory formats, thereby reaching similar energy levels compared to the cutting-edge, precise accelerator.

While prior work [78] and the techniques presented in chapters 5 and 6 have implicitly targeted energy constraints in training by obeying computation constraints, energy is not directly targeted. Especially in [78, 79, 80], metrics like FLOPs and peak memory requirements are used as proxies for energy, which do not accurately reflect the energy cost of training, as memory accesses are not accounted for. The evaluation of these techniques in this chapter (section 8.3) shows that these assumptions lead to vastly different energy estimations compared to a more accurate energy model. Compared to the algorithm-level state of the art, this chapter adheres to energy constraints by only adjusting the hardware design, applying standard FedAvg [27] across all devices. In summary, this chapter makes the following novel contributions:

- The work presented in this chapter introduces *Approximate Computing for Federated Learning (AxFL)*, the first technique that considers hardware design for FL with heterogeneous devices.
- A detailed design of an FL accelerator is described, which uses approximate computing in its MAC units and utilizes compressed arithmetic formats to adhere to design-time energy constraints of devices.
- A fine-grained energy model is provided that takes costly memory operations during training into account.
- We show in the evaluation that, despite the approximate nature of computation and the compressed arithmetic format that allows constrained devices to lower their energy consumption by up to 4×, devices are capable of contributing to the global model, maintaining fairness among devices.

8.2 Accelerator Design

To tackle heterogeneity at design time, in this work, we design a set of ASIC-based training capable NN accelerators. Each accelerator is able to process all operations required for the training of NNs, such as convolutional operations, linear layers, normalizations, and gradient calculations. As the hardware design accounts for the devices' heterogeneous energy budgets through the use of compressed arithmetic formats and hardware approximation, we assume that each device trains the full NN architecture end-to-end.

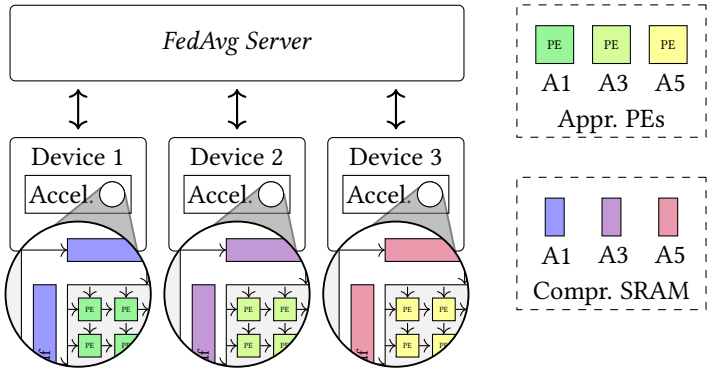


Figure 8.1: Depiction of the proposed hardware design-based FL system. Each device is equipped with an SA-based accelerator that uses approximate PE and compressed SRAM to account for energy constraints at design time.

Figure 8.3 visualizes the full accelerator design, following a scheme proposed in [101]. The accelerator is comprised of two main parts. The first part is an SA that is used to calculate general matrix multiply (GEMM) operations, such as convolutional operations or linear operations (both in the forward and backward pass). The SA consists of a grid of connected PEs, each encapsulating a MAC unit and two registers for the temporary storage and propagation of input activations and partial sums, with the latter acting as an accumulator. The SA is complemented by various SRAM buffers (namely, IBuf, WBuf, and OBuf in fig. 8.3) to store the necessary data for the SA, such as weights, inputs, and outputs.

Additionally, to handle normalization, such as BN, and other non-convolutional operations, a 1D SIMD array comprising arithmetic logic units (ALUs) is used. For the SA, SRAM buffers are used for both instructions (InMem) and data (VMem). Each SRAM buffer communicates directly with the off-chip dynamic random access memory (DRAM) via a memory controller. A schematic overview of the proposed approximate MAC unit is given in fig. 8.2.

Each participating FL device, depending on its design-time constraints, is equipped with an accelerator with a specific approximation level and a specific compressed arithmetic format. Independent of the specific local approximate accelerator, in AxFL, all devices participate in a vanilla FedAvg scheme, as visualized in fig. 8.1.

8.2.1 Hardware Approximations

In NN training, the massive number of MAC operations is a major contributor to energy consumption. Therefore, this work targets the MAC units inside the SA with approximation techniques to lower the energy consumption for resource-constrained devices. The

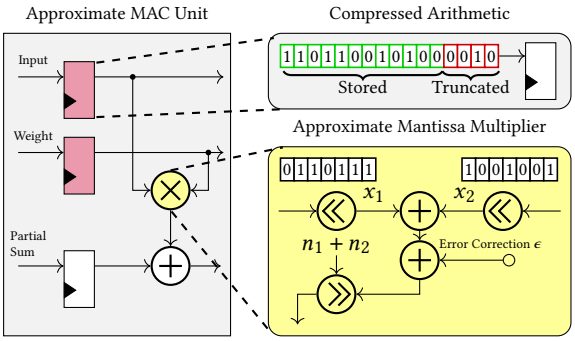


Figure 8.2: Schematic overview of the proposed approximate MAC unit. The exact multiplication operation is replaced by shift and add operations.

proposed MAC unit, depending on the device's energy constraints, can be equipped with either full-precision `float32` or approximate float arithmetic. Approximations in the MAC unit are achieved by using an approximate Minimally Biased Multiplier (MBM) [123] as the mantissa multiplier. However, other types of approximate multipliers could also be implemented [125, 126].

Approximate multiplier details: MBM [123] relies on a linear version of the logarithmic multiplication property for the approximate calculation of the mantissa. More specifically, MBM builds on top of Mitchell's linear approximation [127], which approximates the binary log and antilog mantissas by determining the leading one and treating the remaining bits as the fractional part. The two logs of the operands can then be added. The reverse process is applied to obtain the product again. Assuming two unsigned integer multiplicands (mantissas) of N bits, x_1 and x_2, the output (product) z of the MBM can be calculated using

$$z = \begin{cases} 2^{n_1+n_2}(1 + x_1 + x_2 + \epsilon) & \text{if } x_1 + x_2 < 1 \\ 2^{n_1+n_2+1}(x_1 + x_2 + \frac{\epsilon}{2}) & \text{if } x_1 + x_2 \geq 1 \end{cases}, \tag{8.1}$$

where ϵ is a fixed error correction term derived from the error magnitude curve between unsigned integers, and n_1 and n_2 represent the leading one positions. By representing the multiplication through an addition of logs and bit shift operations, energy is reduced as addition operations require less energy than multiplications. An orthogonal approximation technique that is applied is the truncation of the mantissa bits N, specifically the least significant bits (LSBs) are truncated to a desired degree. The remaining adders that are used in the PE (i.e., accumulation, float exponent addition) are designed to be exact, as the mantissa multiplication dominates the MAC's energy consumption [123]. Truncation in the accumulation results in only minor energy reduction but comes with significant degradation of accuracy.

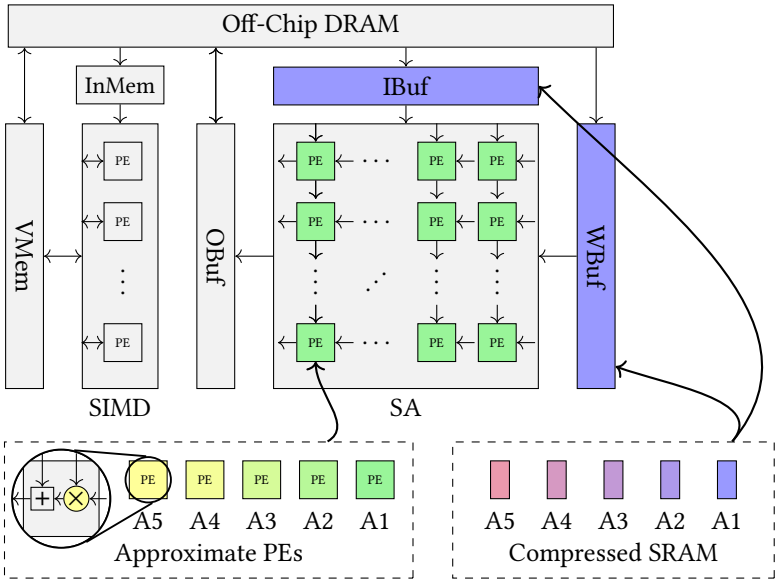


Figure 8.3: Schematic overview of the proposed training-capable FL accelerator, comprising a SA, an SIMD array, on-chip SRAM buffers, and off-chip DRAM. Approximate parts are highlighted in green tones, while compressed formats in SRAM are highlighted in purple tones. Gray components remain in `float32`.

Truncation of LSBs in memory: In case the SA is configured to truncate the LSBs of the mantissa, loading the truncated LSBs into the IBuf and WBuf's SRAM is redundant. Therefore, the SRAM also uses compressed arithmetic formats, essentially lowering the number of bits that have to be loaded from DRAM to SRAM buffers. As an example, if `bfloat16` is used as a base format (i.e., 7 mantissa bits) and 2 LSBs are truncated, only 14 bits are transferred from DRAM to on-chip buffers. To efficiently transfer these truncated formats, the SRAM's bus width is adjusted to transfer such formats in larger batches.

All other components in the accelerator are not targeted with approximation, as they only minimally contribute to the training energy.

8.2.2 Mapping of Operations

The following describes the mappings of operations used, specifically the tiling applied in the SA for convolutional operations. Linear operations can be mapped in a similar way, requiring point-wise multiplication.

Forward pass: A weight-stationary flow is applied in the SA; hence, weights are mapped unaltered to the PEs of the SA. Contrary to that, the input feature maps are tiled and

vertically broadcast in a row- or column-wise fashion to calculate the convolution operations. Partial sums are stored in the PEs' register files and propagated horizontally to their systolic neighbors, thereby forming the output feature maps in the output buffer.

Gradient with respect to the input: For the calculation of the gradient with respect to the input (e.g., applying transposed convolutions), reordering of the weights is required before they can be mapped onto the SA. Therefore, weights are rotated intra-channel-wise and transposed inter-channel-wise. Through this reordering, the convolution operation can be carried out as before, with the only difference being that the gradient from the previous layer (gradients with respect to the layer's output) acts as the streaming input to the SA.

Gradient with respect to the weights: Similar to the forward pass, feature map inputs are vertically broadcast onto the SA. The gradients with respect to the output serve as a horizontally reduced parameter, complying with the weight-stationary setup.

Remaining operations, like normalization layers (i.e., BN), are mapped to the SIMD array.

8.3 Energy Model

The following section describes the energy model used to quantify the individual accelerator components that contribute to the total energy consumption during training. In general, the energy consumption can be classified into two groups: computation-related energy consumption and memory-related energy consumption, labeled E_{comp} and E_{memory}, respectively. Further, these components can be broken down into subcomponents such that the total energy E_{total} can be described as the sum of

$$E_{\text{total}} = E_{\text{comp}} + E_{\text{memory}} = (E_{\text{SA}} + E_{\text{SIMD}}) + (E_{\text{SRAM}} + E_{\text{DRAM}}), \tag{8.2}$$

where E_{SA}, E_{SIMD}, E_{SRAM}, and E_{DRAM} represent the energy contributions of the SA, SIMD, SRAM, and DRAM, respectively. The SRAM energy consumption refers to the energy consumption of the individual on-chip SRAM buffers, where $\mathcal{B}$ represents the set of buffers (i.e., IBuf, WBuf, OBuf, InMem, and VMem). The energy consumption of each buffer can be abstracted as the product of the number of accesses a and the per-access energy cost e, such that

$$E_{\text{mem}} = a_{\text{DRAM}} \cdot e_{\text{DRAM}} + \sum_{i \in \mathcal{B}} a_i \cdot e_i. \tag{8.3}$$

The computational components' energy consumption can be described as the product of the number of operations OPs and the per-operation energy e for the PEs in the SA and SIMD, such that

$$E_{\text{comp}} = \text{OPs}_{\text{SIMD}} \cdot e_{\text{SIMD}} + \text{OPs}_{\text{SA}} \cdot e_{\text{SA}}. \tag{8.4}$$

Table 8.1: Energy measurements for each component of the proposed NN accelerator architectures. Per-operation energy consumption of MAC/ALU, and the per-bit access energy for SRAM/DRAM are reported.

component	description	energy
e_{SA}	`float32` MAC unit (exact)	26.80 pJ
$e_{\text{SA-bfloat16}}$	`bfloat16` MAC unit (exact)	5.35 pJ
$e_{\text{SA-MBM-7}}$	`bfloat16` MAC unit (MBM-7)	3.11 pJ
$e_{\text{SA-MBM-3}}$	`bfloat12` MAC unit (MBM-3)	2.78 pJ
$e_{\text{SA-MBM-1}}$	`bfloat10` MAC unit (MBM-1)	2.65 pJ
e_{SIMD}	`float32` ALU	31.40 pJ
$e_{\text{SRAM-64}}$	SRAM (64 bit bus width)	0.40 pJ
$e_{\text{SRAM-60}}$	SRAM (60 bit bus width)	0.41 pJ
e_{DRAM}	DRAM	41.00 pJ

This energy model allows for accurate estimation of the energy cost during training. Static energy during the idleness of components is omitted from the energy model, as static (leakage) power dissipated by modern MAC units accounts for less than 1% of the dynamic power [128].

8.4 Experimental Evaluation

The evaluation of the proposed technique consists of three major parts: The first part is the energy evaluation of the accelerator, specifically a simulation of its components, including approximate components, and the components' energy consumption during training. This is done by first simulating the training on the accelerator and counting all the performed operations, OPs_{SIMD} and OPs_{SA}, as well as the memory accesses, a_{DRAM} and a_{SRAM}. The second part is a circuit-level simulation of the PEs' e_{SA} and e_{SIMD}, and the SRAMs' e_{SRAM} and DRAM's e_{DRAM} to obtain per-access and per-operation energy estimations. Lastly, the FL training, including the effects of approximation on accuracy, is simulated using TensorFlow.

8.4.1 Simulation Setup

The following section describes the configuration of the accelerator as well as the specific hardware simulation setup. The proposed accelerator is equipped with a 16×16 MAC-based SA and a 1D SIMD array of 16 parallel cores. The off-chip DRAM size is set to 2 GB, and SRAM buffer sizes are configured to be 64 kB. Depending on the compressed format used, the SRAM bus width is set to either 64 bit (exact, `bfloat16`) or 60 bit (`bfloat12`, `bfloat10`). Although a 60 bit bus has a slightly higher per-access energy cost compared to 64 bit (as can be seen in table 8.1), fewer bits have to be transferred, which amortizes

the slightly higher per-access costs. As data is stored solely contiguously in memory, the per-bit energy model accurately reflects the energy consumption caused by data access. The proposed accelerator is evaluated using analytical models by Genesys/SimDIT [101]. For a specific NN architecture, input size, and minibatch size, the analytical model provides the number of memory accesses, a_{SRAM} and a_{DRAM}, and the number of operations of the SA and SIMD, namely, OPs_{SA} and OPs_{SIMD}.

The per-operation energy costs, e_{SA} and e_{SIMD}, are derived from synthesizing the respective components, i.e., the MAC unit of the SA and the ALU of the SIMD, using Synopsys Design Compiler (`compile_ultra` command) with optimized arithmetic components from the commercial Synopsys DesignWare library. Similarly, the approximation in the MAC unit is accounted for. To evaluate the per-operation energy costs, the synthesized designs are mapped to the NanGate 45 nm[1] library. The switching activity (via gate-level simulations) and average power consumption are obtained via QuestaSim and Synopsys PrimeTime, respectively.

CACTI [129] is used to obtain the per-access energy cost estimations, e_{DRAM} and e_{SRAM}, for DRAM and SRAM.

We evaluate the hardware-based approach to tackle heterogeneity in an FL setup using TensorFlow [105]. To account for the lower-precision approximate calculations, which affect the accuracy of training, we build on top of ApproxTrain [130]. To simulate the MBM approximations, custom CUDA[2] GEMM kernels are used.

8.4.2 Heterogeneity Model

To support the various energy budgets that devices have for training, a set of accelerator designs are evaluated, where each accelerator design differs only in the MAC unit's approximation technique and the level of data truncation used in memory. The different configurations are labeled A1–A5 and are presented in table 8.2. Configuration A1 (exact `float32`) is considered the energy baseline for the further evaluation, representing devices without energy constraints. The remaining configurations, A2–A5, enable energy savings through approximation. For example, A2 uses exact `bfloat16`, while A3–A5 use MBM mantissa multipliers (MBM-7 refers to 7 used mantissa bits, while MBM-3 and MBM-1 refer to 4 and 6 truncated LSBs, respectively) and compressed memory formats. Figure 8.4 provides the per-component breakdown of energy consumption in J for a single minibatch of training ResNet20 with a $3 \times 32 \times 32$ input size. It can be observed that both the SA and SRAM are responsible for the majority of the energy footprint in training, which justifies the focus of the approximation techniques on the SA and SRAM. Memory accesses, i.e., SRAM and DRAM accesses, account for 31.7% and 13.3% of the total memory consumption,

[1] `https://si2.org/open-cell-library`
[2] `https://developer.nvidia.com/cuda-toolkit`

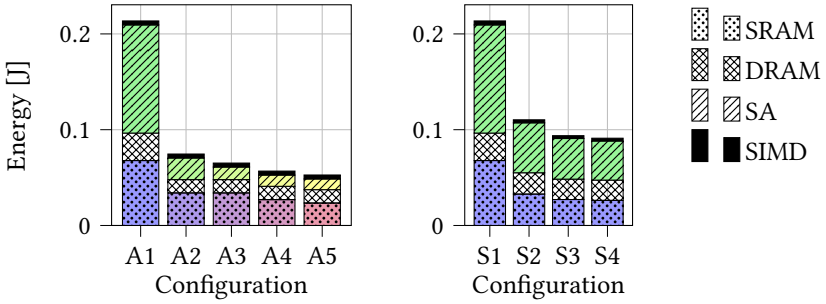


Figure 8.4: Per-component energy breakdown in joules for each accelerator configuration (A1–A5) (left) and NN subset-based techniques (S1–S4) (right) for a single minibatch of training with ResNet20 and a $3 \times 32 \times 32$ input.

respectively. The increase in the approximation level, i.e., from A1–A5, leads to a significant reduction in energy.

Device heterogeneity: We consider three equally sized groups of devices in the evaluation, labeled $\tilde{C}_1$, $\tilde{C}_2$, and $\tilde{C}_3$. Group 1, i.e., $\tilde{C}_1$, is configured to use the baseline accelerator A1, which uses full-precision arithmetic. The remaining groups, 2 and 3, use approximate accelerators A2–A5. Two different data distributions are evaluated: a typical non-iid distribution where no data correlation exists between the groups, and a non-iid distribution where devices using approximate accelerators from groups 2 and 3 have a specific data distribution that varies from group 1. For the non-iid and rc-non-iid distributions, a Dirichlet value of $\alpha = 0.1$ is used.

We compare AxFL against state-of-the-art techniques like HeteroFL [78], FedRolex [80], and FedProx [66]. Additionally, we compare against a small model baseline and the dropping of devices (see section 3.1.2). The subsets in HeteroFL, FedRolex, and the small model baseline are mapped to the same accelerator. The number of operations is evaluated on the same accelerator setup that uses exact arithmetic. We evaluate $s \in [1.0, 0.5, 0.25, 0.125]$, referred to as S1–S4. Decreasing the number of channels by s, as described in section 3.1.1, decreases the number of MAC operations quadratically by $\sim s^2PV$. However, even though MACs are theoretically lowered quadratically, we observe that when mapped to an accelerator performing parallel operations, the small subset of filters is not efficiently tiled anymore. This causes the systolic array to pad dimensions with zeros, essentially wasting energy. This is caused by the scaling of the width, but not the depth, of the NN, which results in very thin but deep NNs that allow for limited parallel execution of operations. The energy breakdown of the subset baselines is depicted in fig. 8.4 on the right. The results show that an analytic reduction of the MACs by 8× only results in an energy reduction of 2.34×. However, while some energy is wasted through additional zero padding, the usage of an accelerator still results in lower energy consumption compared to typical CPU architectures [131]. Similar to before, group 1

Table 8.2: Accelerator designs and individual components of A1-A5 and S1-S5.

	SRAM format	MAC format	Approximation	N (w.r.t. `float32`)
A1	`float32`	`float32`	-	0
A2	`bfloat16`	`bfloat16`	-	16
A3	`bfloat16`	`bfloat16`	MBM-7	16
A4	`bfloat12`	`bfloat12`	MBM-3	20
A5	`bfloat10`	`bfloat10`	MBM-1	22
S1-S4	`float32`	`float32`	-	0

($\tilde{C}_1$) uses the full model ($s = 1$), while groups 2 and 3 use downscaled models, i.e., S2–S4 ($s \in [0.5, 0.25, 0.125]$).

Lastly, we compare against FedProx [66], a state-of-the-art technique that essentially resembles dropping data within a round. Specifically, we evaluate dropping 50% and 75% of minibatches within a round, which results in energy reductions of 2× and 4×, labeled F2 and F3. For all configurations, we assume the `float32` version of the proposed accelerator is used, where devices in group 1 process 100% of their data, while groups 2 and 3 process a subset of minibatches (e.g., F2 or F3). While the energy reduction through the reduction of local minibatches is straightforward, the proximal term (eq. (3.1)) used in FedProx requires additional computations in the backward pass. This cannot be evaluated with the proposed setup. To allow for a fair comparison, the overhead of the proximal term is therefore assumed to have zero cost. For all evaluated scenarios, the hyperparameter μ (eq. (3.1)) is set to $\mu = 0.01$.

8.4.3 FL Setup

To evaluate the accuracy-energy trade-off introduced by approximate computing, several FL benchmarks are evaluated and compared with the state of the art. Specifically, we evaluate CIFAR10, CIFAR100, TinyImageNet, and FEMNIST, where data is split in equal quantities among all devices. In the case of CIFAR10 and CIFAR100, $|C^{(r)}| = 10$ with a total of $|C| = 100$ devices. For FEMNIST, the data is distributed to $|C| = 3550$ devices, while for TinyImageNet, $|C| = 200$ devices are used. For all experiments, an initial learning rate of $\eta = 0.1$, a batch size of 32, cosine decay to $\eta = 0.001$, and a total of $R = 1000$ training rounds are used. Each round, devices iterate once over their local dataset. Standard data augmentation techniques are applied, such as random cropping and horizontal and vertical flips (the last two are omitted for FEMNIST). In tables 8.3 and 8.4, the average and standard deviation from three independent seeds are presented.

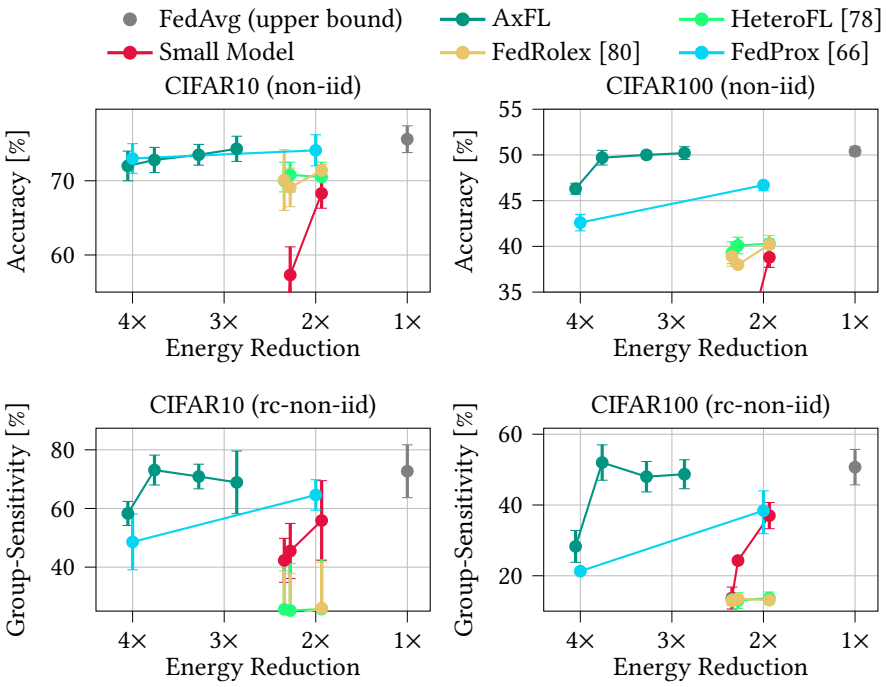


Figure 8.5: Average accuracy and group-sensitivity with standard deviation in % for three independent seeds over energy reduction for AxFL and the baselines with ResNet20 and CIFAR10. Each marker represents the configuration (A1–A5, S1–S4) used by the constrained groups.

8.4.4 Experimental Results

Mainline non-iid results: For non-iid distributions and CIFAR10, it can be observed that using approximate accelerators has only a minor effect on accuracy, as shwon in fig. 8.5. At the same time, AxFL training enables a considerable reduction in the energy consumption of constrained devices. For example, when using A3 instead of A1, devices require 2.8× less energy for training, reducing accuracy by only 2.1 p.p.. In the case of A5, which enables a reduction of 4.0×, there is a drop in accuracy of 3.6 p.p.. This represents a significantly larger energy reduction compared to techniques like HeteroFL, which enables a maximum reduction of ~ 2.3×. Additionally, HeteroFL causes a higher drop in accuracy, i.e., 5.5 p.p.. Similar behavior can be observed for FedRolex. For the small model baseline, it can be observed that downscaling for all devices significantly limits the NN's capacity, resulting in an accuracy drop of 32.2 p.p. in the case of S4.

In general, the trends for CIFAR100 are similar to CIFAR10. However, a larger drop in accuracy with HeteroFL and FedRolex can be noticed (i.e., 10.1 p.p. with S2 using HeteroFL). With AxFL, only when using A5, a larger drop in accuracy of 4.1 p.p. can be observed.

With the small model baseline, accuracy drops even further with CIFAR100, as CIFAR100 is a more complex problem than CIFAR10.

FEMNIST with ResNet8 generally requires less energy (i.e., 0.1 J compared to 0.21 J with CIFAR10 and ResNet20). However, the scaling of approximation and subsets remains in a very similar range. For full-precision training (A1/S1), an accuracy of 83.7% ± 0.5% is reached. Except for the small model baseline, only a minor drop in accuracy can be observed for both AxFL and the subset-based techniques.

Lastly, we evaluate TinyImageNet to explore the behavior of larger, more complex datasets. Importantly, TinyImageNet has a higher resolution input of $3 \times 64 \times 64$, i.e., 4× as many pixels as CIFAR10 or CIFAR100. We observe that this is reflected in the energy consumption using ResNet20, as full-precision training (A1/S1) requires 0.88 J per minibatch of training. The energy scaling changes such that A2–A5 now reach 2.9×, 3.4×, 3.9×, and 4.2×, while S2–S4 reach 2.0×, 2.4×, and 2.5×. Generally, despite the higher complexity of the problem, the same trends are observed as with CIFAR10. However, we observe that A4 reaches a higher accuracy than A3 and A2, which is most likely caused by a regularization effect from the approximate arithmetic.

Mainline rc-non-iid results: For CIFAR10 with rc-non-iid data, it can be observed that approximate accelerators enable constrained devices to make a meaningful contribution to the global model, as the group-sensitivity is only minimally affected by the approximation. The large accuracy gap between approximate accelerators and the dropping of devices highlights this effect. In A2, the group-sensitivity is only lowered by 3.8 p.p.. Counterintuitively, it can be observed that A3 improves the group-sensitivity, even when compared to A1. We think this is caused by quantization noise from the constrained devices that degrades the contribution of the full precision group, as constrained devices make up 66% of the participating devices. In this case, the group-sensitivity between the groups is traded off, as in standard non-iid accuracy, when using A3 is still degraded compared to A1. For the most aggressive approximation (A5), it can be seen that the group-sensitivity drops significantly, i.e., by 14.4 p.p. compared to A1.

The general group-sensitivity behavior among the different approximation levels and baselines is displayed for CIFAR10 in fig. 8.6. The figure suggests that higher levels of approximation can improve the group-sensitivity, albeit at the expense of the average accuracy. Consequently, the group-sensitivity of the full precision group suffers. Overall, the use of approximation for constrained devices enables high average group-sensitivity among the groups with low variance among the groups. In contrast, dropping devices causes the global model to be strongly biased towards the full precision devices and totally fails to represent the classes specific to $\tilde{C}_2$ and $\tilde{C}_3$. The small model baseline provides a fair setup for the various groups, but at a significantly lower level due to its limited model capacity.

Table 8.3: Average accuracy and standard deviation in % for AxFL and the baselines with ResNet20 using CIFAR10 and CIFAR100 with non-iid and rc-non-iid data. Each configuration (A1–A5, S1–S4) is accompanied by its corresponding energy reduction.

Setting	Technique	A1/S1 (1×)	S2 (1.94×)	S3 (2.28×)	S4 (2.34×)	A2 (2.86×)	A3 (3.28×)	A4 (3.76×)	A5 (4.05×)
non-iid CIFAR10 ResNet20	AxFL	75.6±1.8	-	-	-	74.3±1.7	73.5±1.4	72.8±1.7	72.0±2.0
	HeteroFL [78]		70.0±2.0	69.2±2.4	70.1±1.6	-	-	-	-
	FedRolex [80]		71.4±0.8	69.1±2.6	70.1±4.1	-	-	-	-
	Small Model		68.3±2.0	57.3±3.8	43.4±1.8	-	-	-	-
non-iid CIFAR100 ResNet20	AxFL	50.4±0.5	-	-	-	50.2±0.7	50.0±0.4	49.7±0.8	46.3±0.6
	HeteroFL [78]		40.3±0.5	40.1±0.9	39.3±1.2	-	-	-	-
	FedRolex [80]		40.2±1.0	38.0±0.2	38.9±1.1	-	-	-	-
	Small Model		38.8±1.1	25.7±0.8	13.7±1.7	-	-	-	-
rc-non-iid CIFAR10 ResNet20	AxFL	72.7±9.0	-	-	-	68.9±10.7	70.9± 4.2	73.1± 5.1	58.3± 4.1
	HeteroFL [78]		25.7±16.8	25.2±16.0	25.6±16.0	-	-	-	-
	FedRolex [80]		26.0±15.7	23.3±14.5	23.3±15.4	-	-	-	-
	Small Model		55.9±13.6	45.5± 9.4	42.3± 7.5	-	-	-	-
	Drop Devices		23.0±15.1	23.0±15.1	23.0±15.1	23.0±15.1	23.0±15.1	23.0±15.1	23.0±15.1
rc-non-iid CIFAR100 ResNet20	AxFL	50.7±5.0	-	-	-	48.7±4.1	48.0±4.3	52.0±5.0	28.3±4.5
	HeteroFL [78]		13.7±1.7	13.0±2.2	13.3±1.7	-	-	-	-
	FedRolex [80]		13.1±0.2	13.5±1.1	12.9±0.8	-	-	-	-
	Small Model		37.0±3.7	24.3±0.9	13.7±3.1	-	-	-	-
	Drop Devices		13.2±0.6	13.2±0.6	13.2±0.6	13.2±0.6	13.2±0.6	13.2±0.6	13.2±0.6

Table 8.4: Average accuracy and standard deviation in % for AxFL and the baselines with ResNet8 and ResNet20 using FEMNIST and TinyImageNet with non-iid and rc-non-iid data. Each configuration (A1–A5, S1–S4) is accompanied by its corresponding energy reduction.

Setting	Technique	A1/S1 (1×)	S2 (1.96×)	S3 (2.3×)	S4 (2.37×)	A2 (2.88×)	A3 (3.29×)	A4 (3.77×)	A5 (4.06×)
non-iid FEMNIST ResNet8	AxFL	83.7±0.5	-	-	-	82.2±0.7	82.1±0.7	81.4±0.6	78.7±0.2
	HeteroFL [78]		81.1±0.3	80.1±0.8	80.0±0.9	-	-	-	-
	FedRolex [80]		77.9±0.3	78.3±1.3	79.1±1.7	-	-	-	-
	Small Model		80.3±0.5	72.7±0.4	56.9±4.0	-	-	-	-
rc-non-iid FEMNIST ResNet8	AxFL	80.2±4.5	-	-	-	64.5±7.7	41.7±6.1	43.1±8.4	35.9±7.4
	HeteroFL [78]		38.9±11.0	37.6±10.8	37.4±10.1	-	-	-	-
	FedRolex [80]		40.7±10.1	35.7± 9.5	35.2± 9.5	-	-	-	-
	Small Model		80.0± 2.8	71.5± 5.7	48.2± 9.1	-	-	-	-
	Drop Devices		32.4± 8.9	32.4± 8.9	32.4± 8.9	32.4±8.9	32.4±8.9	32.4±8.9	32.4±8.9
Setting	Technique	A1/S1 (1×)	S2 (2.02×)	S3 (2.42×)	S4 (2.50×)	A2 (2.92×)	A3 (3.36×)	A4 (3.88×)	A5 (4.20×)
non-iid TinyImageNet ResNet20	AxFL	30.5±1.1	-	-	-	28.0 ±0.2	27.1±0.7	28.2±0.2	25.5±0.2
	HeteroFL [78]		25.5±0.2	23.4±0.1	23.9±0.4	-	-	-	-
	FedRolex [80]		23.3±0.8	22.5±0.5	23.5±0.7	-	-	-	-
	Small Model		23.5±0.1	17.4±1.2	11.1±0.6	-	-	-	
rc-non-iid TinyImageNet ResNet20	AxFL	26.9±6.5	-	-	-	27.6±9.1	24.0±7.7	23.5±7.5	8.4±2.4
	HeteroFL [78]		6.2±0.9	7.1±1.4	6.8±0.3	-	-	-	-
	FedRolex [80]		5.8±0.3	6.2±0.7	6.4±1.3	-	-	-	-
	Small Model		20.3±7.2	15.9±5.8	8.9±5.6	-	-	-	-
	Drop Devices		7.4±0.8	7.4±0.8	7.4±0.8	7.4±0.8	7.4±0.8	7.4±0.8	7.4±0.8

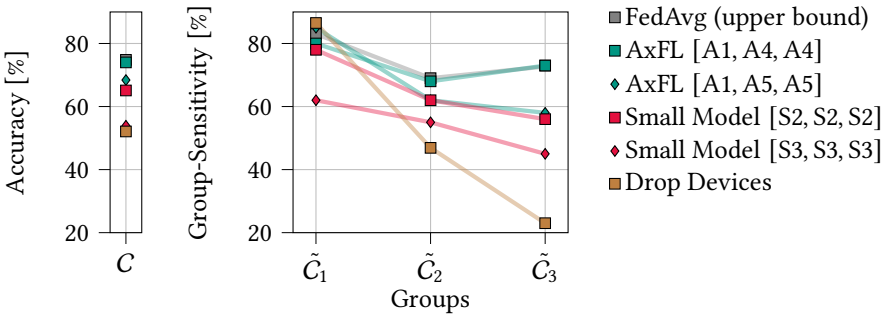


Figure 8.6: Accuracy and group-sensitivity in % of groups $\tilde{C}_1$, $\tilde{C}_2$, and $\tilde{C}_3$ with ResNet20 and CIFAR10 with rc-non-iid data.

For HeteroFL and FedRolex, it can be observed that they totally fail to present class knowledge of $\tilde{C}_2$ and $\tilde{C}_3$ in the global model. Hence, the model is completely biased towards the devices that fully train the model ($\tilde{C}_1$). As a consequence, the group-sensitivity of $\tilde{C}_3$ drops by 47.1 p.p. (HeteroFL) when using S4 instead of S1. Similar trends can be observed for CIFAR100; however, the group-sensitivity is more deteriorated by using A5. For FEMNIST, the same trend remains; however, all evaluated techniques relatively lose more accuracy compared to A1. For TinyImageNet, A5 again results in a stronger deterioration of group-sensitivity. However, apart from A5, with respect to group-sensitivity, the use of approximate accelerators provides significantly higher group-sensitivities than state-of-the-art and straightforward baselines.

Comparison against FedProx: Lastly, the proposed technique is compared against FedProx, which resembles dropping data within a FL round. For non-iid data, it can be seen that dropping data can be a competitive option to retain accuracy and reduce energy consumption, since it performs equally as our technique. Especially for CIFAR10, dropping data (F2 and F3, i.e., 2× and 4× reduction) only results in an on-average accuracy penalty of 0.6 p.p. and 2.6 p.p., respectively. However, if the per-class sample count is significantly lower, as is the case with CIFAR100, accuracy drops by 3.7 p.p. and 7.8 p.p., significantly worse than when using AxFL. The effect is even more amplified in an rc-non-iid setting when group-sensitivity is considered. When evaluating the group-sensitivity of group 3, the on-average group-sensitivity drops by 8.1 p.p. and 24.1 p.p. for F2 and F3 compared to F1 when using CIFAR10. For CIFAR100, the accuracy drops by 12.3 p.p. and 29.4 p.p., providing significantly worse results than AxFL. In summary, the comparison with FedProx shows that it is generally advantageous to train with more minibatches using approximate arithmetic than to train fewer minibatches per round in high precision.

8.5 Summary

This chapter proposed AxFL, a hardware-level design-time technique to tackle heterogeneous memory constraints of FL devices. Specifically, AxFL introduces a set of SA-based accelerators that utilize approximate computing principles in the MAC units of its PEs array and compressed arithmetic formats in the accelerator's SRAM to reduce the energy consumption in training. The simulation of the accelerator with respect to energy shows that approximate computing and compressed arithmetic formats allow a reduction of energy in training by 4×. Further, the evaluation shows that the effect of approximate arithmetic in FL training results in tolerable accuracy losses. Importantly, the proposed hardware-based approach provides a better accuracy-energy trade-off than existing algorithm-level techniques. AxFL allows for a high contribution of energy-constrained devices in the global model, significantly improving fairness aspects of FL. Moreover, the experiments suggest that it is generally advantageous to train more samples on a constrained device with approximate arithmetic than fewer samples with full precision. Lastly, AxFL is designed to be orthogonal to algorithm-level techniques such as FedProx or techniques presented in chapters 5 and 6, potentially enabling even higher energy gains while maintaining fair contribution of constrained devices.

9 Conclusion

This dissertation addresses the challenges of resource-constrained devices, such as IoT and edge devices, in the context of FL. Specifically, it focuses on heterogeneous communication, computation, memory, and energy constraints that limit these devices' ability to contribute to the training process. Chapter 2 outlines in detail how typical devices are constrained, categorizing these limitations into hard constraints, which completely prevent a device from participating, and soft constraints, which primarily slow down the FL process. Heterogeneity among devices can manifest at different scales and levels of granularity. Moreover, the specific data distribution on devices can be intertwined with their ability to participate in training. Since certain types of data may be exclusively available on constrained devices, the FL process must include them in training.

To accommodate these constraints during training, chapter 5 introduces a technique, CoCoFL, that leverages partial NN freezing and quantization. This approach allows constrained devices to participate more effectively in training compared to prior methods. The proposed technique enables independent handling of communication, computation, and memory constraints, with finer granularity and at a larger scale. While prior work theoretically supports large-scale constraint handling, its resource-reduction mechanisms tightly couple all three constraints. This coupling results in inefficiencies, as addressing one constraint limits flexibility in adapting to the others, leading to wasted resources. In contrast, CoCoFL decouples these constraints, allowing devices to utilize their limited resources more effectively. Comparisons with prior methods show that, while they can adapt to constraints at scale, they fail to enable constrained devices to meaningfully contribute to the learning process. As a result, these devices often have little impact on the global model. This limitation becomes even more evident when comparing CoCoFL to methods that fully exclude constrained devices from the FL process—an approach that prior methods fail to improve upon.

Chapter 6 focuses specifically on the peak-memory hard constraint in FL, examining how a set of memory-constrained devices can collaboratively train a large NN structure, even when no individual device is capable of training the structure end-to-end. To address this, SLT is introduced—a technique that enables a gradual increase in the model capacity of the NN. This is achieved by successively adding untrained parameters to the NN structure while freezing early layers to adhere to memory constraints. Unlike prior methods, which prune already trained parameters on a device- and round-specific basis, SLT allows parameters in the NN to better co-adapt, leading to significantly higher accuracies. This effect is highlighted in the evaluation, where prior methods perform significantly worse

than a naive baseline—specifically, a small model that adheres to memory constraints while being trained end-to-end. In contrast to prior methods, SLT converges as quickly as an end-to-end trained small model but ultimately achieves significantly higher accuracies due to the increased capacity of the larger NN.

The technique presented in chapter 7 accounts for the availability of a centralized pre-training dataset on the FL server. As a result, the early layers of the NN do not necessarily require training through the FL process, as they already provide useful upstream features from the pre-training stage. This chapter proposes a framework for selecting a specific NN topology (characterized by the number of layers) to maximize accuracy through fine-tuning while adhering to device constraints, thereby extending chapter 5. The evaluation shows that in this pre-trained scheme, heterogeneously fine-tuning the last layers enables constrained devices to meaningfully participate in the training process.

Lastly, this dissertation addresses design-time known static constraints, such as predefined energy limitations in FL. Specifically, chapter 8 proposes an energy-specific hardware design for ASIC accelerators in FL to tackle these constraints. While the basic architecture of the accelerator remains unchanged regardless of a device's constraints, energy limitations are managed by applying approximate computing techniques in floating-point multiplications and using compressed arithmetic formats in memory. Both approaches significantly reduce energy consumption during training while minimally impacting accuracy. Moreover, the evaluation shows that incorporating a share of devices utilizing approximate arithmetic in training has only a minor effect on overall accuracy. More importantly, it enables constrained devices to meaningfully participate in the training process. A comparison with existing baselines and prior methods demonstrates that approximate computing principles are well-suited for FL, as they achieve higher accuracy than approaches that rely on smaller models, drop devices, or train fewer minibatches per round.

All four presented techniques remain compatible with existing work that enhances FL, such as applying FL-specific norms [66], security enhancements like differential privacy techniques [29], up- and download compression [132], and client-selection schemes [61, 62, 65].

In conclusion, this dissertation presents four different techniques that, throughout the computation stack, address the challenges of heterogeneous devices in FL. These contributions enhance the applicability of FL in real-world use cases and increase its robustness in heterogeneous environments. The techniques presented in this dissertation enable constrained devices to participate and contribute, ensuring that knowledge from these devices is well-represented in the global FL model—an aspect where previous works have fallen short. This enables fairer FL and improves the quality of services for a broader range of devices and users.

9.1 Future Work

The contributions presented in this dissertation open up several future research directions. Firstly, the proposed techniques can, in part, be applied orthogonally. Secondly, existing techniques can be integrated into existing system-level approaches, enabling co-optimization.

Orthogonality of techniques: The techniques presented in this dissertation tackle device heterogeneity at several abstraction levels. For instance, CoCoFL and SLT (chapters 5 and 6) assume existing, already deployed devices and provide algorithm-level techniques. On the other hand, AxFL, introduced in chapter 8, focuses on designing hardware for FL. The design of AxFL is flexible enough to handle frozen layers and partially train the NN to adapt to round-based constraints. While operator fusion remains compatible with AxFL, using floating-point and integer arithmetic simultaneously, as required for quantization, is incompatible with the presented ASIC design. However, both approaches could be partially applied jointly to address a combination of static and dynamic constraints. Such a joint application would enable greater energy reductions than either technique could achieve individually. Similarly, SLT could be integrated with AxFL, as the presented ASIC accelerator relies on off-chip DRAM. The available DRAM memory may vary depending on the device's primary deployment scenario and application. Applying SLT on top of AxFL would allow training larger NNs with approximate arithmetic, even when the available DRAM is insufficient for end-to-end training. Lastly, the fusion and quantization scheme of CoCoFL could be combined with SLT so that the progressively frozen early layers are also quantized, and operator fusion is applied. This combination would improve convergence with respect to wall time on CPU-based platforms, as integer arithmetic speeds up computation compared to floating-point arithmetic.

Integration and co-optimization with system-level techniques: All techniques presented in this dissertation assume a round-based random selection of devices for participation. However, all could be coupled with existing client-selection techniques, such as [61, 62, 65]. While these system-level techniques cannot address hard constraints, such as memory limitations, they can reduce the convergence time of FL by using variable-length rounds, grouping devices, and tiering devices. Co-optimizing the presented techniques with these system-level techniques could further address the heterogeneity challenges in FL. Lastly, future research could incorporate data-centric approaches that specifically optimize which kind of data, available locally on devices, should be used for training in a given FL round.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ALU arithmetic logic unit 87, 92

ASIC application-specific integrated circuit . . . . . ii, 3, 6, 16, 27, 86, 102, 103

AxFL Approximate Computing for Federated Learning 86, 87, 93–96, 99, 103

BN Batch Normalization 9, 23, 31, 35, 37–39, 46, 87, 90

CAFF Constraint-Aware Federated Finetuning 73, 76, 78, 83, 109

CPU central processing unit 3, 16, 17, 27, 51, 85, 93, 103

DRAM dynamic random access memory 87, 88, 90–92, 103

FD Federated Dropout 22, 23, 53–56, 62, 63, 66

FedAvg federated averaging 10, 11, 22, 23, 87

FL federated learning i, ii, 3–7, 10–19, 21–27, 29–31, 33, 34, 36–40, 42, 44–46, 48–51, 53, 54, 56–62, 64, 66, 68–74, 76–78, 80, 82, 83, 85–88, 90–92, 94, 96, 98–103

FLOP floating-point operation 15, 27, 29, 33, 56, 71, 73, 77, 85, 86

FLOPS floating-point operations per second 15, 16, 19, 34

GEMM general matrix multiply 86, 92

GPU graphics processing unit 1–3, 16, 27, 29

iid independent and identically distributed 6, 12, 13, 45, 48, 56, 63, 64

IoT internet of things . . . . . i, 2, 3, 101

LLM Large Language Model 24, 71

LoRA Low-Rank Adaptation 24, 71, 72, 78

LSB least significant bit 88, 92

MAC multiply accumulate 6, 27, 36, 85–88, 90–93, 99

MBM Minimally Biased Multiplier 87, 88, 92

ML machine learning . . . . . i, 1–3, 6, 7, 16, 26, 85

NLP natural language processing 31, 48

NN neural network i, 1–10, 14–17, 22–27, 29, 31–37, 39–44, 46, 48, 51, 53, 54, 56–59, 62, 63, 66, 67, 69, 71–80, 82, 83, 86, 87, 91, 93, 94, 101–103

non-iid non-independent and identically distributed i, 4–6, 12, 13, 21, 45, 56, 64, 76, 78, 93, 94, 96, 99

PE processing element 85, 86, 88–91, 99

RAM random access memory 3, 15–17, 19, 42

rc-non-iid resource-correlated non-independent and identically distributed 5, 6, 12, 13, 15, 25, 33, 45, 46, 48, 73, 76, 78, 93, 96, 99

SA systolic array 6, 27, 86–92, 99

SGD stochastic gradient descent 11, 45, 61

SIMD single instruction, multiple data 27, 87, 90–92

SLT Successive Layer Training 56, 57, 60–64, 66, 67, 69, 101, 103

SoC System on a Chip 51

SRAM static random access memory 85–88, 90–92, 99

SVD singular value decomposition 24

Bibliography

[1] Kilian Pfeiffer, Martin Rapp, Ramin Khalili, and Jörg Henkel. "CoCoFL: Communication- and Computation-Aware Federated Learning via Partial NN Freezing and Quantization". In: *Transactions on Machine Learning Research (TMLR)* (2023). ISSN: 2835-8856. URL: https://openreview.net/forum?id=XJIg4kQbkv.

[2] Kilian Pfeiffer, Ramin Khalili, and Jörg Henkel. "Aggregating Capacity in FL through Successive Layer Training for Computationally-Constrained Devices". In: *Advances in Neural Information Processing Systems (NeurIPS)* 36 (2024).

[3] Kilian Pfeiffer, Martin Rapp, Ramin Khalili, and Jörg Henkel. "Federated Learning for Computationally Constrained Heterogeneous Devices: A Survey". In: *ACM Computing Surveys (ACM CSUR)* 55.14s (2023), pp. 1–27.

[4] Kilian Pfeiffer, Konstantinos Balaskas, Kostas Siozios, and Jörg Henkel. "Energy-Aware Heterogeneous Federated Learning via Approximate Systolic DNN Accelerators". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2024).

[5] Kilian Pfeiffer, Mohamed Aboelenien Ahmed, Ramin Khalili, and Jörg Henkel. Efficient Federated Finetuning of Tiny Transformers with Resource-Constrained Devices. 2024. arXiv: 2411.07826 [cs.LG].

[6] Martin Rapp, Ramin Khalili, Kilian Pfeiffer, and Jörg Henkel. "Distreal: Distributed Resource-Aware Learning in Heterogeneous Systems". In: *AAAI Conference on Artificial Intelligence*. Vol. 36. 7. 2022, pp. 8062–8071.

[7] Mohamed Aboelenien Ahmed, Kilian Pfeiffer, Ramin Khalili, Heba Khdr, and Jörg Henkel. Efficient Zero-Order Federated Finetuning of Language Models for Resource-Constrained Devices. 2025. arXiv: 2502.10239 [cs.LG].

[8] Mohamed Aboelenien Ahmed, Kilian Pfeiffer, Heba Khdr, Osama Abboud, Ramin Khalili, and Jörg Henkel. Accelerated Training on Low-Power Edge Devices. 2025. arXiv: 2502.18323 [cs.LG].

[9] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 25. 2012, pp. 1106–1114.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A Large-Scale Hierarchical Image Database". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2009, pp. 248–255.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 30. 2017, pp. 5998–6008.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding. 2018. arXiv: 1810.04805 [cs.CL].

[14] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. LLaMa: Open and Efficient Foundation Language Models. 2023. arXiv: 2302.13971 [cs.CL].

[15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going Deeper With Convolutions". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9.

[16] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. "Shufflenet V2: Practical Guidelines for Efficient CNN Architecture Design". In: *European conference on computer vision (ECCV)*. 2018, pp. 116–131.

[17] Terrence J Sejnowski and Charles R Rosenberg. "Parallel Networks that Learn to Pronounce English Text". In: *Complex Systems* 1.1 (1987), pp. 145–168.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. "Language Models are Unsupervised Multitask Learners". In: *OpenAI blog* (2019), p. 9.

[20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 33. 2020, pp. 1877–1901.

[21] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, et al. DeepSeek-V3 Technical Report. 2025. arXiv: 2412.19437 [cs.CL].

[22] James Manyika, Michael Chui, Peter Bisson, Jonathan Woetzel, Richard Dobbs, Jacques Bughin, and Dan Aharon. "Unlocking the Potential of the Internet of Things". In: *McKinsey Global Institute* (2015).

[23] Farzad Samie, Lars Bauer, and Jörg Henkel. "IoT technologies for embedded computing: A survey". In: *IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*. 2016, pp. 1–10.

[24] EU. European Union's General Data Protection Regulation (GDPR). Retrieved 2020-13-04. 2020. URL: https://gdpr.eu/.

[25] California. California Consumer Privacy Act. Retrieved 2020-13-04. 2020. URL: https://oag.ca.gov/privacy/ccpa/regs.

[26] Farzad Samie, Lars Bauer, and Jörg Henkel. "From cloud down to things: An overview of machine learning in internet of things". In: *IEEE Internet of Things Journal* 6.3 (2019), pp. 4921–4934.

[27] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. "Communication-efficient learning of deep networks from decentralized data". In: *Artificial Intelligence and Statistics*. Vol. 20. PMLR. 2017, pp. 1273–1282.

[28] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical Secure Aggregation for Federated Learning on User-Held Data. 2016. arXiv: 1611.04482 [cs.CR].

[29] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H Yang, Farhad Farokhi, Shi Jin, Tony QS Quek, and H Vincent Poor. "Federated Learning with Differential Privacy: Algorithms and Performance Analysis". In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 3454–3469.

[30] Binhang Yuan, Song Ge, and Wenhui Xing. A Federated Learning Framework for Healthcare IoT Devices. 2020. arXiv: 2005.05083 [cs.LG].

[31] Ce Ju, Dashan Gao, Ravikiran Mane, Ben Tan, Yang Liu, and Cuntai Guan. "Federated Transfer Learning for EEG Signal Classification". In: *Engineering in Medicine & Biology Society*. IEEE, 2020, pp. 3040–3045.

[32] Dashan Gao, Ce Ju, Xiguang Wei, Yang Liu, Tianjian Chen, and Qiang Yang. Hhhfl: Hierarchical Heterogeneous Horizontal Federated Learning for Electroencephalography. 2019. arXiv: 1909.05784.

[33] Yiqiang Chen, Xin Qin, Jindong Wang, Chaohui Yu, and Wen Gao. "Fedhealth: A Federated Transfer Learning Framework for Wearable Healthcare". In: *IEEE Intelligent Systems* 35.4 (2020), pp. 83–93.

[34] Xinle Liang, Yang Liu, Tianjian Chen, Ming Liu, and Qiang Yang. Federated Transfer Reinforcement Learning for Autonomous Driving. 2019. arXiv: 1910.06001 [cs.LG].

[35] Yuris Mulya Saputra, Dinh Thai Hoang, Diep N Nguyen, Eryk Dutkiewicz, Markus Dominik Mueck, and Srikathyayani Srikanteswara. "Energy Demand Prediction with Federated Learning for Electric Vehicle Networks". In: *2019 IEEE Global Communications Conference*. IEEE, 2019, pp. 1–6.

[36] Bekir Sait Ciftler, Abdullatif Albaseer, Noureddine Lasla, and Mohamed Abdallah. Federated Learning for Localization: A Privacy-Preserving Crowdsourcing Method. 2020. arXiv: 2001.01911 [cs.NI].

[37] Jason Posner, Lewis Tseng, Moayad Aloqaily, and Yaser Jararweh. "Federated Learning in Vehicular Networks: Opportunities and Solutions". In: *IEEE Network* 35 (2021), pp. 1–12.

[38] Boyi Liu, Lujia Wang, and Ming Liu. "Lifelong Federated Reinforcement Learning: A Learning Architecture for Navigation in Cloud Robotic Systems". In: *IEEE Robotics and Automation Letters* 4.4 (2019), pp. 4555–4562.

[39] Boyi Liu, Lujia Wang, Ming Liu, and Cheng-Zhong Xu. Federated Imitation Learning: A Privacy Considered Imitation Learning Framework for Cloud Robotic Systems with Heterogeneous Sensor Data. 2019. arXiv: 1909.00895 [cs.RO].

[40] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied Federated Learning: Improving Google Keyboard Query Suggestions. 2018. arXiv: 1812.02903 [cs.LG].

[41] Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, et al. Federated Evaluation and Tuning for On-Device Personalization: System Design & Applications. 2021. arXiv: 2102.08503 [cs.LG].

[42] Kiwan Maeng, Haiyu Lu, Luca Melis, John Nguyen, Mike Rabbat, and Carole-Jean Wu. "Towards Fair Federated Recommendation Learning: Characterizing the Inter-Dependence of System and Data Heterogeneity". In: *16th ACM Conference on Recommender Systems*. 2022, pp. 156–167.

[43] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[44] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *International Conference on Machine Learning (ICML)*. PMLR. 2015, pp. 448–456.

[45] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization. 2016. arXiv: 1607.06450 [stat.ML].

[46] Daliang Li and Junpu Wang. "FedMD: Heterogenous Federated Learning via Model Distillation". In: *NeurIPS Workshop on Federated Learning for Data Privacy and Confidentiality*. Vol. 33. 2019, pp. 1–4.

[47] Yujing Chen, Yue Ning, Martin Slawski, and Huzefa Rangwala. "Asynchronous Online Federated Learning for Edge Devices with Non-Iid Data". In: *IEEE International Conference on Big Data*. IEEE. 2020, pp. 15–24.

[48] Michael R Sprague, Amir Jalalirad, Marco Scavuzzo, Catalin Capota, Moritz Neun, Lyman Do, and Michael Kopp. "Asynchronous Federated Learning for Geospatial Applications". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*. Springer. 2018, pp. 21–28.

[49] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Asynchronous Federated Optimization. 2019. arXiv: 1903.03934 [cs.DC].

[50] Yuxin Shi, Han Yu, and Cyril Leung. "Towards Fairness-Aware Federated Learning". In: *IEEE Transactions on Neural Networks and Learning Systems* 35.9 (2024).

[51] Ozan Erdinc, Bulent Vural, and Mehmet Uzunoglu. "A Dynamic Lithium-Ion Battery Model Considering the Effects of Temperature and Capacity Fading". In: *International Conference on Clean Electrical Power*. IEEE, 2009, pp. 383–386.

[52] Matthieu Dubarry, Vojtech Svoboda, Ruey Hwu, and Bor Yann Liaw. "Capacity and Power Fading Mechanism Identification from a Commercial Cell Evaluation". In: *Journal of Power Sources* 165.2 (2007), pp. 566–572.

[53] Jörg Henkel, Heba Khdr, and Martin Rapp. "Smart Thermal Management for Heterogeneous Multicores". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 132–137.

[54] Biyi Fang, Xiao Zeng, and Mi Zhang. "NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision". In: *Annual International Conference on Mobile Computing and Networking*. Vol. 24. MobiCom '18. Association for Computing Machinery, 2018, pp. 115–127.

[55] Mingxing Tan and Quoc Le. "Efficientnet: Rethinking Model Scaling for Convolutional Neural Networks". In: *International Conference on Machine Learning (ICML)*. PMLR. 2019, pp. 6105–6114.

[56] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "Mobilenetv2: Inverted Residuals and Linear Bottlenecks". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 4510–4520.

[57] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. "Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization". In: *Machine Learning and Systems (MLSys)*. Vol. 2. 2020, pp. 497–511.

[58] Jörg Henkel, Heba Khdr, Santiago Pagani, and Muhammad Shafique. "New Trends in Dark Silicon". In: *Design Automation Conference (DAC)*. Vol. 52. IEEE. Association for Computing Machinery, 2015, pp. 1–6.

[59] Yuanming Shi, Kai Yang, Tao Jiang, Jun Zhang, and Khaled B Letaief. "Communication-Efficient Edge AI: Algorithms and Systems". In: *IEEE Communications Surveys & Tutorials* 22.4 (2020), pp. 2167–2191.

[60] Urmish Thakker, Jesse Beu, Dibakar Gope, Chu Zhou, Igor Fedorov, Ganesh Dasika, and Matthew Mattina. Compressing RNNs for Iot Devices by 15-38x Using Kronecker Products. 2019. arXiv: 1906.02876 [cs.LG].

[61] Takayuki Nishio and Ryo Yonetani. "Client Selection for Federated Learning with Heterogeneous Resources in Mobile Edge". In: *IEEE international conference on communications*. IEEE. 2019, pp. 1–7.

[62] Zheng Chai, Ahsan Ali, Syed Zawad, Stacey Truex, Ali Anwar, Nathalie Baracaldo, Yi Zhou, Heiko Ludwig, Feng Yan, and Yue Cheng. "Tifl: A Tier-Based Federated Learning System". In: *International Symposium on High-Performance Parallel and Distributed Computing*. Vol. 29. 2020, pp. 125–136.

[63] Amirhossein Reisizadeh, Isidoros Tziotis, Hamed Hassani, Aryan Mokhtari, and Ramtin Pedarsani. "Straggler-Resilient Federated Learning: Leveraging the Interplay Between Statistical Accuracy and System Heterogeneity". In: *IEEE Journal on Selected Areas in Information Theory* 3.2 (2022), pp. 197–205.

[64] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. "Oort: Efficient Federated Learning via Guided Participant Selection". In: *Symposium on Operating Systems Design and Implementation*. Vol. 15. 2021, pp. 19–35.

[65] Chenning Li, Xiao Zeng, Mi Zhang, and Zhichao Cao. "PyramidFL: A Fine-Grained Client Selection Framework for Efficient Federated Learning". In: *International Conference on Mobile Computing And Networking*. Vol. 28. 2022, pp. 158–171.

[66] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. "Federated Optimization in Heterogeneous Networks". In: *Machine Learning and Systems (MLSys)*. Vol. 2. 2020, pp. 429–450.

[67] Zhaohui Yang, Mingzhe Chen, Walid Saad, Choong Seon Hong, and Mohammad Shikh-Bahaei. "Energy Efficient Federated Learning over Wireless Communication Networks". In: *IEEE Transaction on Wireless Communications* 20.3 (2020), pp. 1935–1949.

[68] Dzmitry Huba, John Nguyen, Kshitiz Malik, Ruiyu Zhu, Mike Rabbat, Ashkan Yousefpour, Carole-Jean Wu, Hongyuan Zhan, Pavel Ustinov, Harish Srinivas, et al. "Papaya: Practical, Private, and Scalable Federated Learning". In: *Machine Learning and Systems (MLSys)*. Vol. 4. 2022, pp. 814–832.

[69] Zheng Chai, Yujing Chen, Ali Anwar, Liang Zhao, Yue Cheng, and Huzefa Rangwala. "FedAT: A High-Performance and Communication-Efficient Federated Learning System with Asynchronous Tiers". In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–16.

[70] Zirui Xu, Fuxun Yu, Jinjun Xiong, and Xiang Chen. "Helios: Heterogeneity-Aware Federated Learning with Dynamically Balanced Collaboration". In: *Design Automation Conference (DAC)*. Vol. 58. IEEE. 2021, pp. 997–1002.

[71] Hongyan Chang, Virat Shejwalkar, Reza Shokri, and Amir Houmansadr. Cronus: Robust and Heterogeneous Collaborative Learning with Black-Box Knowledge Transfer. 2019. arXiv: `1912.11279 [stat.ML]`.

[72] Yun Hin Chan and Edith CH Ngai. "Fedhe: Heterogeneous models and communication-efficient federated learning". In: *International Conference on Mobility, Sensing and Networking*. Vol. 17. IEEE. 2021, pp. 207–214.

[73] Yue Tan, Guodong Long, Lu Liu, Tianyi Zhou, Qinghua Lu, Jing Jiang, and Chengqi Zhang. "Fedproto: Federated prototype learning across heterogeneous clients". In: *AAAI Conference on Artificial Intelligence*. Vol. 36. 8. 2022, pp. 8432–8440.

[74] Tao Lin, Lingjing Kong, Sebastian U Stich, and Martin Jaggi. "Ensemble Distillation for Robust Model Fusion in Federated Learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 33. 2020, pp. 2351–2363.

[75] Tao Shen, Jie Zhang, Xinkang Jia, Fengda Zhang, Gang Huang, Pan Zhou, Kun Kuang, Fei Wu, and Chao Wu. Federated Mutual Learning. 2020. arXiv: 2006.16765 [cs.LG].

[76] Geoffrey Hinton. Distilling the Knowledge in a Neural Network. 2015. arXiv: 1503.02531 [stat.ML].

[77] Sebastian Caldas, Jakub Konečny, H Brendan McMahan, and Ameet Talwalkar. Expanding the Reach of Federated Learning by Reducing Client Resource Requirements. 2018. arXiv: 1812.07210 [cs.LG].

[78] Enmao Diao, Jie Ding, and Vahid Tarokh. "HeteroFL: Computation and communication efficient federated learning for heterogeneous clients". In: *International Conference on Learning Representations (ICLR)*. 2020.

[79] Samuel Horvath, Stefanos Laskaridis, Mario Almeida, Ilias Leontiadis, Stylianos Venieris, and Nicholas Lane. "FjORD: Fair and Accurate Federated Learning under Heterogeneous Targets with Ordered Dropout". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 34. 2021, pp. 12876–12889.

[80] Samiul Alam, Luyang Liu, Ming Yan, and Mi Zhang. "FedRolex: Model-Heterogeneous Federated Learning with Rolling Sub-Model Extraction". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 35. 2022, pp. 29677–29690.

[81] Minjae Kim, Sangyoon Yu, Suhyun Kim, and Soo-Mook Moon. "DepthFL : Depthwise Federated Learning for Heterogeneous Clients". In: *International Conference on Learning Representations (ICLR)*. Vol. 11. 2023.

[82] Dezhong Yao, Wanning Pan, Yao Wan, Hai Jin, and Lichao Sun. FedHM: Efficient Federated Learning for Heterogeneous Models via Low-rank Factorization. 2021. arXiv: 2111.14655 [cs.LG].

[83] Sara Babakniya, Ahmed Elkordy, Yahya Ezzeldin, Qingfeng Liu, Kee-Bong Song, Mostafa El-Khamy, and Salman Avestimehr. "SLoRA: Federated Parameter Efficient Fine-Tuning of Language Models". In: *International Workshop on Federated Learning in the Age of Foundation Models in Conjunction with NeurIPS*. 2023.

[84] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. 2021. arXiv: 2106.09685 [cs.CL].

[85] Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. "Training Quantized Nets: A Deeper Understanding". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 30. 2017, pp. 5813–5823.

[86] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. "Deep Learning with Limited Numerical Precision". In: *International Conference on Machine Learning (ICML)*. PMLR. 2015, pp. 1737–1746.

[87] Kelam Goutam, S Balasubramanian, Darshan Gera, and R Raghunatha Sarma. "Layerout: Freezing Layers in Deep Neural Networks". In: *SN Computer Science* 1.5 (2020), p. 295.

[88] Chen Chen, Hong Xu, Wei Wang, Baochun Li, Bo Li, Li Chen, and Gong Zhang. "Communication-efficient federated learning with adaptive parameter freezing". In: *International Conference on Distributed Computing Systems*. Vol. 41. IEEE. 2021, pp. 1–11.

[89] Tien-Ju Yang, Dhruv Guliani, Françoise Beaufays, and Giovanni Motta. "Partial Variable Training for Efficient On-Device Federated Learning". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2022, pp. 4348–4352.

[90] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. "Dynamic Tensor Rematerialization". In: *International Conference on Learning Representations (ICLR)*. Vol. 9. 2021.

[91] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. "Gist: Efficient data encoding for deep neural network training". In: *ACM/IEEE Annual International Symposium on Computer Architecture*. Vol. 45. IEEE. 2018, pp. 776–789.

[92] Mikail Yayla and Jian-Jia Chen. "Memory-efficient training of binarized neural networks on the edge". In: *Design Automation Conference (DAC)*. Vol. 59. ACM. 2022, pp. 661–666.

[93] Georgios Georgiadis. "Accelerating Convolutional Neural Networks Via Activation Map Compression". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 7085–7095.

[94] Chris Hettinger, Tanner Christensen, Ben Ehlert, Jeffrey Humpherys, Tyler Jarvis, and Sean Wade. Forward Thinking: Building and Training Neural Networks One Layer At A Time. 2017. arXiv: `1706.02480 [stat.ML]`.

[95] Sindy Löwe, Peter O'Connor, and Bastiaan Veeling. "Putting An End To End-To-End: Gradient-Isolated Learning of Representations". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 32. 2019, pp. 3033–3045.

[96] Yuwen Xiong, Mengye Ren, and Raquel Urtasun. "LoCo: Local Contrastive Representation Learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 33. 2020, pp. 11142–11153.

[97] Hui-Po Wang, Sebastian Stich, Yang He, and Mario Fritz. "ProgFed: Effective, Communication, and Computation Efficient Federated Learning by Progressive Training". In: *International Conference on Machine Learning (ICML)*. PMLR. 2022, pp. 23034–23054.

[98] Amit Kumar Kundu and Joseph Jaja. "Fednet2net: Saving Communication and Computations in Federated Learning with ;Model Growing". In: *International Conference on Artificial Neural Networks*. Springer. 2022, pp. 236–247.

[99] Nguyen H Tran, Wei Bao, Albert Zomaya, Minh NH Nguyen, and Choong Seon Hong. "Federated Learning over Wireless Networks: Optimization Model Design and Analysis". In: *IEEE Conference on Computer Communications*. IEEE. 2019, pp. 1387–1395.

[100] Sai Qian Zhang, Thierry Tambe, Nestor Cuevas, Gu-Yeon Wei, and David Brooks. "CAMEL: Co-Designing AI Models and eDRAMs for Efficient On-Device Learning". In: *IEEE International Symposium on High-Performance Computer Architecture*. IEEE. 2024, pp. 861–875.

[101] Hadi Esmaeilzadeh, Soroush Ghodrati, Andrew B Kahng, Sean Kinzer, Susmita Dey Manasi, Sachin S Sapatnekar, and Zhiang Wang. Performance Analysis of DNN Inference/Training with Convolution and non-Convolution Operations. 2023. arXiv: 2306.16767 [cs.AR].

[102] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. "On-Device Training Under 256kb Memory". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 35. 2022, pp. 22941–22954.

[103] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. "TPU V4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings". In: *Annual International Symposium on Computer Architecture*. Vol. 50. 2023, pp. 1–14.

[104] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 32. 2019.

[105] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2016. arXiv: 1603.04467 [cs.DC].

[106] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. 2009.

[107] Luke N Darlow, Elliot J Crowley, Antreas Antoniou, and Amos J Storkey. Cinic-10 is Not Imagenet or Cifar-10. 2018. arXiv: 1810.03505 [cs.CV].

[108] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. Leaf: A Benchmark for Federated Settings. 2018. arXiv: 1812.01097 [cs.LG].

[109] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M. Summers. "ChestX-ray8: Hospital-Scale Chest X-Ray Database and Benchmarks on Weakly-Supervised Classification and Localization of Common Thorax Diseases". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[110] Ya Le and Xuan S. Yang. Tiny ImageNet Visual Recognition Challenge. 2015.

[111] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. "Learning Word Vectors for Sentiment Analysis". In: *Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Vol. 49. June 2011, pp. 142–150.

[112] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. 2018. arXiv: 1808.06226 [cs.CL].

[113] Aaron Gokaslan and Vanya Cohen. OpenWebText Corpus. http://Skylion007.github.io/OpenWebTextCorpus. 2019.

[114] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. "Densely Connected Convolutional Networks". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 2261–2269.

[115] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *International Conference on Learning Representations (ICLR)*. Vol. 9. 2020.

[116] Andrea Goldsmith. *Wireless Communications*. Cambridge University Press, 2005. DOI: 10.1017/CBO9780511841224.

[117] Yuxin Wu and Kaiming He. "Group Normalization". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Sept. 2018.

[118] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *The Journal of Machine Learning Research (JMLR)* 15.1 (2014), pp. 1929–1958.

[119] Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. Measuring the Effects of Non-Identical Data Distribution for Federated Visual Classification. 2019. arXiv: 1909.06335 [cs.LG].

[120] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. "Parameter-Efficient Transfer Learning for NLP". In: *International Conference on Machine Learning (ICML)*. Vol. 36. PMLR. 2019, pp. 2790–2799.

[121] Ilya Loshchilov and Frank Hutter. "Decoupled Weight Decay Regularization". In: *International Conference on Learning Representations (ICLR)*. Vol. 7. 2019.

[122] Yae Jee Cho, Luyang Liu, Zheng Xu, Aldi Fahrezi, Matt Barnes, and Gauri Joshi. "Heterogeneous LoRA for Federated Fine-tuning of On-device Foundation Models". In: *International Workshop on Federated Learning in the Age of Foundation Models in Conjunction with NeurIPS*. 2023.

[123] Hassaan Saadat, Haseeb Bokhari, and Sri Parameswaran. “Minimally Biased Multipliers for Approximate Integer and Floating-Point Multiplication”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37.11 (2018), pp. 2623–2635.

[124] Neha Garg and Ritu Garg. “Energy Harvesting in IoT Devices: A Survey”. In: *International Conference on Intelligent Sustainable Systems*. Vol. 236. IEEE. 2017, pp. 127–131.

[125] Jie Li, Yi Guo, and Shinji Kimura. “Accuracy-Configurable Low-Power Approximate Floating-Point Multiplier Based on Mantissa Bit Segmentation”. In: *IEEE Region 10 Conference*. IEEE. 2020, pp. 1311–1316.

[126] Chuangtao Chen, Weikang Qian, Mohsen Imani, Xunzhao Yin, and Cheng Zhuo. “PAM: A Piecewise-Linearly-Approximated Floating-Point Multiplier with Unbiasedness and Configurability”. In: *IEEE Transactions on Computers (TC)* 71.10 (2021), pp. 2473–2486.

[127] John N Mitchell. “Computer Multiplication and Division Using Binary Logarithms”. In: *IRE Transactions on Electronic Computers* EC-11.4 (1962), pp. 512–517.

[128] Harsh Chhajed, Gopal Raut, Narendra Dhakad, Sudheer Vishwakarma, and Santosh Kumar Vishvakarma. “BitMAC: Bit-Serial Computation-Based Efficient Multiply-Accumulate Unit for DNN Accelerator”. In: *Circuits, Systems, and Signal Processing* (2022), pp. 1–16.

[129] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. “Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0”. In: *IEEE/ACM International Symposium on Microarchitecture*. Vol. 40. IEEE. 2007, pp. 3–14.

[130] Jing Gong, Hassaan Saadat, Hasindu Gamaarachchi, Haris Javaid, Xiaobo Sharon Hu, and Sri Parameswaran. “ApproxTrain: Fast Simulation of Approximate Multipliers for DNN Training and Inference”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 42.11 (2023), pp. 3505–3518.

[131] Yunxiang Hu, Yuhao Liu, and Zhuovuan Liu. “A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC”. In: *International Conference on Computer Research and Development*. Vol. 14. IEEE. 2022, pp. 100–107.

[132] Farzin Haddadpour, Mohammad Mahdi Kamani, Aryan Mokhtari, and Mehrdad Mahdavi. “Federated Learning with Compression: Unified Analysis and Sharp Guarantees”. In: *International Conference on Artificial Intelligence and Statistics*. Vol. 24. PMLR. 2021, pp. 2350–2358.