Universidade do Minho
Escola de Engenharia

Diogo André Veiga Costa

A Federated Learning Framework for the Next-Generation Machine Learning Systems

Março de 2022

Universidade do Minho
Escola de Engenharia

Diogo André Veiga Costa

A Federated Learning Framework for the Next-Generation Machine Learning Systems

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Sandro Emanuel Salgado Pinto

Março de 2022

Agradecimentos

Em primeiro lugar, gostaria de agradecer às pessoas que sempre me apoiaram e tornaram possível tudo o que sou hoje, os meus pais e irmão, que sempre foram modelos exemplares em todo o meu percurso. Agradeço por me permitirem voar e ir cada vez mais longe, mas mais do que isso, por me guiar nesse trajeto.

Aos meus orientadores, Professor Doutor João Monteiro, Doutor Sandro Pinto e Mestre Miguel Costa, por todo o conhecimento transmitido, por toda a disponibilidade, e sobretudo pela confiança depositada em mim para a realização deste trabalho. Um sincero obrigado por todas as oportunidades e todas as lições.

Aos meus amigos de curso, que foram um grande apoio em todo o meu percurso académico. Um agradecimento especial para as pessoas com quem tenho o prazer de conviver e partilhar experiências diariamente – André Campos, João Sousa, Luís Cunha, Pedro Sousa e Samuel Pereira. Um obrigado a todos que fizeram parte deste meu percurso de desenvolvimento, e que me deram a oportunidade de crescer e aprender.

Aos meus amigos ao grupo “Boticas” que proporcionou estes cinco anos inesquecíveis. Um agradecimento particular aos meus parceiros João Rego, Paulo Silva e Pedro Araújo por toda a ajuda e momentos de amizade ao longo destes anos.

Por fim, ao meu pilar, Débora Pereira, por todo apoio incondicional, motivação e paciência. Um obrigado pela companhia nesta longa jornada, por me apoiar nos melhores e nos piores momentos. Todas as críticas construtivas, todos as mensagens de apoio, e todo o carinho tornaram este percurso possível.

À restante família e todas as outras pessoas que me apoiaram durante todos estes anos: um sincero obrigado!

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Abstract

The end of Moore's Law aligned with rising concerns about data privacy is forcing machine learning (ML) to shift from the cloud to the deep edge, near to the data source. In the next-generation ML systems, the inference and part of the training process will be performed right on the edge, while the cloud will be responsible for major ML model updates. This new computing paradigm, referred to by academia and industry researchers as federated learning, alleviates the cloud and network infrastructure while increasing data privacy. Recent advances have made it possible to efficiently execute the inference pass of quantized artificial neural networks on Arm Cortex-M and RISC-V (RV32IMCXpulp) microcontroller units (MCUs). Nevertheless, the training is still confined to the cloud, imposing the transaction of high volumes of private data over a network.

To tackle this issue, this MSc thesis makes the first attempt to run a decentralized training in Arm Cortex-M MCUs. To port part of the training process to the deep edge is proposed L-SGD, a lightweight version of the stochastic gradient descent optimized for maximum speed and minimal memory footprint on Arm Cortex-M MCUs. The L-SGD is 16.35x faster than the TensorFlow solution while registering a memory footprint reduction of 13.72%. This comes at the cost of a negligible accuracy drop of only 0.12%. To merge local model updates returned by edge devices this MSc thesis proposes R-FedAvg, an implementation of the FedAvg algorithm that reduces the impact of faulty model updates returned by malicious devices.

Keywords: Federated learning, Machine learning, Artificial neural networks, Artificial intelligence, Machine learning algorithms, Intelligent systems, Internet of Things, Arm Cortex-M.

Resumo

O fim da Lei de Moore aliado às crescentes preocupações sobre a privacidade dos dados gerou a necessidade de migrar as aplicações de *Machine Learning* (ML) da *cloud* para o *edge*, perto da fonte de dados. Na próxima geração de sistemas ML, a inferência e parte do processo de treino será realizada diretamente no *edge*, enquanto que a *cloud* será responsável pelas principais atualizações do modelo ML. Este novo paradigma informático, referido pelos investigadores académicos e industriais como treino federativo, diminui a sobrecarga na *cloud* e na infraestrutura de rede, ao mesmo tempo que aumenta a privacidade dos dados. Avanços recentes tornaram possível a execução eficiente do processo de inferência de redes neurais artificiais quantificadas em microcontroladores Arm Cortex-M e RISC-V (RV32IMCXpulp). No entanto, o processo de treino continua confinado à *cloud*, impondo a transação de grandes volumes de dados privados sobre uma rede.

Para abordar esta questão, esta dissertação faz a primeira tentativa de realizar um treino descentralizado em microcontroladores Arm Cortex-M. Para migrar parte do processo de treino para o *edge* é proposto o L-SGD, uma versão *lightweight* do tradicional método *stochastic gradient descent* (SGD), otimizada para uma redução de latência do processo de treino e uma redução de recursos de memória nos microcontroladores Arm Cortex-M. O L-SGD é 16,35x mais rápido do que a solução disponibilizada pelo TensorFlow, ao mesmo tempo que regista uma redução de utilização de memória de 13,72%. O custo desta abordagem é desprezível, sendo a perda de *accuracy* do modelo de apenas 0,12%. Para fundir atualizações de modelos locais devolvidas por dispositivos do *edge*, é proposto o R-FedAvg, uma implementação do algoritmo FedAvg que reduz o impacto de atualizações de modelos não contributivos devolvidos por dispositivos maliciosos.

Palavras-chave: Treino federativo, *Machine learning*, Redes neuronais arificiais, Intelegência artificial, Algoritmos de *machine learning*, Sistemas inteligentes, Internet das coisas, Arm Cortex-M.

CONTENTS

List of Figures

List of tables

Nomenclature

AI	Artificial Intelligence
ANN	Artificial Neural Network
FCA	Federated Core API
FL	Federated Learning
FLA	Federated Learning API
FLS	Federated Learning System
FPU	Floating-Point Unit
IND-CPA	Indistinguishability Under a Chosen Plaintext Attack
IND-CTXTX	Indistinguishability Under Ciphertext Integrity
IoT	Internet of Things
ISA	Instruction Set Architectures
MCU	Microcontroller Unit
ML	Machine Learning
SGD	Stochastic Gradient Descent
SIMD	Single Instruction Multiple Data
TFF	TensorFlow Federated
UF-CMA	Unforgeable Under Chosen Message Attack

Chapter 1

Introduction

This chapter outlines the problem addressed in this MSc thesis, as well as the goals that must be achieved to develop a solution suitable for solving that problem. The chapter ends with a brief description of the structure of this document.

1.1 Problem Statement

With predictions pointing to 1 trillion connected devices by 2035 [1], Machine Learning (ML) has been used as a key technology for decision-making problems in the Big Data era [2]. Fields like autonomous driving [3], [4], security systems [5], [6], and even healthcare [7], [8] are already exploring ML solutions to develop smart systems capable of aiding or replacing human activity.

As ML depends on the availability of tremendous computational power, ML computations have been predominantly confined to cloud servers, powered by large CPUs, GPUs, and/or ASICs. In this centralized computing paradigm, data collected at the edge is transferred to a central server, which runs ML services and returns the generated output to the edge [9]. Notwithstanding, the expected increase of internet of things (IoT) nodes aligned with the end of Moore's law is threatening this centralized computing paradigm [10]. The expected overload of the network bandwidth and computational power of the cloud can induce unreasonable latencies in decision-making processes, which may delay the adoption of ML in scenarios demanding real-time response [11], [12]. Consequently, academia and industry have been developing software and hardware solutions to shift intelligence to the deep edge, providing it with the ability to autonomously infer and adapt to the surrounding environment, while leveraging the cloud to major model updates [13], [14]. This new paradigm, commonly referred to as federated learning (FL), considers that the inference and part of the training mechanism are performed on the edge, near to the data source, leaving the server with the responsibility of merging the minor model updates performed at the edge.

Moving ML algorithms to the edge can also be perceived as a major initiative for addressing additional requirements besides latency. FL also reinforces system security [10], [15] and power consumption [14] - two major development metrics in embedded systems design [16]. Security is increased as user data is kept on the data source and not transferred through a network, which reduces the attack surface. Power consumption decreases as the severe reduction of data transferred within a network reduce the energy dissipated on communication services.

However, this new computing paradigm comes with a new set of constraints, which are imposed by the tight resources available on the devices typically deployed at the deep edge [17]. Microcontroller units (MCUs) feature:

1. Limited memory footprint: Typical MCUs have their memories (RAM and Flash) limited to some hundreds of KBs or few MBs [15]. This calls for techniques to prune the input and output data of an artificial neural network (ANN), as well as for techniques to shrink and prune their internal weights.

2. Limited compute resources: The computational power is usually low on MCUs [17]. However, many classification tasks have always-on and real-time requirements, which limits the total number of operations per ANN inference and training.

Recent advances have made it possible to efficiently shift the inference part of an ML service to low-power MCUs. The most recent Arm Cortex-M and RISC-V (RV32IMCXpulp) MCUs already feature instruction set architectures (ISAs) that support single instruction multiple data (SIMD) tuned to speed up the inference pass of a quantized ANN with minimal accuracy loss [18]. Supported by open-source libraries such as CMSIS-NN and PULP-NN, porting an ANN to these families of MCUs is already a straightforward process. Nevertheless, to the best of the authors' knowledge, the training pass of an ANN has never been explored in this family of MCUs. Strictly confining the training pass to the cloud server fails the founding principles of FL, imposing the transaction of high volumes of private data over a network. Furthermore, shifting only the inference pass to the edge still overloads cloud servers with the full training process, which may disrupt unpredictable latencies when edge devices try to adapt to the surrounding environment by model retraining.

1.2 Aim and Scope

The main goal of this MSc thesis is the design and development of an FL framework tailored for the tight resource constraints of Arm Cortex-M MCUs. Before delving into more depth, it's important to note that the system must allow edge devices to infer and adapt to their surroundings even when there's no reliable contact with the cloud server. Furthermore, as the work targets Arm Cortex-M MCUs, the inference pass is considered to run upon the CMSIS-NN API. This ensures that this family of MCUs in the ML sector is compatible with current state-of-the-art research. Nevertheless, the framework must follow a modular architecture to allow future compatibility with other computing architectures and scale well with the growing number of connected devices.

In terms of the training algorithm, the research must begin with a review of the conventional methods for training ANNs. The evaluation aims to find a training method optimizable for Arm Cortex-M MCUs. The chosen algorithm is optimized for maximum speed and minimal memory footprint. A version of the training algorithm that only works with int-8 data must be built as the first step toward quantization training. Furthermore, the model accuracy, latency, and memory footprint of the float-32 and 8-bit versions must be compared.

An algorithm must be designed for the cloud server to combine the local parameters provided by edge devices. First and foremost, state-of-the-art aggregation approaches must be evaluated, with emphasis on avoiding data poisoning. If the current state-of-the-art aggregation algorithms' security safeguards are insufficient, a new algorithm must be developed. A malicious edge device cannot affect the accuracy of other FL networks because of this design constraint.

A framework that follows the set of goals defined above may unlock the paradigm shift of ML services to the deep edge, mainly composed of low-power MCUs, such as Arm Cortex-M. The main objectives for this work are outlined below:

- Design and develop an FL system architecture;
- Design and develop a lightweight training algorithm that meets the stringent requirements of ARM Cortex-M;
- Design and develop an algorithm for a reliable global model update that merges the ML parameters supplied by edge devices while avoiding biased models.

1.3 Dissertation Structure

This document divides into five main chapters, and this subsection outlines each chapter's content. The present chapter (Chapter 1) introduces the problem that this MSc thesis addresses, and details the objectives that must be achieved to provide a reliable solution for that problem.

Chapter 2 splits into two main subsections: (i) background knowledge and (ii) related work. The fundamental concepts are presented in the first subsection. The subsection relies on two main topics that consist of machine learning key-knowledges and federated learning concepts. The second subsection relies on developed work on this research area. The research scrutinizes academia and industry solutions proposed in the recent past.

Chapter 3 outlines the design and development of the mechanisms to port the ML process to the deep edge devices. The chapter splits into three main topics: (i) system architecture, (ii) edge training, and (iii) FL server. The first subsection aims to outline the federated learning system (FLS) design and the communication workflow. The second subsection outlines the design of the deep edge training algorithm (L-SGD). Finally, the third subsection focus on the aggregation mechanism (R-FedAvg).

Chapter 4 presents the experimental results of the tests performed on the different FL framework processes are presented. The chapter is divided into two main sections, which address the results for the training (L-SGD) and aggregation algorithms (R-FedAvg). The first subsection evaluates the training algorithm under three defined metrics: (i) accuracy, (ii) latency, and (iii) memory footprint. It evaluates L-SGD against SGD and quantized L-SGD against floating-point L-SGD. The second subsection evaluates the aggregation algorithm (R-FedAvg) for these same metrics and under different data partition and perturbation scenarios.

Finally, Chapter 5 discusses the results of the developed work, pointing out some found limitations. The chapter ends with future work suggestions that intend to solve some current limitations and extend the functionalities of the developed system.

Chapter 2

State of the art

This MSc thesis aims to design and develop an FL framework for next-generation ML systems. The proposed framework is composed of a decentralized learning system architecture and a lightweight training algorithm. The deployment target of the training algorithm is the ARM Cortex-M family of MCUs. The current chapter outlines the state-of-the-art, which covers the FL fundamental concepts and technologies. Consequently, this chapter splits into two sections. The first subsection focuses on providing ML background knowledge to applications development and mechanisms to port these applications to the embedded environment. The second subsection analyzes current state-of-the-art solutions for FLS.

2.1 Background

This section covers some concepts essential for a critical analysis of FLS. The first subsection aims to cover the main concepts to develop this work. It includes an analysis of six different topics: (i) ML, (ii) ANN, (iii) clustering algorithms, (iv) training algorithms, (v) ANN Quantization, and (vi) FL.

2.1.1 Machine Learning

ML, a subfield of Artificial Intelligence (AI), can be defined as a set of algorithms with the ability to learn without being explicitly programmed, based on the experience obtained by the previous analysis of data [19]. It aims to design and develop algorithms to search patterns, allowing the system to build a model capable of performing predictions over unseen data. After all, the goal of ML is to generate the rules of a specific environment, based on examples of the rule's application [20].

Implementation of ML algorithms follows two sub-processes: the training phase, based on collected data, and the prediction/inference phase, based on new data that the model has no previous information. In the first phase, the ML algorithm must extract patterns from the provided training data to develop a predictive model. In the second phase model performs data-driven predictions [4], [21]. This phase matches the inference process and is processed after model deployment. Model training process finishes when it meets a metric requirement, such as accuracy, precision, recall, or F-Score. Figure 2.1 shows a graphical representation of the different processes involved in the ML development workflow and the connection between them.

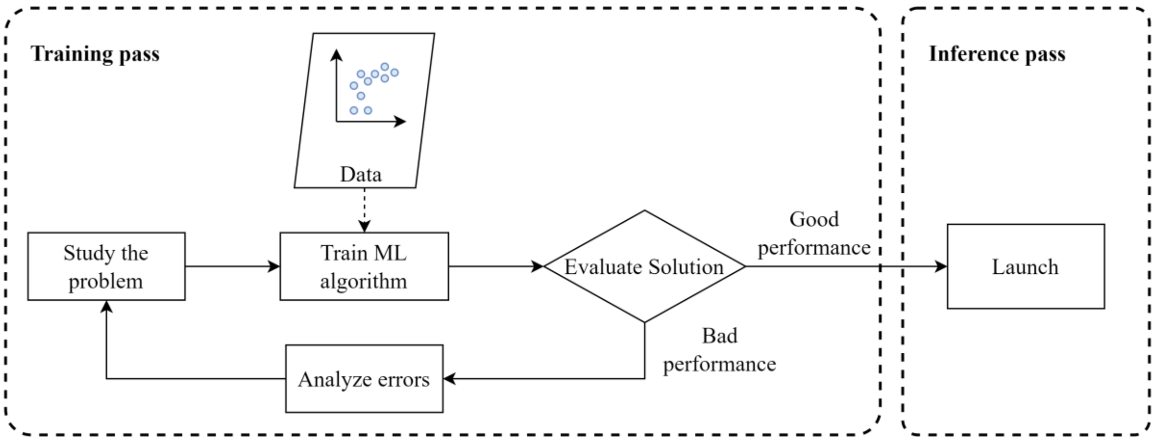


Figure 2.1: ML development workflow

ML algorithms rank into three learning subfields: supervised, unsupervised, and reinforcement learning [19]. Supervised learning focuses on the relation between inputs and outputs, where the main goal is to establish a function able to perform a prediction for new and unseen input data [19]. Therefore, a model trains with input samples and the respective desired output (labeled data). During this process model's parameters are updated to reduce the magnitude of a loss function, where the goal is to minimize the difference between the predicted and the expected values [22]. To ensure that model can perform predictions on unseen data, its development must consider a certain degree of generalization during training, avoiding the overfitting problem. Supervised learning splits into two sub-fields: classification, where the goal is to predict the category inside the discrete number of possible outputs, and regression, which pretends to predict continuous values for the incoming data [23]. Some common applications of supervised learning include predictive analysis, email spam detection, patterns detections, as well as human-related activities, such as natural language processing, image classification, and even sentiment analysis [22].

In contrast, unsupervised learning algorithms overcome the supervised approach when the available training data is not labeled. In this case, the main goal is to learn how a set of samples can be correlated based on hidden patterns. Contrary to supervised learning, there are no output labels to predict. Consequently, there are no absolute error measurements. Unsupervised addresses into two sub-fields: clustering, which consists of grouping elements based on their similarity, and dimensionality reduction, which is the process of reducing the number of features in a given dataset while keeping the most relevant information [24]. Some conventional applications of this learning method include object segmentation [25], similarity detection [26], and automatic labeling [27]. Thanks to the last referred application, it is possible to use supervised learning when it is necessary to categorize a large amount of data with a few labeled examples, or when there is a demand to impose some constraints to clustering algorithms [24]. This new approach is denominated as semi-supervised learning since it is a mashup of both described learning methods.

When the interaction with the environment is very dynamic and not deterministic, being impossible to perform error measures, it is necessary to integrate an algorithm capable of maximizing its performance based on interactions with that same environment. In this segment, reinforcement learning comes up as a set of algorithms in which the training process relies on the feedback given by the interaction with the environment. Such feedback can be positive, usually denominated as a reward, or negative, also designated as a penalty. The goal of reinforcement learning is to automatically determine the ideal behavior of a system, trying to get the highest immediate and cumulative reward, which means that the decisions taken by the system are, within a specific context, correct [24].

2.1.2 Artificial Neural Network (ANN)

ANNs are a set of information processing structures based on the biological nervous system. The goal of these mechanisms is to solve classification and regression problems [23]. The architecture of an ANN follows the human brain's nature. The interconnection of multiple nodes, where each one represents a neuron, builds an analogy to the human brain. Each cell receives one or multiple signals as input, in biology terms denominated as synapsis. The input of a given node is the output of a previous node multiplied by a weight, representing the connection of multiple dendrites on the human brain [28]. Then, the product of every operation is summed, obtaining the total input signal to which a bias value is added. After this sum, the result is filtered by an activation function, producing the neuron output, which can be passed as input to other neurons, forming complex ANNs [28]. Figure 2.2 shows the described process.

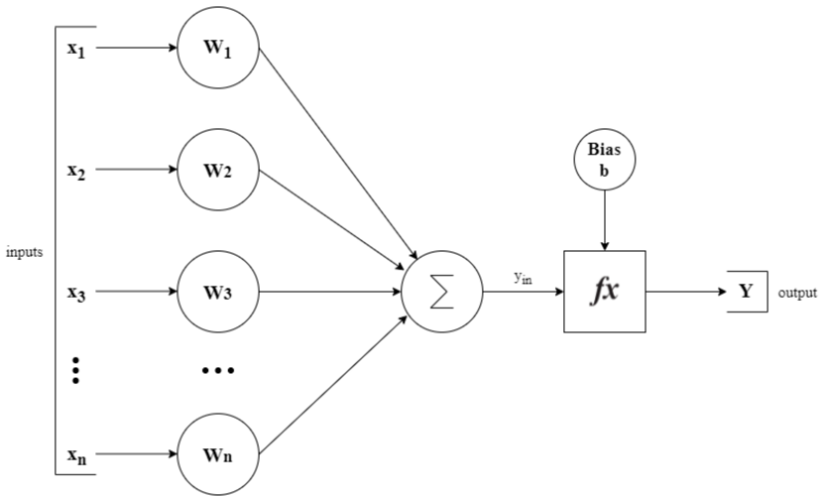


Figure 2.2: Model of a neuron in an ANN

Due to the large number of nodes that constitute an ANN, there is a need to manage the connections between them – define an ANN architecture. The connection setting between different nodes sets the network as feed-forward or feedback. In feed-forward ANNs, nodes are grouped into layers, and only one-direction data flow is allowed. This way, the output of a given layer matches the input of the following layer. Consequently, there are no feedback loops. This way, the layer's output does not affect itself. Otherwise, the feedback network has feedback paths, meaning that a signal can travel in both directions using loops. Hence, the outputs computed from previous layers are fed into the neural network, introducing a memory capacity in the process. Furthermore, each node is connected to all other nodes, working simultaneously as input and output [29].

The activation function is part of each node. As previously described, the computation of the input data of a node is filtered through the integration of an activation function, which is a mathematical function to delimit the output range of a single neuron [28]. These functions can either be linear or non-linear. The function choice process falls on the domain application of the neural network, which can be part of areas such as object recognition and classification, speech recognition, or even healthcare applications [30]. In the linear activation functions, the neuron only maps the inputs and introduces an output range. The non-linear functions can truncate the output by limiting the output range. The most common activation functions as well as their mathematical expression are detailed in Table 2.1.

Table 2.1: Activation functions

Activation function	**Mathematical expression**
Binary Step function	$\begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$
Hyperbolic Tan (tanh)	$\frac{e^x - e^{-x}}{e^x + e^{-x}}$
Rectified Linear Units (ReLU)	$\begin{cases} 0 \; for \; x < 0 \\ x \; for \; x \geq 0 \end{cases}$
Sigmoid	$\frac{1}{1 + e^{-x}}$
SoftMax	$1 + \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$

As there are a set of configurable parameters in an ANN, there is a training process where those parameters, such as weights and bias, are tuned to improve the ANN performance, usually measured in terms of accuracy, recall or precision. The training can be performed under a supervised, unsupervised, or reinforcement approach, as previously described. The selection of a loss function is related to the application domain. Probabilistic losses, regression losses, or hinge losses “maximum-margin” classification, are three categories of loss functions with different mathematical approaches, being each one of them composed per multiple functions.

2.1.3 Clustering Algorithms

On the opposite side of supervised learning, unsupervised algorithms do not depend on a labeled dataset. This paradigm shift allows the analysis of large amounts of data through pattern identification [31]. Clustering algorithms are unsupervised learning methods that allows the organization of similar instances into clusters, as is presented in Figure 2.3. Input samples with similar characteristics belong to the same class. This approach allows the implementation of anomaly detection such as fraud or malicious operations [32]-[34]. The optimization of datasets to feed supervised learning algorithms is relies on unsupervised methods to find samples that don't fit into a certain prediction class. This section outlines a brief analysis of two clustering algorithms: (i) k-means clustering and (ii) hierarchical clustering.

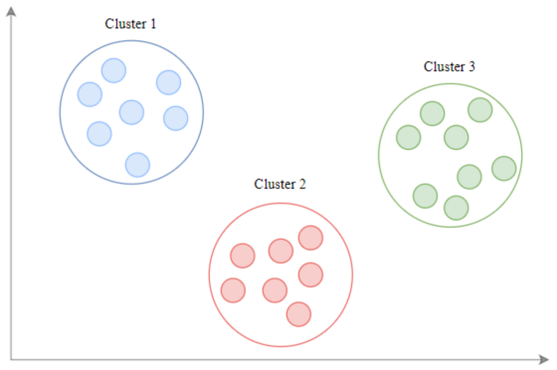


Figure 2.3: Data clustering

k-means: K-means clustering is an algorithm that aims to split data into k clusters. The number of selected clusters matches the number of centroids used in the clustering process. Each sample assigns to a cluster based on the distance to the near centroid. After classifying each sample, there is a cluster's centroids recalculation, and the process repeats [35]. However, k-means cannot evaluate which clustering is the best option, as it happens on supervised learning. The only option is to keep track of the clusters and their total variance and repeat the process with different starting points for each centroid.

Improving dataset quality is a practical application of k-means [36] in which k is an easy tunable parameter. In some scenarios, k values demand an iterative tuning process to achieve low variance values. In an extreme case, variance equals zero when k is equal to the number of samples to be clustered. However, this approach leads to a redundant relation between samples, which goes against the clustering purpose. This way, the elbow method is usually used to tune the k parameter. Figure 2.4

presents a hypothetical example of the elbow method implementation. In this case, variance reduction is achieved at a high rate until the setting of k equals two. After that, increasing the k values does not present a significant variance reduction.

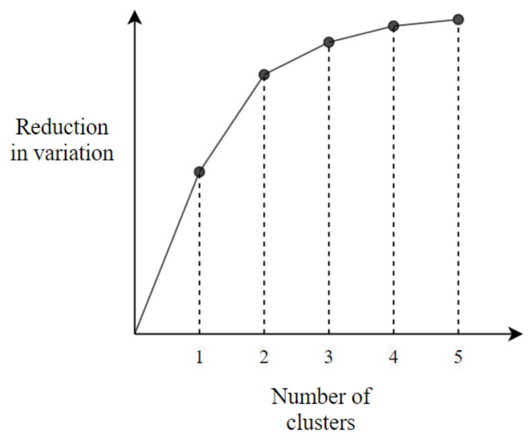


Figure 2.4: Elbow plot method

Hierarchical clustering: Hierarchical clustering is a distinct technique to clustering. Rather than establishing the number of clusters, hierarchical clustering algorithms try to create a hierarchy of clusters. Furthermore, each cluster border is limited by a threshold distance. The construction of a dendrogram, as shown in Figure 2.5, is a basic method for defining clusters. In this case, two clusters are easily identified. However, the threshold sets the clustering granularity. The higher the threshold value, the lower number of clusters. Furthermore, hierarchical approaches can either be divisive (top-down) or agglomerative (bottom-up) [37]. The divisive technique requires a way for iteratively splitting clusters until samples are clustered independently. The agglomerative technique takes each sample as a single token and defines each sample as a member of a cluster until all of the data is gathered into a single cluster. To summarize, divided clustering is a more difficult procedure. The top-down technique, on the other hand, yields more accurate outcomes.

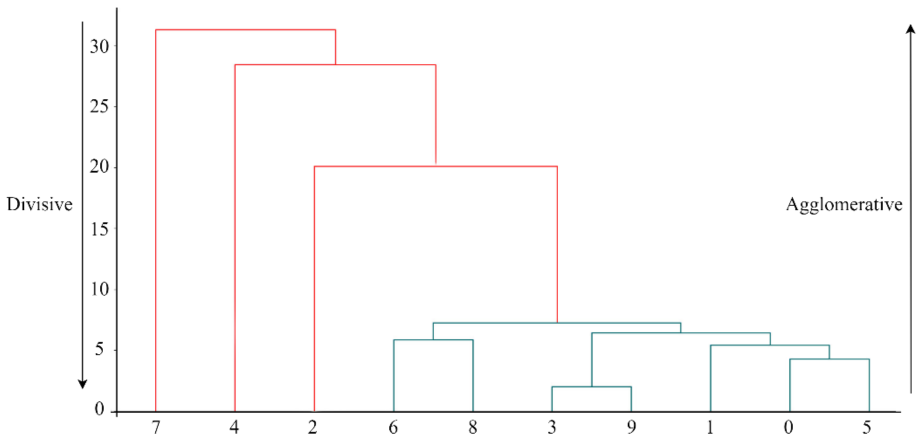


Figure 2.5: Hierarchical clustering

2.1.4 Training Algorithms

The ANN training process is an optimization problem [38]. The optimization target is the model's weights that are optimal when the model's prediction error is minimum. The learning process of an ANN is the process of updating weights values taking the model to improve prediction precision. As a result of a large number of configurable parameters of an ANN, the training process is an iterative and exhaustive task. In opposite to the ANN inference process where there is feed-forward pass [39], the model update requires a different approach. Minimum loss value requires multiple backward pass steps [40].

The backpropagation technique needs to perform for several iterations/epochs before reaching an optimal parameter configuration. For this reason, there are multiple stopping rules to define the end of the training process. The most common stopping rules are (i) stop after a specified number of epochs, (ii) stop when an error measure reaches a threshold, and (iii) stop when the error measure has seen no improvement over a certain number of consecutive epochs [41]. Since the training process can iterate the entire dataset multiple times through the definition of multiple epochs, it is essential to avoid data layout correlation during the training. This means that the dataset layout must not induce any learning effect on the training model. This way, shuffling data is one of the most important steps in data preparation to achieve the best prediction accuracy. Furthermore, there are three popular shuffling mechanisms: (i) Fisher-yates (old version), (ii) random values sorting, and (iii) Fisher-Yates (modern version).

Fisher-Yates (old version): Fisher-Yates's algorithm (old version) [42] suggests the random selection of an element from the non-shuffled elements until there is no unshuffled data. The original approach design does not aim at software devices. Thus, time complexity relies on data size due to algorithm complexity which is unscalable to large datasets. The memory allocation of this mechanism equals the original data size.

Random values sorting: An algorithm tweak performs by generating random values and assign them to each data index. Further, the generated numbers sorted in ascendant or descendent order create the shuffled data. Once more, the time complexity is directly affected by the amount of data to be shuffled. Thus, time complexity can escalate very fast, even still at a lower rate than the first solution. However, space complexity keeps the same, requiring the memory allocation equal to the original data size that doubles the original storage usage [42].

Fisher-Yates (modern version): The most recent version of Fisher-Yates was developed to minimize both time and space complexity [42]. This shuffling algorithm operates directly on the original data by splitting the unshuffled data and already shuffled data. An index pointing to the last shuffled sample ($last_{index}$) is defined and a random number between zero and the last number index ($shuffle_{index}$) is generated. The sample at the index $shuffle_{index}$ is swapped with the sample in the $last_{index}$ position, and the $last_{index}$ is decremented. The process iterates over the data until de last index matches the data's first position meaning that the dataset is shuffled [42]. Table 2.2 present different shuffling algorithms.

Table 2.2: Shuffling algorithms comparison

Solution	**Time Complexity**	**Space Complexity**
Fisher-Yates (old version)	O(n^2)	O(n)
Random values sorting	O(n log(n))	O(n)
Fisher-Yates (modern version)	O(n)	O(1)

2.1.4.1 Gradient Descent (GD)

The GD is the most basic optimizer. The optimization relies on the first-order derivative of a loss function. Based on the value returned by the loss function, the GD calculates the direction of change of the model's parameters (weights and bias). Updating the parameters multiple times through this mechanism leads to an optimal solution that reduces the prediction loss. The low computational complexity of this algorithm allows its deployment on deep edge devices. However, this mechanism requires an entire dataset at a time to update the weights and bias, which substantially increases the memory footprint [43].

2.1.4.2 Stochastic Gradient Descent (SGD)

SGD is an extension of GD that aims to tackle the large memory footprint concern [44]. SGD uses only one sample at a time rather than the entire dataset. Besides the memory footprint reduction, SGD is less vulnerable to local minima stuck. However, the time to complete a single epoch is large compared to the GD algorithm. A simple example of an ANN, where it is analyzed the computation process of the backpropagation algorithm, is represented in Figure 2.6. For simplicity, the model considers weights only and does not include bias values. The ANN is feed with a training sample, and the input features are propagated through the network. For each neuron, an input (in_{neuron}) and an output (out_{neuron}) value are defined, as well as an activation function (F). The forward pass equations of the first hidden layer are defined in Equations (1)-(4). The output layer follows the same method.

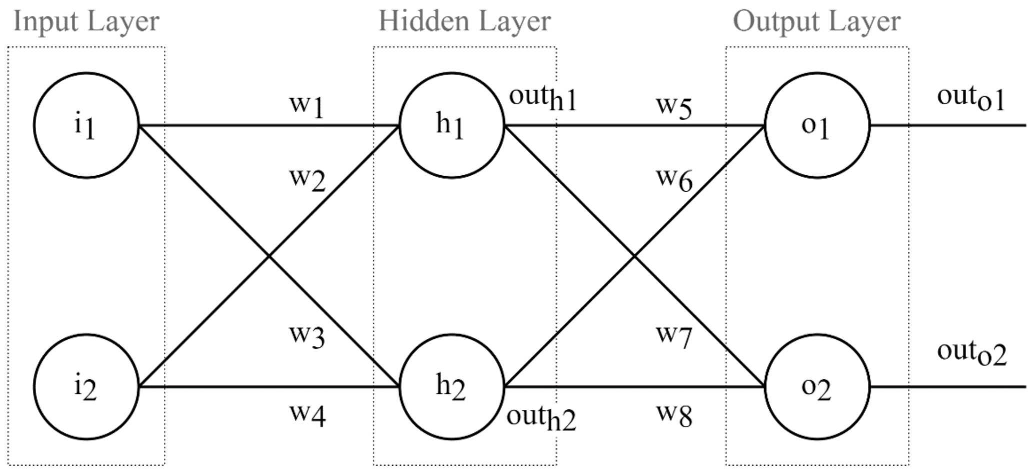


Figure 2.6: ANN overview

$$in_{h_1} = i_1 * w_1 + i_2 * w_2 \tag{1}$$

$$in_{h_2} = i_1 * w_3 + i_2 * w_4 \tag{2}$$

$$out_{h_1} = F(in_{h_1}) \tag{3}$$

$$out_{h_2} = F(in_{h_2}) \tag{4}$$

After producing the model prediction, a loss function calculates the distance between the prediction of the ANN and the corresponding true label. This error is optimized using the so-called training algorithms. From the set of training algorithms available, the most prominent is the SGD, which provides information about the direction of the weight update. According to SGD, the update of a given weight is calculated as described in Equation (5), in which η is the learning rate.

$$w_i^+ = w_i - \eta * \frac{\partial E_{total}}{\partial w_i} \tag{5}$$

$$\frac{\partial E_{total}}{\partial w_1} = \left(\frac{\partial E_{o_1}}{\partial in_{o_1}} * \frac{\partial in_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial in_{o_2}} * \frac{\partial in_{o_2}}{\partial out_{h_1}}\right) * \frac{\partial out_{h_1}}{\partial in_{h_1}} * \frac{\partial in_{h_1}}{\partial w_5} \tag{6}$$

After comparing the model prediction with the true label, the loss function calculates the loss that will be used to update the weights in the backward pass. In the backward pass, the effect of each weight in the prediction error is calculated. This way, each weight is updated proportionally to its error magnitude effect. The main goal is to reduce the error disrupted by each weight to ideally zero. Since an error at a given layer propagates to the forwarding layers, the weights are updated only after the computation of the error propagation effect. The error propagation effect is represented in Equation (5) by the term $\frac{\partial E_{total}}{\partial w_i}$. The calculus of this term follows a chain rule as detailed in Equation (6). As can be observed for the particular case of w_1, the chain rule depends on the values returned in the previous training iteration for the weights w_5 and w_7. This pattern is verified for any weight in a hidden layer of an ANN and is responsible for making SGD require the allocation of a memory block that doubles the size of the weights.

2.1.4.3 AdaGrad

The major obstacle of GD-based approaches is the setting of the learning rate. Both GD and SGD mechanisms keep this hyperparameter constant along the training process. AdaGrad [45] is a solution that allows the implementation of an optimizer without a manual tunning of the learning rate. Moreover, this hyperparameter decreases as the number of training iterations increase. Nevertheless, as the number of iterations becomes very high, the learning rate decrease to a very low value, which leads to a much slower convergence [43]. Adagrad automatically adjusts the learning rate based on a parameter. To avoid overshooting the minimum value, parameters with larger gradients or frequent updates should have a slower learning rate. Parameters with low gradients or few updates should increase the learning rate to ensure that they are quickly trained. AdaGrad divides the learning rate by the sum of squares of previous gradients of the parameter. When the sum of the squared gradients is high it divides the learning rate by a large number, leading the learning rate to decrease. Similarly, if the total of the squared prior gradients is low, the learning rate is divided by a smaller number, resulting in a high learning rate. This means that the learning rate is inversely proportional to the sum of the squares of the parameter's prior gradients. Equation 7 outlines the AdaGrad update process.

$$G_t = \sum_{\tau}^{t} g_\tau \times g_\tau^T$$
$$w_{t+1} = w_t - \eta * diag(G_t)^{\frac{-1}{2}} \tag{7}$$

2.1.4.4 Adam

The introduction of the exponential update of the learning rate showed improvements in the training process execution time. Moreover, exploring this technique led to the development of new mechanisms. Adam introduces a new term to the calculus of the learning rate update. Adam [46] optimizer stores both the first and second-order moment of the gradient, rather than storing exponential decaying averages of the square of gradients to update the learning rate [43]. First, it computes the exponentially weighted average of past gradients (mt). Then, it computes the exponentially weighted average of the squares of past gradients (v_t). Then, these averages have a bias towards zero and to counteract this a bias correction is applied (m_{t+1}, v_{t+1}). Lastly, the parameters are updated using the information from the calculated averages. This process is outlined by Equation 8. Adam optimizer demands the tunning of three hyperparameters: β_1, β_2, and η. The term ϵ is a very small value introduced to avoid division by zero.

$$\begin{gathered} m_{t+1} = \beta_1 \times m_t + (1 - \beta_1) \times g_t \\ v_{t+1} = \beta_2 \times v_t + (1 - \beta_1) \times g_t^2 \\ m_{t+1} = \frac{m_{t+1}}{1 - \beta_1^t} \\ v_{t+1} = \frac{v_{t+1}}{1 - \beta_2^t} \\ w_{t+1} = w_t - \frac{\eta}{\sqrt{v_{t+1}} + \epsilon} \times m_{t+1} \end{gathered} \tag{8}$$

2.1.4.5 Gap analysis

In comparison to other optimizers, the GD [43] optimizer stands out for its computational simplicity. This method uses the complete training dataset, which increases memory footprint. Furthermore, SGD [44] mitigates this impact by using a subset of the dataset in each iteration (mini-batch). Despite the memory decrease achieved by employing this method, the process becomes slower, and the memory footprint for deep edge devices is still prohibitive. Nonetheless, other methods for speeding up the training process have been proposed (comparison depicted in Table 2.3).

Table 2.3: Functionality of optimizers

Optimizer	**Tunable parameters**	**Advantages**	**Disadvantages**
GD [43]	1	Low computational complexity	High memory footprint Vulnerable to local minima stuck
SGD [44]	1	Low computational complexity Reduced memory footprint	Vulnerable to local minima stuck Slower than GD
AdaGrad [45]	1	Automatic learning rate decay	Slower convergence at later iterations
Adam [46]	3	Faster training	High memory footprint Higher number of tunable parameters

AdaGrad [45] is proposed as a possible solution for reducing the training process execution time. This technique emphasizes the continuous adjustment of the learning rate resulting in a shorter training period. The convergence process gets significantly slower as the number of iterations grows. To counteract this, Adam [46] outperforms the other algorithms by showing the fastest training mechanism and reducing the effect of increasing the number of training steps. These techniques are quicker than GD-based mechanisms, but their resource usage is insufficient to fulfill the hardware constraints of the target platforms. The computations performed by these approaches are more exhaustive than those performed by GD-based methods, in addition to the significant increase in memory use. As a result, there is a demand for a solution that assures the most efficient use of resources while still ensuring appropriate training times in a federated training environment.

2.1.5 ANN Quantization

ANN models training typically performs over 32-bit floating-point data, which gives a high precision that is usually not required during inference [47]. Research has shown that neural networks with low-precision fixed-point representation can produce equivalent results to the traditional approach [47], [48]. Further, a fixed-point quantization approach can reduce the cost of floating-point computation and the memory footprint for storing both weights and activations, two critical metrics in resource-constrained platforms. This process, known as quantization, involves the encoding of (i) the sign, (ii) the integer part, and (iii) the fractional part of a float in a single integer value. Quantized values are typically represented in $Qm.n$ format, where m specifies the number of bits for the integer part and n the number of bits for the fractional part. Quantization not only decreases the inference latency but also reduces the memory footprint of ANNs [49]. Nevertheless, quantization must always attend to the ISA of the platform where the ANN will be deployed. This is fundamental to speed up memory access and math operations. Moreover, the quantization process relies on five fundamentals: (i) scaler and zero-point, (ii) quantization aware training vs. post-training quantization, (iii) quantization granularity, (iv) fixed bit-width vs mixed bit-width, and (v) static vs. dynamic quantization.

Scaler and zero-point: The quantization of a floating-point value always requires scaling and zero-point factors [50], [51]. Scalers are typically calculated as detailed in Equation 9 and require the previous definition of the quantization format $Qm.n$ and the range of floating-point values to be quantized. The zero-point is usually calculated as detailed in Equation 10 and is always rounded to an integer value.

$$scaler = \frac{x_{max} - x_{min}}{2^n - 1} \tag{9}$$

$$zero = -round(x_{min} * scaler) - 2^n - 1 \tag{10}$$

Quantization aware training vs. Post-training quantization: As quantization reduces the number of bits to represent a value, saturation is a common problem that affects the accuracy of a quantized ANN. A common strategy to address this problem is to perform quantization-aware training, in which the quantization error is considered as part of the loss returned by the loss function [50], [52]. Nevertheless, this comes at the cost of increased overhead and latency in the training pass. In contrast, post-training quantization does not incur additional overhead during training time, as quantization is only performed after the training of the floating-point ANN.

Quantization granularity: The quantization granularity must consider two metrics: (i) impact on model accuracy and (ii) computational cost [50], [51]. Computing the scaling factor to each weight and activation leads to almost null accuracy loss. Nevertheless, this is not feasible as each neuron requires a specific scaling operation during inference. Besides increasing the decision latency, it also tremendously increases the memory footprint of the quantized ANN, as scaling and zero factors need to be stored for each weight. The most common quantization strategies consider a layer-by-layer or filter-by-filter granularity [50], [51].

Fixed bit-width vs. mixed bit-width: In fixed bit-width, the number of bits to represent a given weight or activation is the same in the whole ANN [50], [52]. In contrast, in a mixed bit-width setting, the number of bits to represent weights and activations may vary between layers or filters, depending on the quantization granularity [50], [52].

Static vs. dynamic quantization: In static quantization, the quantization format is set before inference, using a representative dataset [50], [51]. In contrast, dynamic quantization computes the quantization format during the inference process for each input [50], [51]. Therefore, model parameters and activations are stored in low-precision bit-width but the operations are performed in floating-point data. For each calculus, this approach requires the dequantization of the inputs and the quantization of the outputs. Dynamic quantization lowers the impact of quantization on model accuracy when compared with the static approach; however, it incurs an overhead that may be prohibited for Arm Cortex-M MCUs, especially those not featuring hardware floating-point units (FPU). Dynamic quantization is usually used when memory is a concern but processing power is not.

2.1.6 Federated Learning

ANN's inference process is a heavy process resulting from the operations involved in each layer computation. Considering that typically ANNs use parameters (weights and bias) defined as 32-bit floating-point values, the inference process becomes an even more demanding problem, especially for devices that do not feature FPU. Since that edge hardware is usually resourceless, most ML applications follow a centralized approach. This approach takes the inference process to be performed on the cloud, reducing the edge overload. In centralized ML systems, the inference process runs on the cloud. Lastly, the prediction is sent back to the edge [50], [51].

The main goal of FL is to port the inference and at least part of the training process to the data source [53]. Edge devices are considered part of a network and are directly connected or connected through a central server[54]. Nevertheless, each edge device is considered autonomous as it can infer about the surrounding environment without any interaction with third parties. Furthermore, they can autonomously adapt to the surrounding environment by performing model re-training. In the re-training process, the architecture of the ML model is usually maintained and only the weights or parameters are updated. Nevertheless, some authors also propose the update of the ML model architecture [55]. Edge devices will be periodically asked to share their new models or model parameters. Figure 2.7 depicts the workflow of a FLS.

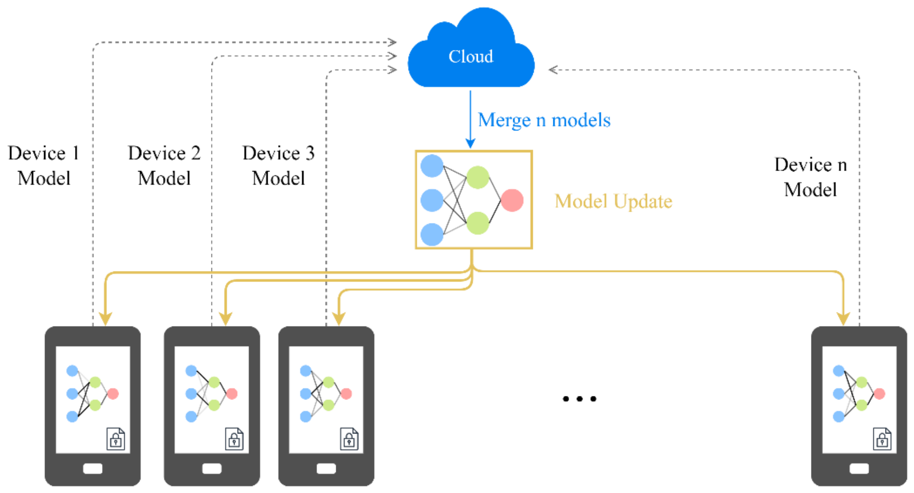


Figure 2.7: Overview of an FLS

The decentralized approach also deals with the privacy risk of centralized learning. Moving the training process to the edge allows data-held on the device. The development of smart systems requires the collection of user data, usually private. User private data is not transferred between nodes and servers, reducing the attack surface of ML systems. Instead of privacy-sensitive data, only model-related data is transmitted. Thus the communication process is reduced to the download of training plans and the upload of training results. The uploaded results are aggregated on the server to create an updated model based on multiple local training processes.

Private FLS faces a challenge: efficiently distribute computation to data centers under the constraint of privacy models [56]. Further, due to the energy consumption concern, complex training tasks can not be demanded by devices. Considering edge hardware constraints and data-privacy concerns, FLS should be robust enough to manage a large number of parties, considering the possibility of connection issues between each device and the central server.

Li et al. [56] proposed a taxonomy for the design of a FLS. As shown in Figure 2.8, the taxonomy of FLSs comprises six components: (i) data partitioning, (ii) ML model, (iii) privacy mechanism, (iv) communication architecture, (v) scale of federation, and (vi) motivation of federation. There are three components essential to define the training method at the edge and synchronizing with the server: (i) data partitioning, (ii) communication architecture, and (iii) scale of federation.

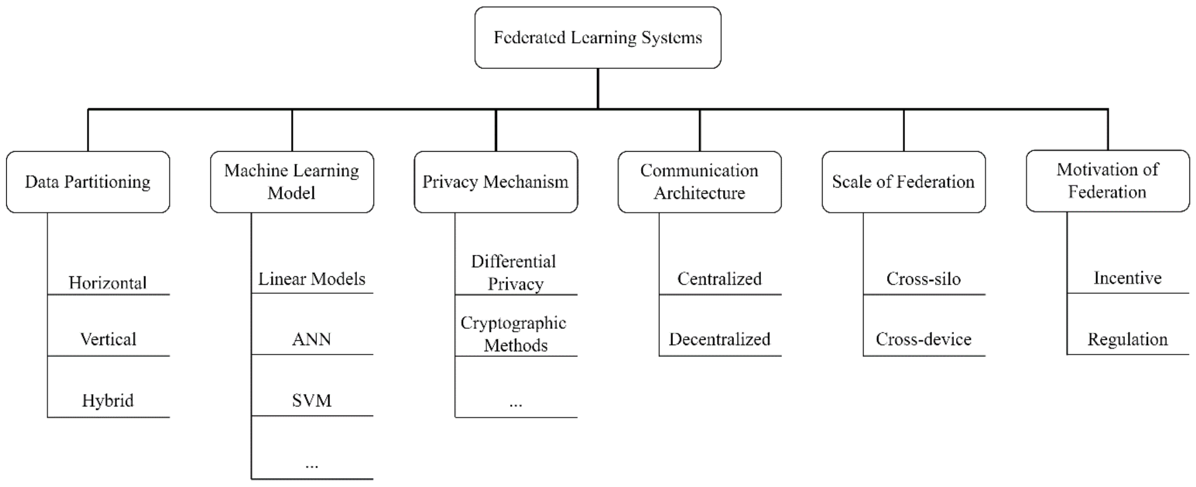
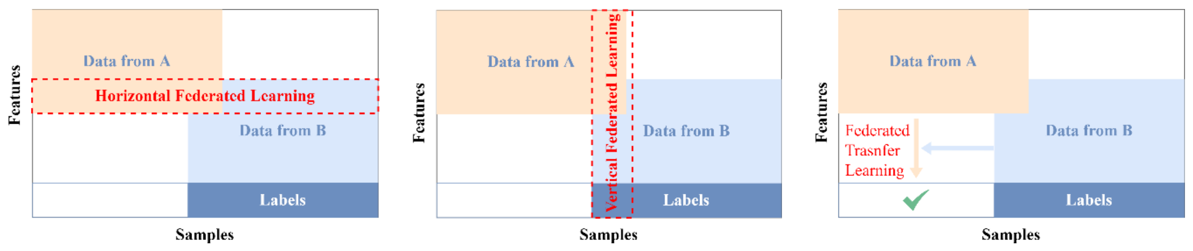


Figure 2.8: Taxonomy of FL

Data partitioning: Data partitioning can be categorized as horizontal, vertical, and hybrid, depending on how data distributes over the sample and feature spaces [57]. While the sample space matches the set of all possible input samples, the feature space is related to the properties of the input data. Horizontal, or sample-based data from different devices, share the same feature space but low intersection on the sample space (Figure 2.9a). This strategy is a common data partitioning in FLSs due to the cross-device scenario, where multiple devices try to improve their model performance through FL. The training process performs locally where devices share the same feature space and use the same model architecture. The main challenge is to aggregate models from different parties in the cloud server. Averaging all the local models has shown decent results, and it is a simple and effective approach [56].

In vertical or feature-based FL, data from different devices have the same, or at least similar, sample space but differ in the feature space (Figure 2.9b). Vertical data partitioning usually adopts entity alignment techniques to collect the overlapped samples of different devices. Therefore, vertical FL is the process of aggregating these features and computing the training loss and gradient using multiple device data, preserving its privacy. There are some scenarios where datasets differ not only in the sample but also in feature space. In this case, parties are a hybrid of horizontal and vertical partitions. Transfer learning [58] can be a solution to such scenarios (Figure 2.9c).



(a) Horizontal Federated Learning (b) Vertical Federated Learning (c) Federated Transfer Learning

Figure 2.9: Data partitioning in FLS

Communication architecture: Communication architecture is a basic design decision for the FL system. Typically, there are two major approaches: (i) centralized design and (ii) decentralized design. In a centralized design, the data flow is asymmetric. A central server is responsible for aggregating the ML models or parameters, returned by each edge device, and sending back the training results and the updated ML model [16]. In a centralized communication architecture, the data transfer between the manager and the edge can be (i) synchronous or (ii) asynchronous. In a synchronous setting, the central server is responsible for signalizing the beginning of the training pass to the edge [59]. The central server waits until every edge device or at least a portion of them sends the updated parameters. In an asynchronous setting, an edge device can start a new training pass at any time [60]. When the central server wants to deploy a new global ML model, it aggregates the most recent parameters available from each edge device.

Scale of federation: The number of users of a FLS defines the scale of federation of the system. Typically, such categorization splits into two types: cross-silo FLS and cross-device FLS [61]. In cross-filo FLS, there are a few parties and each one stores large amounts of data and presents high computational power. This is a typical situation in organizations with data privacy constraints, such as hospitals and bank corporations. In opposition, cross-device FLS is composed of a large set of parties, and each one stores low amounts of data and presents a low computation power [62].

2.2 Related work

This section is dedicated to the analysis of the current solutions, provided by the scientific community to implement FLSs. Rising concerns about data privacy aligned with the end of Moore's law and the ever-growing number of IoT devices are forcing intelligence to shift from the cloud to the deep edge, near to the data source. This paradigm gave rise to new computing paradigms, from which stands out federated learning. In this computing paradigm, the inference and part of the training passes must be performed on the edge, while the cloud is left for periodic global model updates.

2.2.1 CMSIS-NN and PULP-NN

Arm developed CMSIS-NN, a library to maximize the performance and minimize the memory footprint of ANNs, being Arm Cortex-M processors the target [12]. Neural network inference process based on CMSIS-NN kernels reveals a runtime/throughput improvement, as well as lower energy consumption [12], [63]. Figure 2.10 depicts the structure of CMSIS-NN kernels, which are composed by two main parts: (i) *NNFunctions* and (ii) *NNSupportFuntions*. The first one includes the functions that allows the implementation of neural network layers, including convolution, fully-connected, pooling, and activation. In contrast, *NNSupportFuntions* include utility functions, such as data type conversion and activation function tables, which are used in *NNFunctions*.

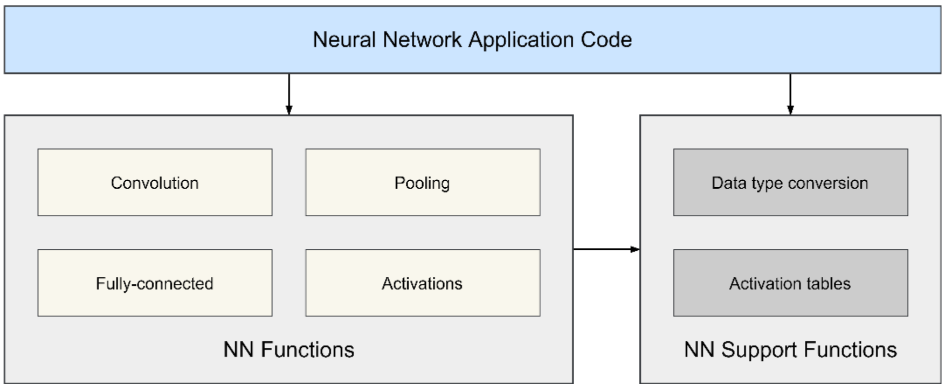


Figure 2.10: ANN kernel structure overview

CMSIS-NN kernels support data in 8 and 16 bits. For this reason, a neural network being normally trained in float-32 has to be quantized. CMSIS-NN assumes a static quantization, with fixed bit-width, and conducted layer by layer. The quantization process follows the flow presented in Algorithm 1. As detailed in Algorithm 1, for each layer, the quantization format of the input and output data must be defined. Since quantization is static, you must define these formats prior to the inference time. Quantization can be performed either during training, using quantization-aware training, or immediately after training, using a reference dataset (post-training quantization). The critical point in this quantization process is the definition of the maximum value. The output Q-format of some layers (i.e. fully connected and convolutional layers) can be defined independently from the input format. As the quantization relies on a power-of-two scaling (Algorithm 1), the scaling of a layer output is implemented as a bitwise shift operation. Such adjustment is introduced over two shifting parameters: *bias_shift* and *out_shift*. Both parameters are calculated over Equation 11 and Equation 12 and set individually to each layer.

Algorithm 1: Quantization algorithm

```
1:  get $Qm.n$ format:
2:      find maximum absolute value $max_{abs}$
3:      find amount $m$ of bits to represent the range $[-\mathrm{max_{abs}}, \mathrm{max_{abs}}]$: the ceiling of $\log_2(\mathrm{max_{abs}})$
4:      calculate amount $n$ of bits for the fractional part: $7 - m$
5:  convert to quantized integer:
6:      multiply floating-point values $A$ by 2 powered to the number of fractional bits: $A \times 2^n$
7:      clip quantized integers for the range $[2^n, 2^n - 1]$
```

$$bias_shift = (n_{input} + n_{weights}) - n_{biases} \tag{11}$$

$$out_shift = (n_{input} + n_{weights}) - n_{output} \tag{12}$$

PULP-NN API is a similar solution for the RISC-V (RV32IMCXpulp) architecture [64]. The key innovation of PULP-NN is the support for multi-core processing. However, the support of activation functions is more limited - it only supports ReLU. Both APIs allow reliable execution of the inference pass of ANNs with negligible accuracy loss.

2.2.2 Google Federated Learning System

Google is testing a FLS on Android devices through the Gboard application [65]. The base application consists of a keyboard predictor that, based on the current context, presents multiple suggestions that can be picked by the user. When the prediction process is performed it creates a history on-device that will further allow federated learning.

The implementation of FL by Google had to overcome many algorithmic challenges, such as the training algorithm. Typically, ML systems perform training over a large dataset partitioned homogeneously across servers in the cloud, using an optimization algorithm such as SGD. This approach requires high-throughput connections to the training data, which is not feasible when data is distributed across thousands or millions of devices that have associated a latency-response due to communication process, a problem that is worsening as a result of intermittent availability for training.

To face bandwidth and latency limitations, Google developed a federated averaging algorithm, able to train deep networks using 10 to 100 times fewer communications in comparison to a classical federated version of SGD [59]. Through a round-based approach, the training process is performed over an FL population where only some devices are chosen to participate in a specific task. Therefore, a training round is defined by a procedure that consists of three different phases: (i) selection, (ii) configuration, and (iii) reporting [16], and its representation is present in Figure 2.11.

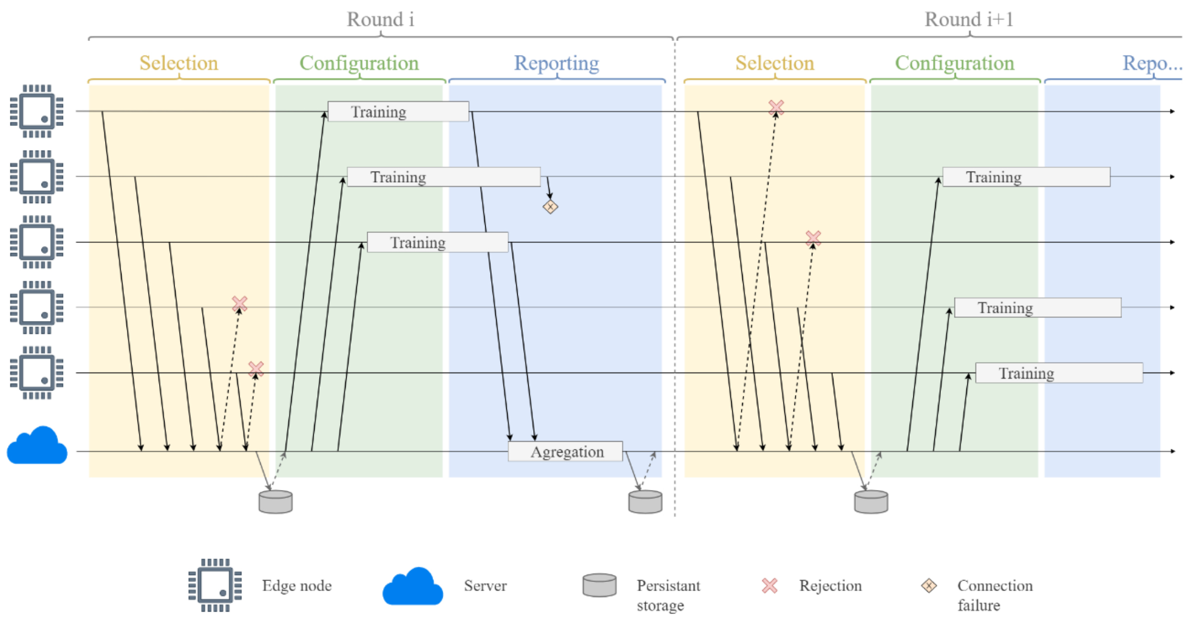


Figure 2.11: FL rounds system protocol

The selection process chooses a portion of the potential devices announcing availability to the server in a specific time window, forming a subset of selected devices that are invited to participate in an FL specific task. A device should only announce the availability to the server when it meets the eligibility requirements. The scheduler must ensure that a device is only selected to participate in the update process if it is in an idle mode, plugged in, and on a free wireless connection [65]. Once that a device is selected, a bi-directional stream is set between the device and the server that allows track liveness and management multi-step communication. After selecting the devices to participate in a round, the server must respond to the remaining devices informing them to reconnect later.

The second phase is the configuration, where the server sends the FL plan and an FL checkpoint with the global model to each of the devices. The selected aggregation mechanism (simple or secure aggregation) is the base of server configuration [16]. After each device receives the FL plan, the training process must be performed, and the results sent back to the server. The final phase of a round, the reporting phase, is where the server waits for devices' responses and aggregates them using an aggregating algorithm [60]. If the training round is completed, the server instructs the device when to reconnect and closes the existing stream. However, if the device does not respond in time the stream is closed and the task is considered as failed. If enough devices complete the task, the round is considered as completed, and the server will update its global model. Otherwise, the round is abandoned [16].

Using computational power present in modern mobile devices allows to compute higher quality updates than simple gradient steps, reducing the number of interactions with the server and, consequently, the required iterations of communication. Nevertheless, the bandwidth of the network infrastructure pops up as a possible limitation to Google FLS. To address this issue, Google uses random rotations and quantization as model compression mechanisms [60].

The model updates require a secure, efficient, reliable, and scalable aggregation mechanism. This is achieved through the usage of a secure aggregation protocol that uses cryptographic techniques, and it is managed by a server that can only decrypt data if hundreds or thousands of users have participated in the model update. The secure aggregation protocol is built over six main cryptography primitives: (i) secret sharing, (ii) key agreement, (iii) authenticated encryption, (iv) pseudorandom generator, (v) signature scheme, and (vi) public key infrastructure [66]. A high-level view of secure aggregation protocol is presented in Figure 2.12.

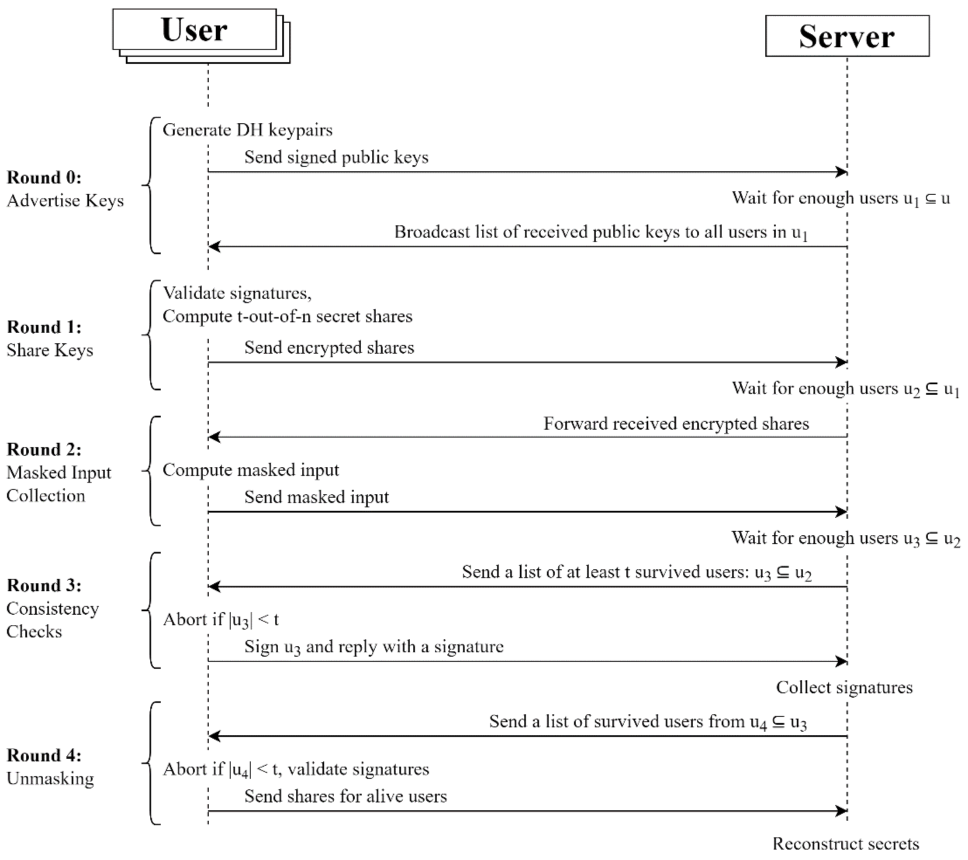

User
Server
Round 0:
Advertise Keys
Generate DH keypairs
Send signed public keys
Wait for enough users $u_1 \subseteq u$
Broadcast list of received public keys to all users in u_1
Round 1:
Share Keys
Validate signatures,
Compute t-out-of-n secret shares
Send encrypted shares
Wait for enough users $u_2 \subseteq u_1$
Round 2:
Masked Input
Collection
Forward received encrypted shares
Compute masked input
Send masked input
Wait for enough users $u_3 \subseteq u_2$
Round 3:
Consistency
Checks
Send a list of at least t survived users: $u_3 \subseteq u_2$
Abort if $|u_3| < t$
Sign u_3 and reply with a signature
Collect signatures
Round 4:
Unmasking
Send a list of survived users from $u_4 \subseteq u_3$
Abort if $|u_4| < t$, validate signatures
Send shares for alive users
Reconstruct secrets

Figure 2.12: High-level overview of secure aggregation protocol used by Google FLS

Secret sharing: Secret sharing relies on Shamir's t-out-of-n [67], which allows users to split a secret *s* into *n* shares. This way, any *t* shares can be used to reconstruct *s*, but any set of at most *t*-1 shares gives any information about *s*. This approach is called a threshold scheme, and an efficient threshold scheme can be very helpful in the management of cryptographic keys. Data protection is achieved through encryption methods; however, encryption key protection requires a different method. The most secure key management scheme keeps the key in a single and protected location. Nevertheless, this procedure is highly unreliable since that misfortune can make the information permanently inaccessible. Despite increasing the possibility of security breaches, a possible solution is to store multiple copies of the key in multiple locations. Using a (k, n) threshold scheme, with $n = 2k - 1$, it can be reached a consistent and robust key management scheme: the original key can be recovered even when $[n/2] = k - 1$ of the *n* pieces are destroyed, but the reconstruction of the key is impossible even when security breaches expose $[n/2] = k - 1$ of the remaining k pieces [67].

Key agreement: Key agreement consists of a set of three algorithms that provide: (i) the production of some public parameters (over which the scheme will be parameterized), (ii) generate a private-public key-pair, and (iii) combine a private key with a public one to obtain a private shared key. The specific scheme used in the secure aggregation protocol is the Diffie-Hellman key agreement [68], composed with a hash function. The main goal of key agreement is to develop systems, such the one presented in Figure 2.13, where two parties can communicate over a public channel with a secure connection. In this scheme, the public file of enciphering keys is protected from external and unauthorized modification. This can be achieved through the nature of the file, avoiding the need for reading protection. Since the file is not frequently modified, the cost of implementation of write protection mechanisms is reduced [68].

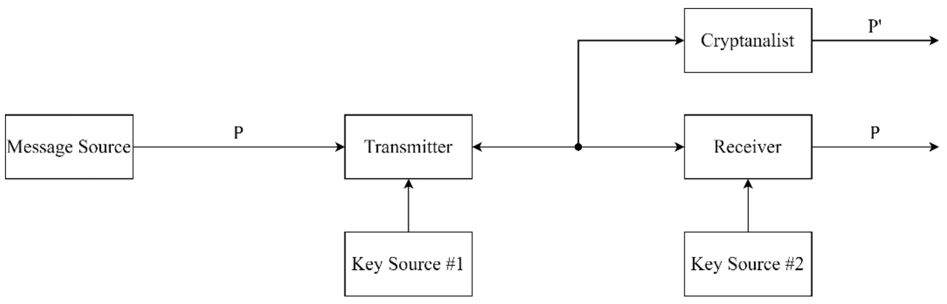


Figure 2.13: Public key system information flow

Authenticated encryption: Authenticated encryption combines confidentiality and integrity guarantees for messages shared between two parties and consists of three algorithms: (i) a key generation algorithm, (ii) an encryption algorithm, and (iii) a decryption algorithm. For security, it requires indistinguishability under a chosen plaintext attack (IND-CPA) and ciphertext integrity (IND-CTXT) [66], [69]. The guarantee is that for any attacker *M* that is given messages encrypted under a randomly sampled key c (where c is unknown to *M*), *M* cannot distinguish between unique encryptions under c of different messages, neither can *M* create new valid ciphertexts with respect to c with meaningful advantage.

Pseudorandom generator: Secure aggregation protocol also requires a pseudorandom generator [70], [71] that takes in a uniformly random seed of a given fixed length and whose output space is the protocol input space. The usage of such mechanism grants that its output is computationally indistinguishable from a uniformly sampled element of the output space, once that the seed is hidden from the analyzer.

Signature scheme: Additionally, the protocol signature scheme relies on a standard unforgeable under chosen message attack (UF-CMA), which means that it is computationally infeasible for an external analyzer to find a new valid message-tag pair after querying *p* pairs of message-tag [72]. The scheme is composed by three algorithms: (i) key generation algorithm, which takes as input the security parameters and outputs a secret key, (ii) the signing algorithm, that takes as input the secret key and a message and outputs a signature, and (iii) the verification algorithm, that takes as input a public key, a message, and a signature, and returns a bit corresponding to validation or not of the signature.

Public key infrastructure: To prevent the server from simulating an arbitrary number of clients it is required the support of public key infrastructure. This mechanism allows users to register identities and sign messages using their identity, which can be verified by other users but not taken off. Secure aggregation protocol demands that each client registers to a public bulletin board during the setup phase. During this setup process each client can only register a self-key, avoiding attacking parties to impersonate honest users [66], [69].

2.2.3 Tensorflow Federated

TensorFlow Federated (TFF) is a ML and decentralized data computation open-source library, developed by Google. This framework allows developers to simulate decentralized computation into the hands of all TensorFlow users, as well as experiment with new algorithms.

Furthermore, base elements of TFF provide utility functions that grant non-learning computations, such as aggregated analysis in decentralized data. TFF splits into two layers [73]: (i) API Federated Learning (FLA) and (ii) API Federated Core (FCA). FLA makes available a set of high-level interfaces that allow developers to implement applications with the incorporation of evaluation mechanisms to measure the FL performance over TensorFlow existing models. Differently, FCA is a set of low-level primitives that allows the expression of a wide range of computations over a decentralized dataset. It is a programming environment for the implementation of distributed computation, such as the FL scenario. TensorFlow targets to reach the major device platforms and pretends to improve the security of sensitive user data through the integration of technologies such as differential privacy and secure aggregation.

2.2.4 Aggregation Mechanisms

In the recent past, multiple solutions have been proposed to enable decentralized learning. One of the main focuses of research on the FL field relies on the development of robust and reliable aggregation mechanisms. In this subsection, it is reviewed the most relevant aggregation mechanism developed to date.

FedAvg: McMahan et al. [59] introduced the concept of FL and proposed FedAvg, a technique for training a shared model across clients. The main goal of this approach is to minimize the global loss by performing a weighted average of the local weights and biases. Parameters are weighted by the size of the client's dataset. FedAvg was developed considering data heterogeneity non independent and identical distributed (non-IID) and the volatile availability of edge devices. FedAvg is the basis of most of the works developed in the FL field.

FedMA: Wang et al. [74] proposed FedMA, an aggregation method that aims to tackle the data and processing power heterogeneity on the edge for convolutional neural networks and long-short term memory models. FedMA builds the shared model in a layer-wise approach. At each training round, FedMA selects the set of edge devices to train a given model layer. The resulting weights and biases are merged by a matched averaging technique. Results show a reduction of communication rounds during the training pass and tolerance to non-IID data partition.

FedProx: FedProx [75] aims to tackle heterogeneity in FLS and it can be represented as a generalization and re-parametrization of FedAvg. The main contribution of FedProx is how it handles stragglers. FedProx considers that edge devices often have different resource constraints and allows variable amounts of work to be performed locally across devices based on their available systems resources. In contrast to FedAvg, FedProx aggregates the partial information returned by stragglers instead of dropping them out.

q-FedAvg: To provide a fairer accuracy distribution over edge devices, Li et al. [76] proposed q-FedAvg. q-FedAvg introduces a penalty factor q to the loss function of FedAvg such that edge devices with higher loss get a higher relative weight q. This leads the aggregation algorithm to improve the accuracy on the worst-performing devices.

Per-FedAvg: Fallah et al. [77] addressed the lack of user personalization in the FedAvg algorithm. Per-FedAvg is a variant of FL which considers that the cloud should return a global ML model that users can easily adapt to their local dataset by performing some additional training steps on their own data. Fallah et al. [77] advocate that this strategy maintains all the benefits of FL while delivering personalized ML models to each user.

Gap Analysis: The different aggregation mechanisms analyzed use FedAvg as their development baseline since it is a simple and effective approach, which does not increase the computational complexity in the deep edge. This approach shows tolerance for real-world data partitions (non-IID). However, this approach does not prevent the definition of a model biased for a small percentage of devices. To address this concern different mechanisms were proposed. FedMA is a method for performing layer-by-layer training, minimizing the computing cost of the training process. Furthermore, it reduces the amount of modification that each device may make to the global model. Following that, the mechanisms FedProx and q-FedAvg were introduced aiming to produce a more generalized model that is equally efficient across all clients. As a result, the model adapts to these devices, resulting in a more accurate model.

However, all mechanisms focus the evaluation on a portion of centralized data. As a result, Per-FedAvg emerges as a technique that aims to produce models that can be optimized for each local dataset. Although all of the approaches try to help with the feasibility of federated training, they all miss the mark on the same point: the impact of undesirable behaviors on the deep edge is not considered. Therefore, decentralization of the training process exposes a window to possible attacks following the currently available aggregation mechanisms. Table 2.4 depicts the reviewed aggregation mechanisms.

Table 2.4: Functionality of aggregation mechanisms

R&D Study	**Non-IID Data**	**Use Personalization**	**Stragglers**	**Dataset Poisoning**	**Train Report Poisoning**
FedAvg [59]	Yes	No	Drop-out	No	No
FedMA [74]	Yes	No	n.d.	No	No
FedProx [75]	Yes	No	Tolerates partial work	No	No
q-FedAvg [76]	Yes	No	n.d.	No	No
Per-FedAvg [77]	Yes	Yes	n.d.	No	No

Chapter 3

System Specification

The previous chapter uncovered the main concepts related to this MSc thesis. In addition, it also outlined the best practices and guidelines for the design of the deep edge and the cloud server systems. Consequently, this chapter specifies the proposed FLS, focusing on the overall system architecture in Section 3.1, the deep edge system in Section 3.2, and the FL server in Section 3.3. This chapter introduces the design of a lightweight training algorithm that targets the architectures of Arm Cortex-M and a reliable centralized aggregation mechanism to replace the FedAvg method.

3.1 System Architecture

The designed system relies on a centralized FL architecture with horizontal data partitioning. As expected in a centralized design, our system architecture relies on two parts: (i) the edge, composed by Arm Cortex-M MCUs, and (ii) the cloud server. The inference and the main part of the training pass are confined to the edge. For the training pass, it is proposed a lightweight version of the SGD (L-SGD), described in Section 3.2. During the training pass, the architecture of the starting ANN remains untouched but its weights and biases are updated to generalize better for the data collected locally. The resulting weights are sent to the central server to be aggregated following a synchronous communication mechanism based on rounds. For weights and biases aggregation, it is proposed R-FedAvg, which is a more reliable implementation of the FedAvg algorithm described in Section 3.3. R-FedAvg is expected to be less vulnerable to data poisoners and stragglers, avoiding the creation of biased models.

After aggregating the weights and biases returned by edge devices, the server evaluates the new global ML model. As there is no user data transferred from the edge to the server, the accuracy of the new ML model is evaluated over a small portion of centralized data. Consequently, the accuracy in some devices can be penalized due to the overall model generalization. To deal with this accuracy loss, an additional training step is introduced after the deployment of the new global ML model in edge devices. This additional training iteration is also performed by L-SGD. Figure 3.1 summarizes the general workflow of the proposed FLS.

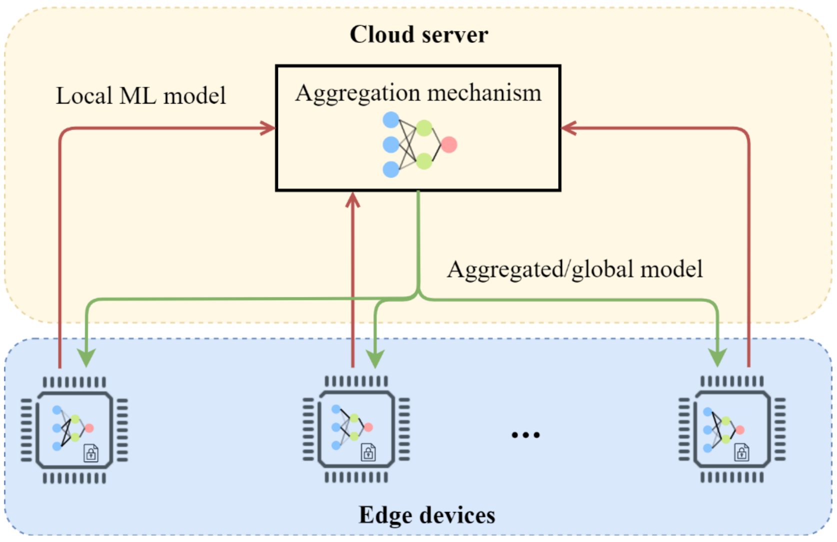


Figure 3.1: FLS overview

3.2 Edge Training

Training an ANN is an exhaustive computation process due to the large amount of data used in the training process. Faster training typically requires more computational resources. The selection of the training algorithm optimizer relies on a trade-off between convergence speed and memory consumption. Adam shows up as the fastest and most efficient optimizer. However, the design of a training algorithm that targets deep edge devices must prioritize the algorithm memory footprint reduction and reduced computational complexity. From this perspective, the SGD mechanism shows up as the best solution. Although the increased latency is introduced in the learning process, the lower complexity and the memory requirements scalability fits the hardware constraints associated with the deep edge devices. Consequently, the designed algorithm relies on the SGD mechanism.

3.2.1 Lightweight SGD (L-SGD)

L-SGD is a lightweight implementation of SGD, optimized for low-memory footprint and latency, while guaranteeing negligible accuracy loss. To evaluate the feasibility of fully-quantized training, it was developed three versions of L-SGD: (i) one that operates over 32-bit floating-point data, (ii) one that operates over 8-bit integer data, and (iii) another that operates over 16-bit integer data. As detailed in Section 4.1, quantized training, suing 8-bit integer data, allows a 4x reduction in memory footprint and a 2x faster training process. However, results have shown the quantization error can compromise the training as it may disrupt severe accuracy losses. This arises as a consequence of the fast saturation of weights and bias during the first iterations of the training process.

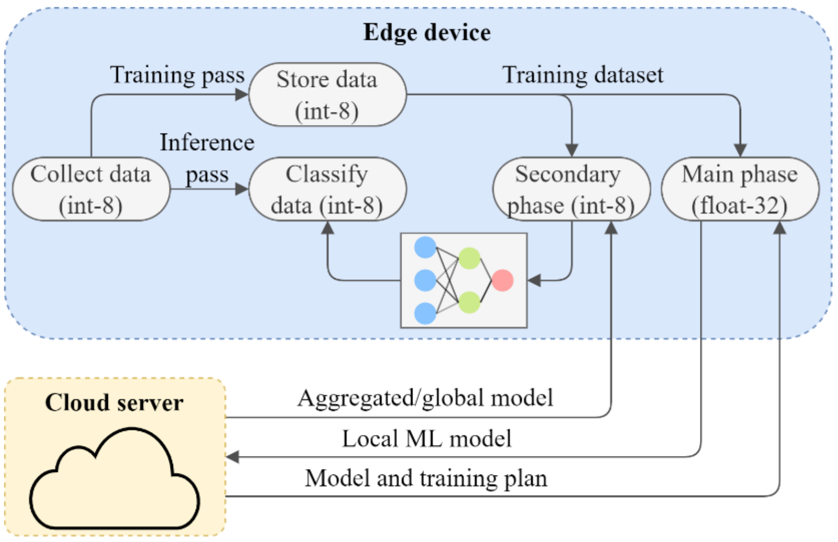


Figure 3.2: Hybrid training workflow

In an attempt to reduce the quantization error effect disrupted by 8-bit integer data, the bit-width of the quantized L-SGD operands was increased to 16-bit. As detailed in Section 4.1.4, increasing data precision to 16-bit reduces the quantization error effect in intermediate calculus of L-SGD. Nevertheless, the effect in accuracy loss is not substantial. Aligned with the increased complexity and latency, the 16-bit integer approach underperforms the 8-bit integer method. Nevertheless, quantized training can be accurate for the later training iterations when the loss starts to converge to zero. In this phase, the magnitude of updates is so small that it is very unlikely for the weights and biases to be saturated. Given the background, this work proposes a hybrid mechanism and divides the training on the edge into two phases: (i) main and (ii) secondary. Both phases consider the same dataset as input.

The main training phase performs on 32-bit floating-point data and starts whenever the cloud server sends a training plan to the edge. The main phase is responsible for generating the ANN that is further sent to the cloud server as a local model update. As the ANN is stored in int-8 in the MCU, the ANN is dequantized to 32-bit floating-point data before the training itself. When L-SGD finishes the training, the ANN is quantized back again to int-8 data and sent to the cloud server, which will aggregate the weights and biases returned by each edge device.

The secondary phase starts whenever a new ANN is received on the edge. This phase is performed in int-8 data and can be seen as a small update to tweak the model to local dataset. To prevent overfitting, it is used a very low learning rate (0.1% of the learning rate used in the main phase). The ANN is assumed to be quantized according to the policy followed by CMSIS-NN: static quantization performed layer-by-layer with a fixed bit-width of 8-bits. L-SGD (int-8) was selected over L-SGD (int-16) for the secondary training phase as the higher resolution returned by the later version is not significant at this training phase. Furthermore, as shown in Section 4.1.4, L-SGD (16-bit) induces higher latency and memory footprint than the equivalent 8-bit version, neglecting the ultimate purpose of quantization. Figure 3.2 depicts an overview of the hybrid training mechanism. For both main and secondary phases, this work optimizes the loss according to a strategy hereinafter referred to as node delta optimization.

3.2.1.1 Node Delta Optimization

Classic SGD requires the allocation of a memory block that doubles the size of the ANN parameters. This arises as a consequence of the error propagation effect. As detailed in Section 2.1.4.2, the update of a given parameter in the backward pass depends on values returned in the previous iteration for parameters in other layers. L-SGD reduces the memory footprint and the computation overhead of SGD through an optimization that we define as node delta.

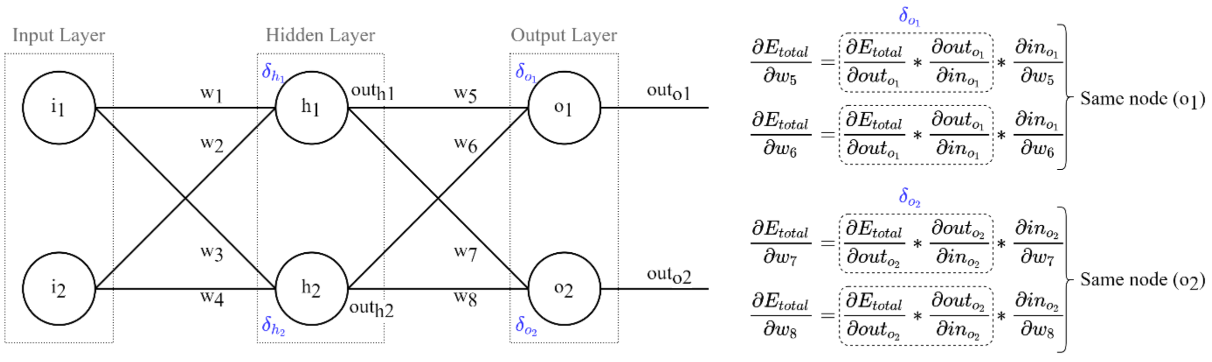

Input Layer
Hidden Layer
Output Layer
i_1
i_2
h_1
h_2
o_1
o_2
w_1
w_2
w_3
w_4
w_5
w_6
w_7
w_8
out_{h1}
out_{h2}
out_{o1}
out_{o2}
δ_{h_1}
δ_{h_2}
δ_{o_1}
δ_{o_2}
$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial in_{o_1}} * \frac{\partial in_{o_1}}{\partial w_5}$
$\frac{\partial E_{total}}{\partial w_6} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial in_{o_1}} * \frac{\partial in_{o_1}}{\partial w_6}$
Same node (o_1)
$\frac{\partial E_{total}}{\partial w_7} = \frac{\partial E_{total}}{\partial out_{o_2}} * \frac{\partial out_{o_2}}{\partial in_{o_2}} * \frac{\partial in_{o_2}}{\partial w_7}$
$\frac{\partial E_{total}}{\partial w_8} = \frac{\partial E_{total}}{\partial out_{o_2}} * \frac{\partial out_{o_2}}{\partial in_{o_2}} * \frac{\partial in_{o_2}}{\partial w_8}$
Same node (o_2)

Figure 3.3: Node delta parameter in the chain rule of SGD

As shown in Figure 3.3, parameters that share the same output neuron also share some terms in the chain rule. These are the only terms that depend on the initial values of other weights in the ANN. L-SGD allows a memory footprint reduction of SGD by merging these terms in a single parameter, hereinafter referred to as node delta. This parameter is a characteristic of each neuron in the ANN and its value reflects the error effect propagated from the final model output to the neuron input. Instead of saving all weights of the previous training iteration, L-SGD only saves a parameter for each neuron. Considering that the number of neurons is less than the number of weights and these difference increases as layers get wider, L-SGD has the potential to considerably reduce the memory footprint of SGD. The calculus of node delta depends on the layer to which the neuron belongs.

Output layer: As observed in Figure 3.3 for the output layer, the chain rule for weights with the same output neurons has two terms in common: (i) the loss function partial derivative and (ii) the activation function partial derivative. The multiplication of these two terms composes the delta for a given neuron in the output layer. As differentiation requires high computational power, the training algorithm is designed to not support the differentiation process itself, but to reduce the scope of functions subject to differentiation and implement the derivative in code. For loss functions, this framework integrates two functions that cover most of the classification problems: (i) binary cross-entropy and (ii) cross-entropy. Both loss functions and the correspondent partial derivative are outlined in Table 3.1. Regarding the activation functions, the same functions of CMSIS-NN are supported: (i) ReLU, (ii) sigmoid, and (iii) tanH. The activation functions and the correspondent partial derivative functions are described in

Table 3.2.

Table 3.1: Loss functions supported by L-SGD. Model prediction is represented by $(\hat{y})$ and the true label by (y).

Loss function	**Equation**
Binary cross entropy (BCE)	$-\sum_{i=1}^{N}\left(y_i * log(\hat{y}_i)\right) + (1 - y_i) * log(1 - \hat{y}_i)$
BCE partial derivative	$\frac{1 - y_i}{1 - \hat{y}_i} - \frac{y_i}{\hat{y}_i}$
Cross entropy (CE)	$-\sum_{i=1}^{N}\left(y_i * log(\hat{y}_i)\right)$
CE partial derivative	$\frac{y_i}{\hat{y}_i}$

Table 3.2: Activation functions supported by L-SGD. Neuron's input is represented as x and the output as $f(x)$.

Activation function	**Equation**
ReLU	$\begin{cases} 0 \; if \; x \leq 0 \\ x \; if \; x > 0 \end{cases}$
ReLU partial derivative	$\begin{cases} 0 iff(x) \leq 0 \\ 1 iff(x) > 0 \end{cases}$
Sigmoid	$\frac{1}{1 + e^{-x}}$
Sigmoid partial derivative	$f(x) * (1 - f(x)$
TanH	$\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
TanH partial derivative	$1 - f(x)^2$

Hidden and input layers: As detailed in Equation 13, the calculus of the node delta for a hidden or input layer splits into two main terms. The first term (δ'_{h1}) represents the propagation of the error effect from the ANN output to the output of the neuron. As the error is transmitted through the weight set to each connection between neurons, the first term is a weighted sum of the node deltas belonging to the neurons in the following layers (Figure 3.4). The second term ($\frac{\partial A(out_{o1})}{\partial out_{o_1}}$) represents the error propagated from the neuron output till its input and is calculated as the derivative of the activation function. The derivative of the activation function follows the same policy described in the calculus of node delta for neurons in the output layer.

$$\delta_{h1} = \delta'_{h1} * \frac{\partial A(out_{o1})}{\partial out_{o_1}} \qquad (13)$$
$$\delta'_{h1} = \delta_{o1} * w_5 + \delta_{o2} * w_7$$

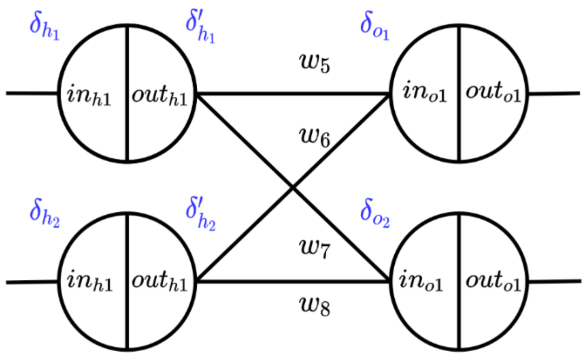


Figure 3.4: Calculus of node delta in hidden and input layers

3.3 Federated Learning Server

As specified in Section 3.1, the centralized server is in charge of the aggregation process. Aiming to balance the computation cost between the server and the deep edge the server integrates the mechanisms to tackle the decentralization challenges. Besides reducing the local overload, this design decision adds a new robustness layer to local poisoning or communication security faults (even if there is data manipulation in the communication channel, the server can filter this effect). In this subsection, it is presented the design of the FL sever by covering three main topics: (i) decentralization challenges, (ii) aggregation mechanisms vulnerabilities, and (iii) R-FedAvg aggregation mechanism design.

3.3.1 Decentralization Challenges

The decentralization of the training process on ML addresses new challenges. Although multiple works proposed different solutions to speed up the convergence of the learning process, there are still some real-world threats to FLS. The data partition is a must-consider variable of decentralized learning. Currently, most aggregation mechanisms reveal a weakness in balancing the training process to generate a model that fits every user's application without local accuracy loss [78]. Furthermore, the dataset of each device can be noisy or contain non-representative samples that penalize the overall model. In addition, communication problems or local hardware specifications can take some devices to abort the training round.

Data partition: Since the training performs among the edge devices, the training data is partitioned over different users. Ideally, a dataset used in the training process must be balanced, i.e., that each class is represented by the same number of samples in every edge device. Furthermore, datasets must have the same size in different devices. This specification referred to as independent and identical distributed (IID) data [60], is the optimal training condition. However, the development of a FLS must consider real-world conditions. A common situation is the unbalanced data distribution over devices, where different devices vary by orders of magnitude in the number of training samples they hold [60], [61]. An even more realistic setting than the unbalanced data partition is the Non-IID. In Non-IID the data partition is unbalanced and the number of samples per class available in a given device is heterogeneous. Figure 3.5 depicts the different data partitioning strategies.

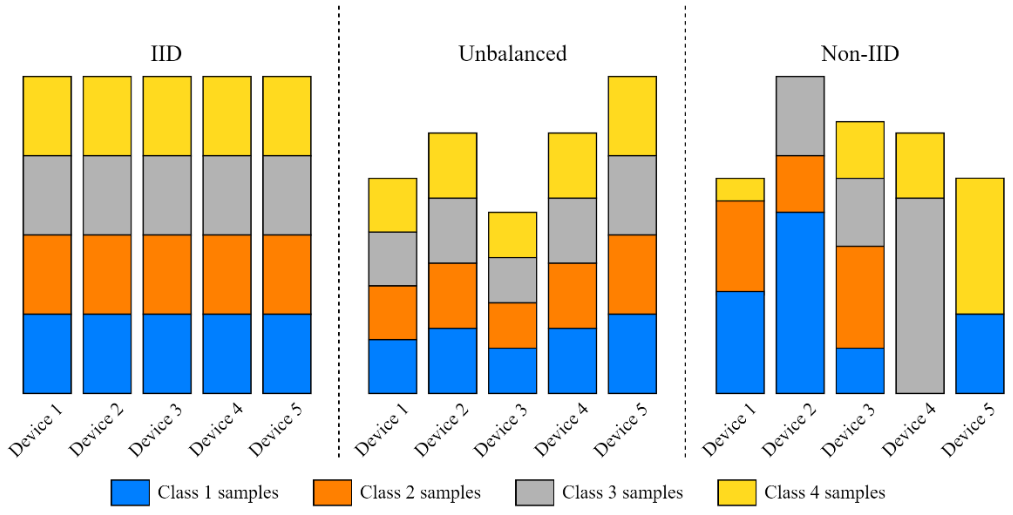


Figure 3.5: Data partitions: IID vs. Unbalanced vs. Non-IID

Data poisoners: A struggle of partitioning the training data over the deep edge devices is the fact that there are no mechanisms to evaluate the data quality. Noise is an unpredictable and undesired factor that is injected into data, which penalizes the learning process [61]. An extreme scenario is premeditated data manipulation to subvert the ML model accuracy. A common approach to introduce noise on data is by adding Gaussian noise [61]. Feeding meaningless data as input produces high variance on model parameters, which affects the general model accuracy.

Stragglers: Since the updated parameters are produced in different devices, the convergence speed of the training process depends directly on the returned reports. Stragglers are edge devices that penalize the convergence of the optimization algorithm during the training process [57]. More specifically, stragglers are edge devices that abort the training rounds due to communication loss with the server or by sending faulty training reports.

3.3.2 FedAvg Vulnerabilities

FedAvg was introduced in [59] to address the aggregation of ML model parameters in a centralized FLS. In this aggregation algorithm, the training report of each edge device is composed of the updated ML model parameters and the local dataset size. As the dataset is a fundamental part of the training process, FedAvg considers that a fair model update must consider the distribution of data over the edge. Therefore, the update of a given parameter is performed as an average of the value returned by each edge device for that parameter, weighted by the local dataset size. The fraction of selected devices to participate in each training round is set by the server through a hyperparameter. FedAvg considers that every edge device must follow the same training plan with the same hyperparameters.

Data poisoning: The main flaw of already-available aggregation mechanisms is related to the training report. As a fair aggregation is highly dependent on the dataset size reported by each edge device, this parameter becomes the preferred attack surface in a FLS based on FedAvg. To not break the founding principles of FL, the local dataset is never transferred to the cloud server. Therefore, FedAvg cannot know when an edge device is cheating about its local dataset size. In FedAvg, a malicious edge device can subvert the global model update by simply reporting a bigger local dataset size than it is. This leads FedAvg to give a higher weight to parameters returned by malicious edge devices, which may disrupt the creation of biased ML models. Ultimately, this attack can subvert the entire FLS. Aggregation algorithms developed to date have never considered data poisoning as a priority (Table 2.4).

User personalization: Besides system reliability, the available mechanisms fail in evaluating the trained model. The FedAvg variations aim to tackle data heterogeneity consequence of data partitions. However, the evaluation typically relies on centralized data. Furthermore, R-FedAvg is the only alternative to Per-FedAvg in terms of user personalization. Except for Per-FedAvg, all FedAvg-based algorithms fail on the issue of user personalization. By user personalization it is meant the customization of the global ML model to the local data of the edge. This happens because the global model that is sent to the edge never undergoes a tuning to the local dataset before being used in inference. Per-FedAvg is the only one that solves this situation by including additional training steps in the edge after the deployment of the global model.

3.3.3 Reliable-FedAvg (R-FedAvg)

R-FedAvg is a more reliable implementation of the FedAvg algorithm, which pretends to tackle the data poisoning and user personalization issues identified in Section 3.3.2.

Data poisoning: R-FedAvg softens the impact of malicious edge devices by employing a mechanism to detect local model updates whose parameters diverge considerably from the parameters returned by the remaining edge devices and from the initial ML model itself. The parameters returned by devices that fall in this category are discarded. For outlier detection, it is employed an unsupervised learning technique at each training round: hierarchical clustering algorithm [32]. The first point to be added to the algorithm is the one that represents current model parameters. Thus, every point that belongs to the same cluster as the initial point is an inlier. This mechanism ensures system robustness even to synchronized attacks, not assured by density-based algorithms [36]. This method does not require the definition of a specific number of clusters. Instead, it relies on a distance threshold, which must be tuned for each FL setting. The algorithm clusters each sample to the initial point if the distance between data is lower or equal to the threshold value. If the distance is higher than the threshold, the respective parameters are discarded and the device is identified as malicious. To tune this hyper-parameter, it must be considered that for a classic training process, parameter updates tend to be much lower in magnitude in later stages of the training. Similarly, the distance between the local model parameters and the current global model is expected to be small.

User personalization: Some device's performance can be penalized due to the overall model generalization of FedAvg. To deal with this performance loss, R-FedAvg introduces an additional training step after the federated averaging algorithm. The server sends the updated model to the devices when the aggregation process is completed. Then, new local training is performed with minor model changes. This second iteration allows the global model to tweak to edge local data, improving the local's performance. Ensuring minor changes on this local model update is essential to avoid local model overfitting. Therefore, this is achievable using a significantly lower learning rate (e.g., using a local learning rate a hundred times lower than the value used on the training round process). The parameter λ represents the learning rate reduction factor and is introduced to control the model parameters adjustment. For the global model training rounds, it's set to a unit value, and for the local model tweak, it's set to a tunable value between zero and one. The full process is described in Algorithm 2.

Algorithm 2: R-FedAvg. The K clients are indexed by k; β is the local minibatch size, E is the number of local epochs, η is the learning rate, and λ is the learning rate reduction factor

```
1:   Server executes:
2:   Initialize $w_0$
3:   for each round $t = 1,2,...$ do
4:       $m \leftarrow max(C \cdot K)$
5:       $S_t$ random set of $m$ clients
6:       for each client $k \in S_t$ in parallel do
7:           $w_{t+1}^k \leftarrow ClientUpdate(E, w_t, \eta)$
8:       end for
9:       Validate models updates
10:      $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$
11:      for each client $k \in S_t$ in parallel do
12:          $w_{t+1}^k \leftarrow ClientUpdate(E, w_t, \eta, \lambda)$
13:      end for
14   end for
```

3.3.4 Round Participants Selection

Each training round performs over a set of devices. For instance, the number of available devices to participate in a training round can exceed the minimum number of devices required to complete the round. This scenario requires a selection mechanism that allows a fair and distributed selection. The goal is that every device has the same chance to participate in the training process as the remaining ones, not favoring any parties. The most straightforward approach is to implement a simple random selection. This way, from a given population, a fraction is randomly selected. This approach avoids bias and theoretically gives the same chance to each device to participate in a training round. However, there are some problems related to sample selection beyond bias. As covered in the previous subsection, data partition depends directly on each device's usage. This way, a simple random selection is not representative of the entire population.

The selection process is the first step to reduce the unbalanced data partition effect. The more random the training data is, the more generalistic the final model will be. Furthermore, there are two solutions for fair device selection: (i) stratified sampling and (ii) clustered sampling. Stratified sampling relies on selecting devices that represent a group or a sub-population. The main goal of this approach is to select different parties that represent each of the model class predictions. However, this approach requires that the server have some information about each local dataset. Even that this avoids raw data exposure, some representative information needs to be shared. This way, clustered sampling comes up as an alternative. This approach selects random devices from clusters. In a real-world scenario, this algorithm allows the server to choose different users to participate in a training round based on the device's location or time zone. Different random selection approaches are presented in Figure 3.6. The FLS developed in this framework prioritizes users that performed fewer training steps. To accomplish so, a table is created wherein the entries are the devices and the number of performed training steps associated with each device. If all devices have been selected the same number of times, the selection is random.

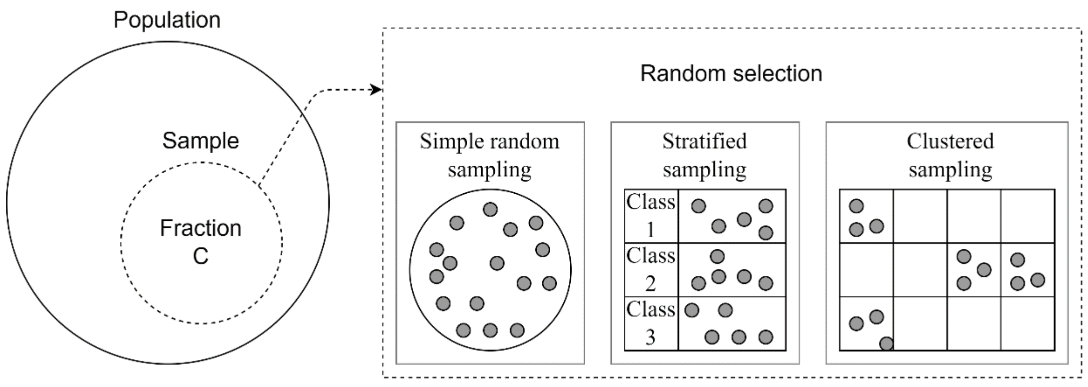


Figure 3.6: Selection mechanisms

3.3.5 Outlier Detection Mechanism

Unsupervised learning methods allow the detection of anomalies during each training round. Outlier detection performs through the implementation of a hierarchical clustering algorithm. Rather than defining the number of clusters as in the k-means approach, this mechanism allows the evaluation of different data points distances. Furthermore, it ensures system robustness even to synchronized attacks, not assured by density-based approaches. The algorithm clusters each sample to the initial point if the distance between data is lower or equal to the threshold value.

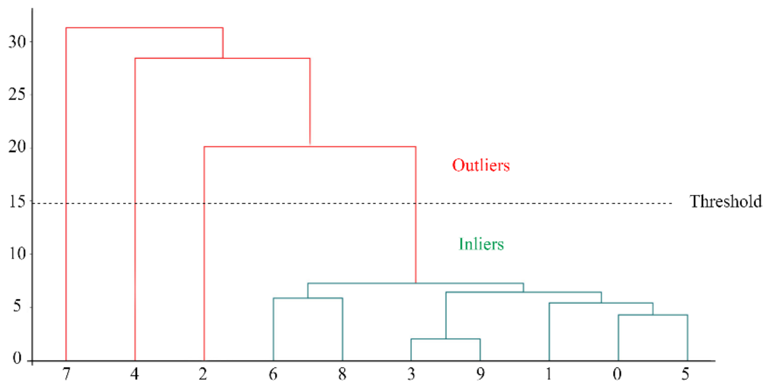


Figure 3.7: Clustering process dendrogram

The clustering starting point is the current global model. The distance between the current model state and each participant's model is used as a selection criterion. The received model updates that are not too different from the current model are seen as helpful to the global model update. Moreover, such updates are classified as inliers. Devices that are not part of this cluster, on the other hand, are considered outliers. The outlier detection system enables the detection of malicious devices and, as a result, their removal. The procedure of detecting outliers is depicted in Figure 3.7.

Since threshold definition is a critical point to the deployment of the FLS, there is a demand to define the threshold value. The threshold value is a hyperparameter that demands a tuning process and represents the model change tolerance in each training round. However, the training process on the server before the federated training deployment can be a starting point for this hyperparameter tuning. Empirical and experimental results have shown that averaging the models distance values on the convergence state and doubling the result value is a good start point for the threshold value tuning, covering the federated training updates produced by different valid parties. The procedure of defining the threshold value is depicted in Figure 3.8.

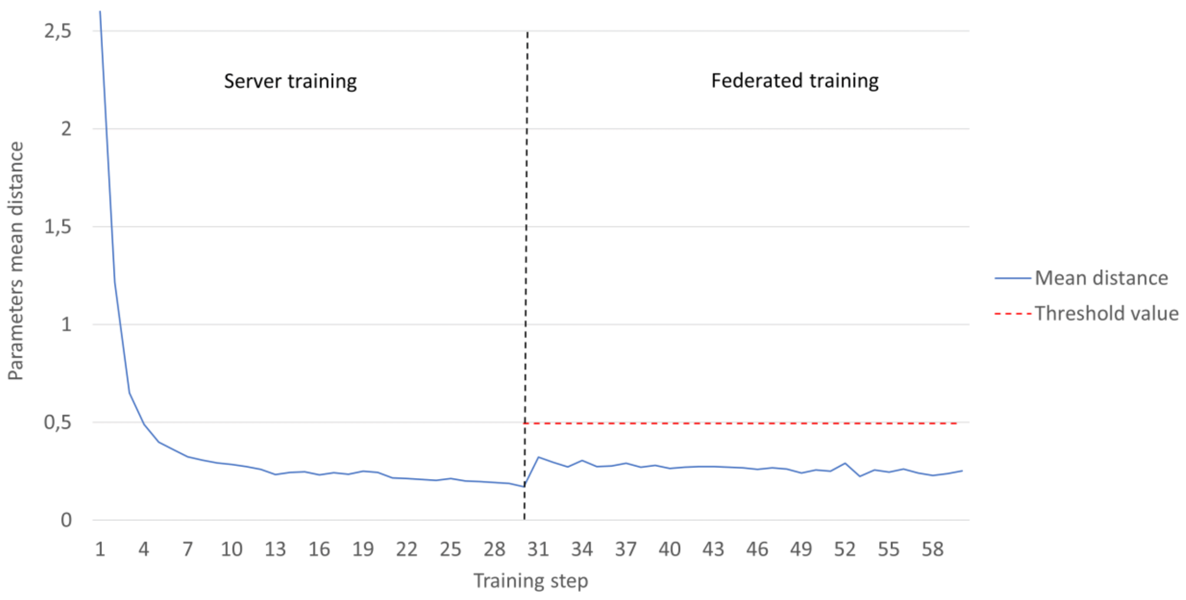


Figure 3.8: Outlier detection threshold definition

Chapter 4

Experimental Results

Previous chapters outline the design stage of the local training algorithm and the FLS central server proposed in this MSc thesis. This chapter aims to present the experimental results of the developed system validation process. The chapter splits into three main subsections: (i) L-SGD results and (ii) R-FedAvg results. The first outline the evaluation metrics used in this chapter. The second subsection presents the experimental results of the L-SGD algorithm. Finally, the third subsection focuses on decentralized learning and presents the evaluation of R-FedAvg in the FL environment.

4.1 L-SGD

The evaluation of L-SGD is performed in terms of (i) accuracy, (ii) latency, and (iii) memory footprint. As this work focus on classification applications, our main metric to evaluate a model relies on the prediction accuracy, which can be obtained through Equation 14.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions\ made} \tag{14}$$

Firstly, its presented a comparison of the floating-point version of L-SGD against the classic SGD implemented by TensorFlow and running on an AMD Ryzen 7 3700X. Then, the quantized and floating-point versions of L-SGD are compared on an STM32F767-ZI (Cortex-M7 @ 216MHz). For the tests, it was borrowed a dataset from [4], which represents the cognitive distraction state of casual drivers under different driving scenarios, and built an ANN with the following architecture:

- input layer: 6 nodes
- hidden layer: 40 nodes with sigmoid activation;
- hidden layer: 32 nodes with sigmoid activation;
- output layer: 1 node with softmax activation.

For both test scenarios previously described, it was performed two training sessions. The first is common to both test scenarios and was performed on the cloud server, using TensorFlow. A portion

(70%) of the dataset is used for training the ANN previously defined along 100 epochs with a batch size of 10. The floating-point version of the ANN achieved an accuracy of 83.02%, while the quantized version achieved an accuracy of 82.58%.

4.1.1 L-SGD (float-32) vs. SGD

To compare the floating-point version of L-SGD against the classic SGD, the floating-point ANN from the first training session was re-trained on an AMD Ryzen 7 3700X using both algorithms. The remaining (30%) of the dataset is used to re-train the ANN along 100 epochs, with a batch size of 1, and no data shuffling. Results are detailed in Table 4.1.

Table 4.1: L-SGD (float-32) vs. SGD: accuracy, latency, and memory footprint

Algorithm	Accuracy (%)	Latency (s/epoch)	Memory footprint (bytes)
SGD	91.94	2.5290	6792
L-SGD (float-32)	91.82	0.1546	5860

Accuracy: The maximum accuracy is registered for the classic SGD implemented by TensorFlow. Nevertheless, the accuracy loss of L-SGD is only 0.12%. This is not a direct consequence of the node delta optimization, but an effect of the different approximations made in the mathematical calculations performed during the forward and backward passes of the training algorithms.

Latency: The floating-point L-SGD is 16.35x faster than SGD to re-train the ANN previously described. This is a result of the node delta optimization and of the strategy used to differentiate the loss and activation functions in the chain rule for each weight. As observed in Figure 3.3, the node delta optimization leads to one less multiplication in the chain rule of weights belonging to the output layer. This difference is increased as the backward pass approaches the input layer, as the optimization delivered by the node delta parameter is even bigger in hidden and input layers.

Memory footprint: For the ANN previously detailed, L-SGD requires 86.28% of the memory required by SGD. As ANNs get wider, the number of weights increases much faster than the number of neurons. Considering that SGD has a memory footprint that doubles the weight size and L-SGD a memory footprint that equals the weight size plus the neuron size, L-SGD has the potential to drastically reduce the memory footprint in the training of large ANNs. To prove this effect, the floating-point L-SGD and SGD are compared in the training of three more ANNs, with more and larger hidden layers. As detailed in Figure 4.1 the higher the number of model parameters, the higher the reduction on memory footprint. For the largest model, L-SDG only requires 38.00% of the memory required by SGD.

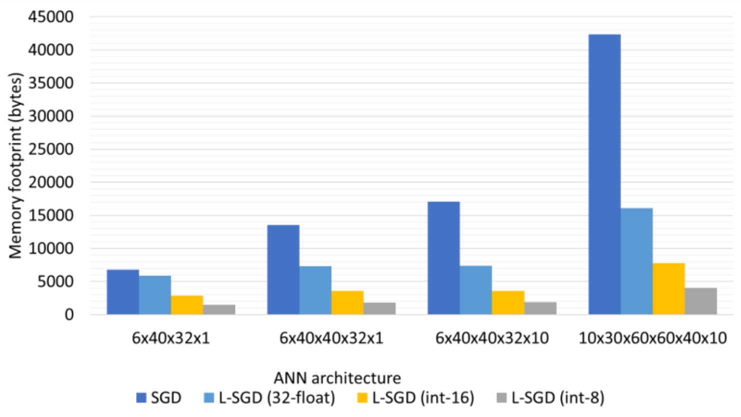


Figure 4.1: Memory footprint test: SGD vs. L-SGD (float-32 vs. int-8 vs. int-16)

4.1.2 Q-format Update Under CMSIS-NN Policy

The forward pass follows the CMSIS-NN quantization policy and is based on the previously defined q-format. Additionally, changes in the model require the update of these previously defined q-formats. On the target systems, updating the quantization formats of the weights and biases is a simple but time-consuming process. However, updating each layer's input and output q-formats is a procedure that cannot be performed on the deep edge. This update is impractical due to the requirement for a large amount of memory. The forward pass process becomes obsolete since each layer's input and output quantization q-formats are not updated. Instead of the prediction error being caused exclusively by incorrect model parameter settings, it also results from incorrect q-format settings.

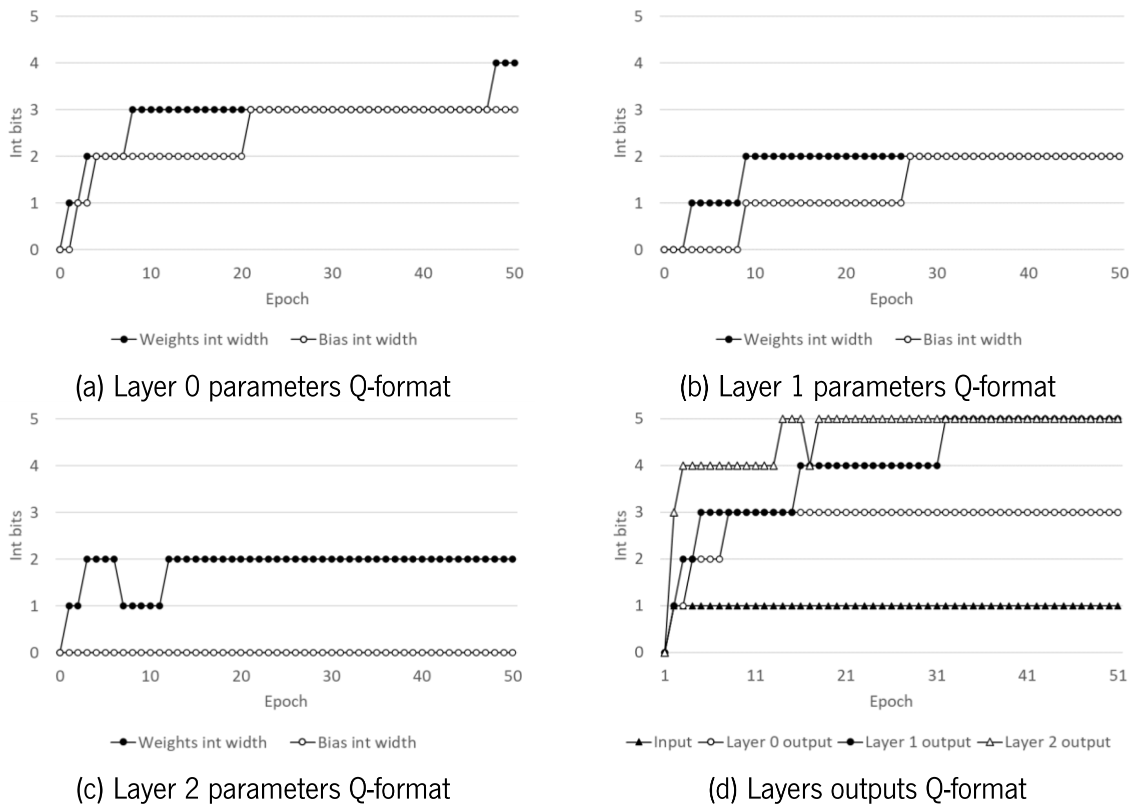


Figure 4.2: Q-format evolution

As a result, training becomes redundant, and model accuracy decreases rather than improves. However, experimental results show that as the training process advances, the updating of quantization formats tends to stall. The model accuracy convergence leads to progressively smaller model modifications. Figure 4.2 depicts the evolution of the model parameters' q-formats and the inputs and outputs of each layer. It is then possible to analyze that from epoch 30 onwards there are practically no changes in q-formats. Reaching the accuracy convergence allows the implementation of the quantized training. As a result, the version of L-SGD (int-8) can be used to fine-tune the model to the local datasets.

4.1.3 L-SGD (float-32) vs. L-SGD (int-8)

To compare the float-32 and int-8 versions of L-SGD, the quantized ANN from the first training session is deployed to an STM32F767-ZI MCU. Then, there is a re-train using both versions of L-SGD under the same conditions of the first test scenario. Results are detailed in Table 4.2.

Table 4.2: L-SGD (float-32) vs. L-SGD (int-8): accuracy, latency, and memory footprint

Algorithm	Accuracy (%)	Latency (s/epoch)	Memory footprint (bytes)
L-SGD (int-8)	75.50	16.36	1465
L-SGD (float-32)	91.82	32.28	5860

Accuracy: As detailed in Table 4.2, the int-8 version of L-SGD induces significant accuracy loss when compared to the floating-point version. While the floating-point version increases the accuracy of the base ANN by 8.8%, the quantized version decreases it by 7.52%, neglecting the purpose of model re-training in FL. This is a consequence of the fast saturation of weights and biases during the first iterations of the quantized training. One solution to this problem would be dynamic quantization. However, this would impose tremendous computation overhead and neglect the purpose of quantized training. Nevertheless, quantized training is a solution for the later training iterations.

Since gradient steps tend to be lower at this point, the magnitude of the update reduces, which avoids the saturation threat. To prove this point, the quantized version of L-SGD is compared against the floating-point version to tweak a global ANN returned by our FLS to the local data distribution. Results are detailed in Table 4.3. The quantized version of L-SGD even outperforms the floating-point version. The lower accuracy of floating-point L-SGD is a consequence of the quantization error. As ANNs received in the edge are quantized, they need to be dequantized before the training process and quantized again when training finishes.

Table 4.3: L-SGD (float-32 vs. int-8) for small updates in quantized ANNs

	Accuracy (%) (float-32)	**Accuracy (%) (int-8)**
Baseline	91.82	90.96
L-SGD (float-32)	91.17	91.95
L-SGD (int-8)	93.39	92.79

Latency: The quantized version of L-SGD is 1.97x faster than the floating-point version. Although the quantized version tends to be always faster, the actual speedup is highly dependent on the target platform. The results were extracted for an STM32F767-ZI MCU, which features an FPU. For an MCU with no FPU, this speedup is expected to be even higher, as floating-point L-SGD tends to be slower.

Memory footprint: The quantized version of L-SGD requires only 25% of the memory used by the floating-point version. As shown in figure Figure 4.1, this high memory saving is independent of the ANN architecture. This characteristic makes the quantized L-SGD a very interesting option to perform small tweaks on ANNs in applications with very limited memory resources.

4.1.4 L-SGD (int-8) vs. L-SGD (int-16)

To compare the int-8 and int-16 versions of L-SGD, the quantized ANN from the first training session was deployed to an STM32F767-ZI MCU and retrained using both versions of L-SGD. The re-train was performed along 100 epochs, with a batch size of 1, and no data shuffling. Training an ML model relies on multiple small steps. Taking large model parameters changes in each gradient can take the learning process to a performance loss since the minimum loss is unreachable. This way, a learning rate is introduced in the training process to reduce each parameter update variance. However, quantization errors can override the learning update and consequently discard the learning step. As a consequence, it was developed and compared two versions of L-SGD that operates over quantized data: (i) L-SGD (int-8) and (ii) L-SGD (int-16). Results are detailed in Table 4.4.

Table 4.4: L-SGD (int-8) vs. L-SGD (int-16): accuracy, latency, and memory footprint

Algorithm	**Accuracy (%)**	**Latency (s/epoch)**	**Memory footprint (bytes)**
L-SGD (int-8)	75.50	16.36	1465
L-SGD (int-16)	75.89	28.23	2857

Accuracy: The extension of 8-bit to 16-bit data allows an error reduction on the intermediate computations. Table 4.5 and

Table 4.6 show that increasing the precision on the backward pass reduces the error in the derivative of the loss and activation functions, when considering the equivalent floating-point values as reference. This is a consequence of extending the data range which softens the saturation effect introduced by the quantization process. The Q-format of the weights and bias are updated at each training iteration. However, the q-format of the layers' input and output remains constant, which induces errors on forward pass. Consequently, the accuracy loss of the 16-bit integer data method is similar to the 8-bit integer data method (Table 4.4).

Latency: The 16-bit integer version of L-SGD requires a data extension operation, leading to a computation overhead in the training process. As expectable, the latency of this version is higher than the 8-bit version. Experimental results (Table 4.4) show that L-SGD(int-16) version is 1.73x slower than the L-SGD (int-8).

Memory footprint: The data extension introduced in the 16-bit integer version leads to an expansion of the memory footprint. This mechanism requires 1.95x of the memory allocation of the 8-bit integer version (Table 4.4). The memory footprint required by L-SGD (int-16) for training different ANN architectures is detailed in Figure 4.1.

Table 4.5: Loss function gradient computation analysis

	Mean error		**Mean error (%)**	
Loss function	**Int-8**	**Int-16**	**Int-8**	**Int-16**
MSE	0,0086	0,0000	4,29%	0,00%
CE	2,2604	0,0003	57,91%	0,06%
BCE	2,8910	0,0008	68,50%	0,03%

Table 4.6: Activation function gradient computation analysis

	Mean error		**Mean error (%)**	
Activation function	**Int-8**	**Int-16**	**Int-8**	**Int-16**
ReLU	0,0777	0,0000	7,77%	0,00%
Sigmoid	0,0144	0,0009	13,34%	0,75%
TanH	0,0195	0,0006	7,06%	0,23%

4.2 R-FedAvg

The R-FedAvg evaluation performs on a synthetic federated scenario, in which the dataset splits over 100 edge devices. The trained ANN (architecture defined in Section 4.1) is evaluated under (i) different data partition schemes and (ii) data perturbation.

4.2.1 Data Partition

The R-FedAvg is evaluated under four data partition schemes: (i) IID, (ii) unbalanced, (iii) Non-IID Fair, and (iv) Non-IID Random. IID refers to the optimal data partition scheme - every user has the same local dataset size and each class is equally represented. An unbalanced scheme considers different amounts of data in each device. The Non-IID schemes match the most realistic data partition - the dataset size differs among devices and different classes are not equally represented in each device. Non-IID fair ensures that all edge devices are called the same number of times during training, while Non-IID random not. For each data partition, the simulation performed over 2000 rounds and 10% of the dataset was used on the server to evaluate the global model accuracy. The remaining 90% of the dataset was partitioned over 100 edge devices according to the four described partition schemes.

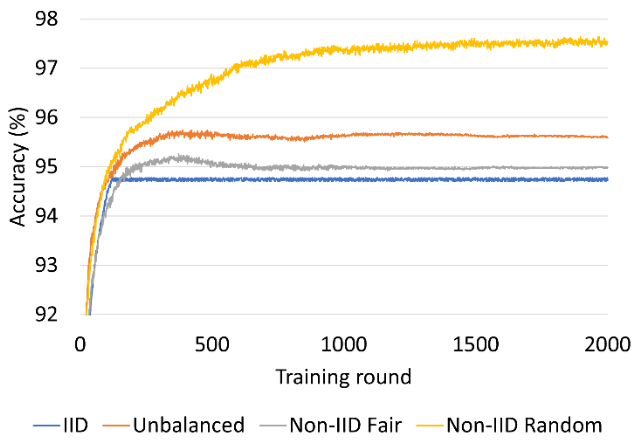
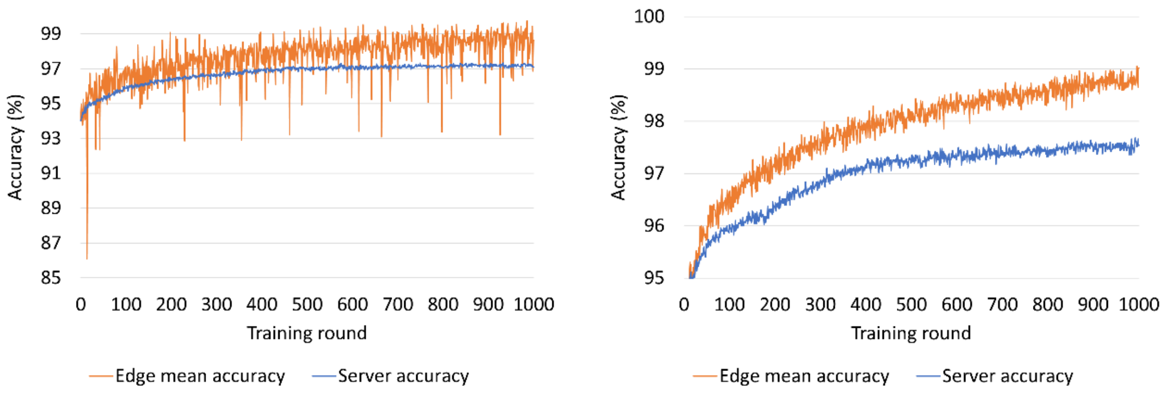


Figure 4.3: Accuracy on cloud server for R-FedAvg/FedAvg

Figure 4.3 shows the average accuracy of 10 tests. As observed, the aggregation mechanism of R-FedAvg, which is the same as FedAvg, converges independently of the data partition scheme. Considering the centralized dataset of the server, the difference between the most and least accurate model is no more than 2.81%. However, the ANN must fit the data of deep edge devices, rather than the centralized data. This is where FedAvg shows one of its flaws. Figure 4.4-a) shows the average accuracy of edge devices per training round in contrast to server accuracy. As observed, the ANN shows a very unstable accuracy. The edge accuracy drops on average 4% after the training round 100. The maximum average accuracy drop was registered before this round and achieved a value of 8.3%. If considered the accuracy losses on individual devices, these values go even further - 60% of the devices registered drops higher than 15% before the training round 100 and higher than 9% after that round.

To tackle this issue, R-FedAvg is integrated with an additional training round, which is strictly performed on the edge after the deployment of the new global ANN. This extra training round allows users to tweak the ANN to fit the local dataset. For this training round, the process relies on L-SGD (int-8) described in Section 3.2.1. As observed in Figure 4.4-b), the average accuracy on the edge is now far more stable and outperformed the accuracy measured in the cloud



(a) Average edge accuracy for FedAvg (b) Average edge accuracy for R-FedAvg

Figure 4.4: Average edge accuracy: FedAvg vs. R-FedAvg

4.2.2 Data Perturbation

Stragglers: Lost connections due to connectivity problems, device energy constraints, or device availability expose the system to middle-round dropouts. The main problem of this real world-scenario is the slow down of the overall learning process. Since those devices do not complete the training plan, the aggregation discards their progress and performs over a smaller portion of clients. Figure 4.5 shows that the presence of stragglers does not affect the convergence speed. However, the higher the percentage of stragglers, the higher the accuracy variance. The oscillating pattern results from increasing the weight of devices that completed the round. Increasing the weight of a device takes the ANN to fit that device instead of improving the overall model. Biased ANNs will outperform on some edge devices, but will underperform on the majority of them, leading to accuracy loss. After the analysis of Figure 4.5, it is proposed to discard a training round if the percentage of stragglers is over 50%.

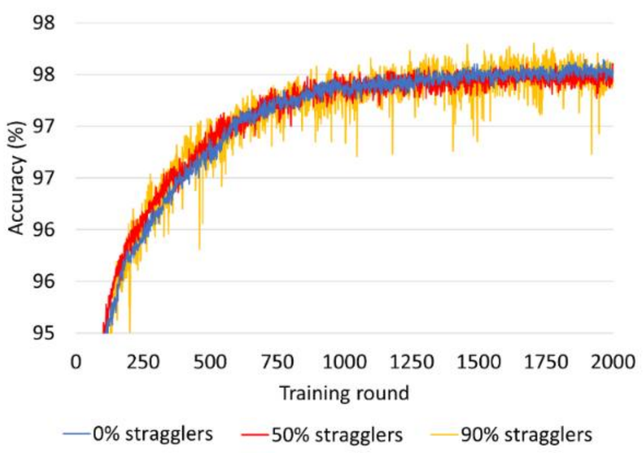


Figure 4.5: R-FedAvg under different percentage of stragglers

Data poisoning: An attacker can poison the local dataset of an edge device or the respective training report. To evaluate the data poisoning effect, it was considered four types of noise: (i) gaussian, (ii) salt&pepper, (iii) speckle, and (iv) poison, as presented in Figure 4.6. Furthermore, the selection of devices that contain poisoned data is random, and each device can be affected by more than one type of noise and varying intensity. Once a device is selected as a poison data holder, the entire local dataset is poisoned. Figure 4.7 details the accuracy of the global ANN returned by R-FedAvg for different percentages of data poisoners in the network. The worst test case scenario considered that 90% of the devices stored poisoned data. As can be observed, there is no relevant accuracy loss or degradation in the convergence speed when the percentage of poisoners increase.

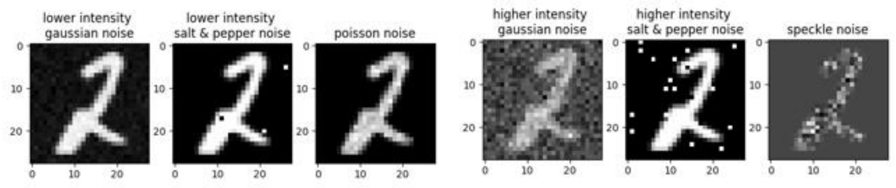

lower intensity gaussian noise
lower intensity salt & pepper noise
poisson noise
higher intensity gaussian noise
higher intensity salt & pepper noise
speckle noise

Figure 4.6: Different noise added to data

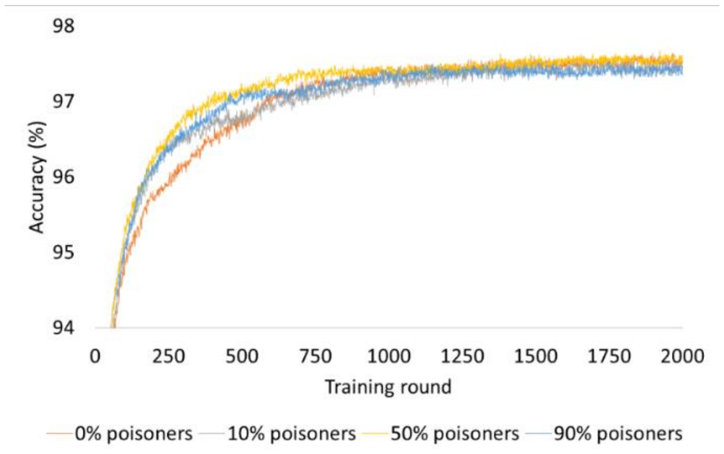


Figure 4.7: R-FedAvg under different percentage of data poisoners

To test the behavior of FedAvg against R-FedAvg when facing training report poisoning, it was considered a federated training with 50 training rounds and poisoned the training report of 1 out of 100 edge devices at the training round 31. The amplitude of weights in this device is augmented by 30000x and the size of the local dataset size by 500x. Results are detailed in Figure 4.8. As observed, compromising the training report of a single device can compromise the whole aggregation mechanism of FedAvg. Amplifying the magnitude of weights to extremely high values and increasing their relevance on the whole aggregation mechanism, by cheating on the local dataset size, is sufficient to cause an overflow on the loss calculated during the aggregation, leading the FLS to a non-return point. To tackle this issue, R-FedAvg softens the impact of data poisoning by employing a mechanism to detect and discard local model updates whose parameters diverge considerably from the parameters returned by the remaining edge devices and from the current global ANN. Figure 4.8 shows that R-FedAvg is not vulnerable to this attack as the accuracy of the aggregated model continues to increase after the attack.

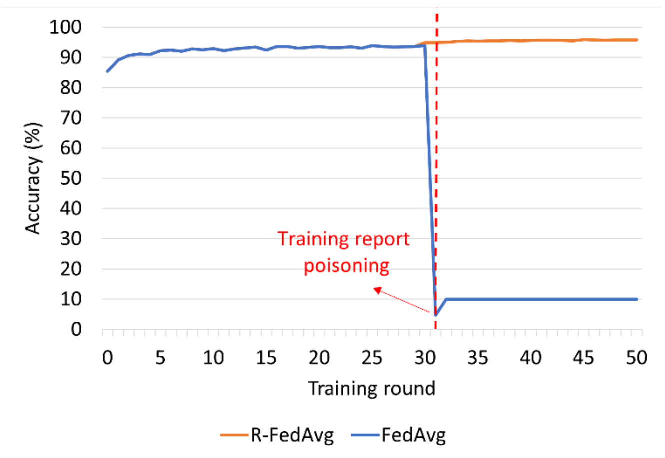


Figure 4.8: FedAvg vs. R-FedAvg under training report poisoning

Chapter 5

Conclusions

The last chapter of this MSc thesis presents the conclusion of this work based on the analysis of the results presented in the previous section. Furthermore, this chapter outlines some suggestions aiming to improve and extend the developed system. The struct of this chapter splits into two main subsections: (i) discussions and (ii) future work.

5.1 Discussion

The design of an FL framework requires intrinsic knowledge in a wide range of scientific areas. The development of the local edge mechanisms addresses multiple topics to enable memory and complexity reduction associated with the stringent requirements of low-end devices. The development of the edge process fundamentally relies on two main topics: (i) ML internal computations and (ii) quantization schemes, outlined in Chapter 2 aiming to cover every key knowledge to understand the designing steps.

The deployed edge mechanisms are composed of two different methods: (i) inference pass and (ii) training pass. The inference mechanism relies on the CMSIS-NN library. To ease the process of porting ANNs developed using conventional tools such as TensorFlow to edge devices, it was developed a framework that abstracts the programmer from the constraints of CMSIS-NN. The framework also allows the definition of a training plan to be deployed on edge. The developed tool ensures dataset quantization and code generation.

Training on edge is a high complexity process. The analysis of different optimization methods allowed the identification of SGD as the most reliable algorithm to be deployed on the deep edge. The developed training algorithm (L-SGD) is a version of the classic SGD algorithm optimized for low-memory footprint and latency on Arm Cortex-M MCUs. To test the accuracy of quantized training it was developed three versions of this algorithm: (i) L-SGD (float-32), (ii) L-SGD (int-8), and (iii) L-SGD (int-16). Results show that L-SGD (float-32) is 16.35x faster than the classic SGD implemented by TensorFlow while using 13.72% less memory. This comes at the cost of a negligible accuracy drop of 0.12%.

Although the quantization effect does not produce significant loss performance on the inference process, the learning process requires a most precise approach. Furthermore, fully-quantized training over 8-bit data is still not feasible as it induces severe accuracy losses. To tackle the accuracy loss produced by the 8-bit version of L-SGD, a 16-bit version of the L-SGD was introduced. However, the accuracy loss was similar to the 8-bit version, and the overhead introduced by the data extension operation increases the process latency. This result shows that training only on quantized parameters is unreliable under the defined quantization policy. However, L-SGD (int-8) is very useful to tweak global ANNs to the local data distribution when the FLS deploys a new ANN to the edge. At this stage, it outperforms L-SGD (float-32) by 2.22% with a speedup of 1.97x while requiring only 25% of the memory used by the floating-point version.

Table 5.1 summarizes the basic properties of the optimizers previously discussed in Section 2.1.4.5 and the developed training mechanism, L-SGD. Due to limited hardware resources, there is a demand for methods that prioritize memory footprint reduction while maintaining low computational complexity. Besides the presented solution, there are other optimization mechanisms. However, the introduction of complex mathematical operations is prohibitive to deep edge devices. Furthermore, implementing such techniques on target devices is unreliable, and they are not explored for this purpose. In opposition, L-SGD aims to tackle the addressed problem by reducing the memory footprint and the computation cost of the training process.

Table 5.1: Functionality of previous research optimizers against L-SGD

Optimizer	**Tunable parameters**	**Advantages**	**Disadvantages**
GD [43]	1	Low computational complexity	High memory footprint Vulnerable to local minima stuck
SGD [44]	1	Low computational complexity Reduced memory footprint	Vulnerable to local minima stuck Slower than GD
AdaGrad [45]	1	Automatic learning rate decay	Slower convergence at later iterations
Adam [46]	3	Faster training	High memory footprint Higher number of tunable parameters
L-SGD	1	Low computational complexity Reduced memory footprint	Vulnerable to local minima stuck Slower than GD

The development of a robust centralized aggregator is crucial to FLS. There is a demand for a design of a reliable server that meets the fair aggregation, robustness to data heterogeneity, and robustness to adversarial behaviors. For this purpose, it was evaluated a series of aggregation algorithms, from which stand out the FedAvg (detailed in Section 2.2.4). FedAvg is resistant to data heterogeneity, but it has major shortcomings in terms of fair aggregation and overcoming adversarial behavior. This MSc thesis focused on increasing the robustness of FedAvg in these three metrics, which concluded in a novel aggregation algorithm called R-FedAvg. The developed mechanism tackles two major flaws of the current state-of-the-art aggregation mechanism. The introduction of an additional training step allows users to personalize the model to local data, boosting the model accuracy. This paradigm shift reduces the data heterogeneity effect, reducing the effect of data partitions. R-FedAvg pushes the state-of-the-art by providing mechanisms to detect and discard poisoned training reports. Results show that while FedAvg under this attack leads an FLS to a non-return point, R-FedAvg continues with the training pass without any accuracy loss.

Table 5.2 summarizes the basic properties of the aggregation mechanisms previously discussed in Section 2.2.4 and the developed mechanism, R-FedAvg. As can be observed, aggregation algorithms developed to date have never considered data poisoning as a priority. R-FedAvg pretends to push the state-of-the-art by proposing a mechanism to detect and discard poisoned training reports. Furthermore, R-FedAvg is the only alternative to Per-FedAvg in terms of user personalization, allowing edge devices to perform some additional training steps after the deployment of the global ANN to the edge. These additional training steps enable edge devices to tweak the global model to the local data distribution.

Table 5.2: Functionality of previous research aggregation mechanisms against R-FedAvg

R&D Study	**Non-IID Data**	**User Personalization**	**Stragglers**	**Dataset Poisoning**	**Train Report Poisoning**
FedAvg [59]	Yes	No	Drop-out	No	No
FedMA [74]	Yes	No	n.d.	No	No
FedProx [75]	Yes	No	Tolerates partial work	No	No
q-FedAvg [76]	Yes	No	n.d.	No	No
Per-FedAvg [77]	Yes	Yes	n.d.	No	No
R-FedAvg	Yes	Yes	Drop-out	No	Yes

In conclusion, the developed framework presents a solution that allows a robust distributed learning solution that can perform training on multiple low-end devices. Training only on quantized conditions over

CMSIS-NN constraints with negligible performance loss has shown as an unreliable process. However, training the model over floating-point data, and performing a model tweak over quantized procedures has shown a significant performance increase. Thus, the features included in this framework on the FL process revealed considerable robustness which hikes the deployment of a real-world FLS.

5.2 Future work

Analyzing the experimental results and the conclusions in the previous section allows to define some points where the developed FLS can be improved or extended with new functionalities. Although the goals for the FLS have been met, there are currently some limitations. Although the floating-point solution reduced the memory footprint, full quantization processes allow an even more significant reduction as presented before. The next steps on the development of FLS on low-end devices must consider the research of new quantization schemes that reduce the training process error.

Direct attacks do not affect the users and server ecosystem directly. However, the learning effect process is delayable through malicious behaviors in communications. As a consequence, secure communication channels must be integrated in a future version of this FLS. Although the introduced overhead by the encryption and decryption operations, this mechanism adds a new layer of security. Exposing data on an open communication channel is always an open window to malicious actors.

Finally, a future step is to extend the current framework to support different model architectures. Currently, the developed framework only supports ANNs with fully-connected layers. However, CNN architectures are a commonly explored architecture in image classification applications, registering state-of-the-art accuracy in a myriad of applications, from healthcare to face recognition. Future versions of this FLS must provide support for different ML model architectures such as CNN and Long Short-Term Memory (LSTM). To extend the framework application, future versions of this FLS must also seek to provide support for RISC-V (RV32IMCXpulp) MCUs [10], [79], [80], which is a rising computing architecture for ML in low-power applications.

Bibliography

[1] P. Sparks, “The route to a trillion devices,” *White Paper, ARM*, 2017.

[2] D. Jiang, “Application of Artificial Intelligence in Computer Network Technology in big data era,” in *2021 International Conference on Big Data Analysis and Computer Science (BDACS)*, 2021, pp. 254–257, DOI: 10.1109/BDACS53596.2021.00063.

[3] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, “A survey of deep learning techniques for autonomous driving,” *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020, DOI: https://doi.org/10.1002/rob.21918.

[4] M. Costa, D. Oliveira, S. Pinto, and A. Tavares, “Detecting Driver’s Fatigue, Distraction and Activity Using a Non-Intrusive Ai-Based Monitoring System,” *Journal of Artificial Intelligence and Soft Computing Research*, vol. 9, no. 4, pp. 247–266, 2019, DOI: doi:10.2478/jaiscr-2019-0007.

[5] O. A. Alimi, K. Ouahada, and A. M. Abu-Mahfouz, “A Review of Machine Learning Approaches to Power System Security and Stability,” *IEEE Access*, vol. 8, pp. 113512–113531, 2020, DOI: 10.1109/ACCESS.2020.3003568.

[6] C. Feng, S. Wu, and N. Liu, “A user-centric machine learning framework for cyber security operations center,” in *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, 2017, pp. 173–175, DOI: 10.1109/ISI.2017.8004902.

[7] F. Ahamed and F. Farid, “Applying Internet of Things and Machine-Learning for Personalized Healthcare: Issues and Challenges,” in *2018 International Conference on Machine Learning and Data Engineering (iCMLDE)*, 2018, pp. 19–21, DOI: 10.1109/iCMLDE.2018.00014.

[8] K. Shailaja, B. Seetharamulu, and M. A. Jabbar, “Machine Learning in Healthcare: A Review,” in *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2018, pp. 910–914, DOI: 10.1109/ICECA.2018.8474918.

[9] J.-B. Wang, J. Wang, Y. Wu, J.-Y. Wang, H. Zhu, M. Lin, and J. Wang, “A machine learning framework for resource allocation assisted by cloud computing,” *IEEE Network*, vol. 32, no. 2, pp. 144–151, 2018.

[10] J. Jiang, L. Hu, C. Hu, J. Liu, and Z. Wang, “BACombo—Bandwidth-Aware Decentralized Federated Learning,” *Electronics*, vol. 9, no. 3, 2020 [Online], DOI: 10.3390/electronics9030440, ISSN: 2079-9292. Available: https://www.mdpi.com/2079-9292/9/3/440

[11] J. Jiang, L. Hu, C. Hu, J. Liu, and Z. Wang, “BACombo—Bandwidth-Aware Decentralized Federated Learning,” *Electronics*, vol. 9, no. 3, p. 440, 2020.

[12] L. Lai, N. Suda, and V. Chandra, “Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus,” *arXiv preprint arXiv:1801.06601*, 2018.

[13] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, “Adaptive federated learning in resource constrained edge computing systems,” *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1205–1221, 2019.

[14] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, “Edge intelligence: The confluence of edge computing and artificial intelligence,” *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7457–7469, 2020.

[15] D. Oliveira, M. Costa, S. Pinto, and T. Gomes, “The future of low-end motes in the Internet of Things: A prospective paper,” *Electronics*, vol. 9, no. 1, p. 111, 2020.

[16] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Kone\vcn\`y, S. Mazzocchi, H. B. McMahan, and others, “Towards federated learning at scale: System design,” *arXiv preprint arXiv:1902.01046*, 2019.

[17] N. Suda and D. Loh, “Machine learning on arm cortex-m microcontrollers,” *Arm Ltd.: Cambridge, UK*, 2019.

[18] M. Costa, D. Costa, T. Gomes, and S. Pinto, “Shifting Capsule Networks from the Cloud to the Deep Edge,” 2021.

[19] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O’Reilly Media, 2019.

[20] G. Hackeling, *Mastering Machine Learning with scikit-learn*. Packt Publishing Ltd, 2017.

[21] P. Kashyap, *Machine learning for decision makers: Cognitive computing fundamentals for better decision making*. Springer, 2017.

[22] G. Bonaccorso, *Machine learning algorithms*. Packt Publishing Ltd, 2017.

[23] S. Gollapudi, *Practical machine learning*. Packt Publishing Ltd, 2016.

[24] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[25] Y. Artan, “Interactive Image Segmentation Using Machine Learning Techniques,” in *2011 Canadian Conference on Computer and Robot Vision*, 2011, pp. 264–269, DOI: 10.1109/CRV.2011.42.

[26] A. Nagaraja, U. Boregowda, K. Khatatneh, R. Vangipuram, R. Nuvvusetty, and V. Sravan Kiran, “Similarity Based Feature Transformation for Network Anomaly Detection,” *IEEE Access*, vol. 8, pp. 39184–39196, 2020, DOI: 10.1109/ACCESS.2020.2975716.

[27] Q. Sun, W. He, and L. Chen, “Multi-Label Automatic Labeling for Question Attributes Based on Adaboost and Bayes Algorithms,” in *2018 Chinese Automation Congress (CAC)*, 2018, pp. 2955–2960, DOI: 10.1109/CAC.2018.8623611.

[28] C. Gallo, “Artificial neural networks tutorial,” in *Encyclopedia of Information Science and Technology, Third Edition*, IGI Global, 2015, pp. 6369–6378.

[29] B. M. Wilamowski, “Neural network architectures and learning algorithms,” *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 56–63, 2009.

[30] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, “Activation functions: Comparison of trends in practice and research for deep learning,” *arXiv preprint arXiv:1811.03378*, 2018.

[31] T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009 [Online], ISBN: 9780387848846. Available: https://books.google.pt/books?id=eBSgoAEACAAJ

[32] M. A. Rahman, M. M. Rahman, M. N. H. Mollah, and others, “Robust Hierarchical Clustering for Metabolomics Data Analysis in presence of Cell-wise and Case-wise outliers,” in *2018 International Conference on Computer, Communication, Chemical, Material and Electronic Engineering (IC4ME2)*, 2018, pp. 1–4.

[33] S. Jiang and Q. An, “Clustering-based outlier detection method,” in *2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery*, 2008, vol. 2, pp. 429–433.

[34] R. Pamula, J. K. Deka, and S. Nandi, “An outlier detection method based on clustering,” in *2011 Second International Conference on Emerging Applications of Information Technology*, 2011, pp. 253–256.

[35] L. Morissette and S. Chartier, “The k-means clustering technique: General considerations and implementation in Mathematica,” *Tutorials in Quantitative Methods for Psychology*, vol. 9, no. 1, pp. 15–24, 2013.

[36] H. C. Mandhare and S. Idate, “A comparative study of cluster based outlier detection, distance based outlier detection and density based outlier detection techniques,” in *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2017, pp. 931–935.

[37] D. Greene, P. Cunningham, and R. Mayer, “Unsupervised learning and clustering,” in *Machine learning techniques for multimedia*, Springer, 2008, pp. 51–90.

[38] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[39] G. Kim, C. S. Hwang, and D. S. Jeong, “Stochastic Learning with Back Propagation,” in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[40] J. Han, M. Kamber, and J. Pei, “Data mining concepts and techniques, third edition,” 2012. [Online], ISBN: 0123814790. Available: http://www.amazon.de/Data-Mining-Concepts-Techniques-Management/dp/0123814790/ref=tmm_hrd_title_0?ie=UTF8&qid=1366039033&sr=1-1

[41] H. Bhavsar and A. Ganatra, “A comparative study of training algorithms for supervised machine learning,” *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, no. 4, pp. 2231–2307, 2012.

[42] D. O’Connor, “A Historical Note on Shuffle Algorithms,” *Retrieved Maret*, vol. 4, p. 2018, 2014.

[43] S. Sun, Z. Cao, H. Zhu, and J. Zhao, "A survey of optimization methods from a machine learning perspective," *IEEE transactions on cybernetics*, vol. 50, no. 8, pp. 3668–3681, 2019.

[44] B. Bharath and V. S. Borkar, "Stochastic approximation algorithms: Overview and recent trends," *Sādhanā*, vol. 24, no. 4–5, pp. 425–452, 1999 [Online], DOI: 10.1007/bf02823149. Available: https://app.dimensions.ai/details/publication/pub.1003895514

[45] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *J. Mach. Learn. Res.*, vol. 12, no. null, pp. 2121–2159, 2011, ISSN: 1532-4435.

[46] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," 2014 [Online]. Available: http://arxiv.org/abs/1412.6980

[47] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, and others, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[48] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International conference on machine learning*, 2016, pp. 2849–2858.

[49] L. Lai, N. Suda, and V. Chandra, "Deep convolutional neural network inference with floating-point weights and fixed-point activations," *arXiv preprint arXiv:1703.03073*, 2017.

[50] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *arXiv preprint arXiv:2103.13630*, 2021.

[51] P.-E. Novac, G. B. Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantization and Deployment of Deep Neural Networks on Microcontrollers," *Sensors*, vol. 21, no. 9, p. 2984, 2021.

[52] P. Wang, Q. Chen, X. He, and J. Cheng, "Towards accurate post-training network quantization via bit-split and stitching," in *International Conference on Machine Learning*, 2020, pp. 9847–9856.

[53] A. Imteaj and M. H. Amini, "FedAR: Activity and Resource-Aware Federated Learning Model for Distributed Mobile Robots," 2021.

[54] X. Qu, J. Wang, and J. Xiao, "Quantization and Knowledge Distillation for Efficient Federated Learning on Edge Devices," in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2020, pp. 967–972, DOI: 10.1109/HPCC-SmartCity-DSS50907.2020.00129.

[55] E. Rizk, S. Vlaski, and A. H. Sayed, "Dynamic Federated Learning," in *2020 IEEE 21st International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, 2020, pp. 1–5, DOI: 10.1109/SPAWC48557.2020.9154327.

[56] Q. Li, Z. Wen, and B. He, "Federated learning systems: Vision, hype and reality for data privacy and protection," *arXiv preprint arXiv:1907.09693*, 2019.

[57] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[58] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009, DOI: 10.1109/tkde.2009.191.

[59] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, 2017, pp. 1273–1282.

[60] J. Kone\vcn\`y, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *arXiv preprint arXiv:1610.02527*, 2016.

[61] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, and others, "Advances and open problems in federated learning," *arXiv preprint arXiv:1912.04977*, 2019.

[62] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, 2018, pp. 63–71.

[63] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[64] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164, p. 20190155, 2020.

[65] "Federated Learning: Collaborative Machine Learning without Centralized Training Data," *Google AI Blog*, 2017 [Online]. Available: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html

[66] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1175–1191, DOI: 10.1145/3133956.3133982.

[67] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979, DOI: 10.1145/359168.359176.

[68] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976, DOI: 10.1109/TIT.1976.1055638.

[69] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2000, pp. 531–545, DOI: 10.1007/s00145-008-9026-x.

[70] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudorandom bits," *SIAM journal on Computing*, vol. 13, no. 4, pp. 850–864, 1984, DOI:

10.1109/SFCS.1982.72.

[71] A. C. Yao, "Theory and application of trapdoor functions," in *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*, 1982, pp. 80–91, DOI: 10.1109/SFCS.1982.45.

[72] P. I. S. Peña and R. E. G. Torres, "Authenticated Encryption based on finite automata cryptosystems," in *2016 13th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, 2016, pp. 1–6, DOI: 10.1109/ICEEE.2016.7751254.

[73] "TensorFlow Federated," *TensorFlow* [Online]. Available: https://www.tensorflow.org/federated

[74] H. Wang, M. Yurochkin, Y. Sun, D. Papailiopoulos, and Y. Khazaeni, "Federated Learning with Matched Averaging," 2020.

[75] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *arXiv preprint arXiv:1812.06127*, 2018.

[76] T. Li, M. Sanjabi, A. Beirami, and V. Smith, "Fair resource allocation in federated learning," *arXiv preprint arXiv:1905.10497*, 2019.

[77] A. Fallah, A. Mokhtari, and A. Ozdaglar, "Personalized Federated Learning with Theoretical Guarantees: A Model-Agnostic Meta-Learning Approach," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[78] S. Ek, F. Portet, P. Lalanda, and G. Vega, "Evaluation of federated learning aggregation algorithms: application to human activity recognition," in *Adjunct Proceedings of the 2020 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2020 ACM International Symposium on Wearable Computers*, 2020, pp. 638–643.

[79] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, "Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10375–10383, 2019, DOI: 10.1109/JIOT.2019.2939008.

[80] S. Pinto and J. Martins, "The industry-first secure IoT stack for RISC-V: a research project," in *RISC-V Workshop,(Zurich)*, 2019.