Chair of
Information Theory
and Data Analytics

RWTH AACHEN
UNIVERSITY

DEEP COMPRESSION IN FEDERATED LEARNING

TOWARDS LIGHTWEIGHT AND ENERGY-EFFICIENT DISTRIBUTED INTELLIGENCE

Elouan COLYBES

Chair of Information Theory and Data Analytics
RWTH Aachen University

Chair of
Information Theory
and Data Analytics

RWTH AACHEN
UNIVERSITY

DEEP COMPRESSION IN FEDERATED LEARNING

TOWARDS LIGHTWEIGHT AND ENERGY-EFFICIENT DISTRIBUTED INTELLIGENCE

Elouan COLYBES

A thesis submitted to the
Faculty of Electrical Engineering and Information Technology,
RWTH Aachen University,

reviewed by Univ.-Prof. Dr. Ing. Anke Schmeink and supervised by Shirin Salehi, Ph.D.

Chair of Information Theory and Data Analytics
RWTH Aachen University

Eidesstattliche Versicherung

Colybes, Elouan	461176
Name, Vorname	Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

Deep Compression in Federated Learning: Towards Lightweight and Energy-Efficient Distributed Intelligence

selbständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt; dies umfasst insbesondere auch Software und Dienste zur Schprch-, Text-; und Medienproduktion. Ich erkläre, dass für den Fall, dass die Arbeit in unterschieden Formen eingereicht wird (z.B. elektronisch, gedruckt, geplottet, auf einem Datenträger) alle eingereichten Versionen vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, den 25.04.2025	
Ort, Datum	Unterschrift

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, den 25.04.2025	
Ort, Datum	Unterschrift

Abstract

This thesis focuses on reducing the environmental footprint of artificial intelligence (AI) by compressing model representations within the Federated Learning (FL) framework. In particular, it investigates deep compression techniques applied to deep neural networks (DNNs), aiming to optimize communication efficiency and computational demands. In FL, minimizing model size offers a significant advantage: it reduces the amount of data exchanged between clients and the central server, thereby alleviating communication bottlenecks, especially when multiple clients send updates simultaneously. The proposed approach combines pruning, quantization, and Huffman encoding into a unified Full Compression Pipeline (FCP). Through a series of experiments on various neural network architectures, this thesis demonstrates that the FCP can substantially compress model representations with minimal impact on final model accuracy. The pipeline is evaluated in an independent and identically distributed (IID) data setting. In one representative scenario, training a ResNet-12 model on the CIFAR-10 dataset with ten clients and a 5 Mbps bandwidth, the FCP achieves a 6× reduction in model size, with only a 5% drop in accuracy compared to the uncompressed baseline. This results in communication rounds that are up to 1.75 times faster. These findings highlight the potential of the FCP to enhance communication and computational efficiency in FL, while contributing to more sustainable AI practices.

Acknowledgements

I would like to thank my thesis supervisors Prof. Dr.-Ing. Anke Schmeink geb. Feiten and Shirin Salehi, Ph.D. for providing me with the opportunity to work on a challenging and really interesting problem.

I would also like to sincerely express my gratitude to the entirety of the INDA research team and in particular Mrs. Salehi for her insightful suggestions and constructive feedback that greatly contributed to the development of this thesis.

Lastly, I would like to thank all my friends and family who supported me during this thesis.

Contents

1 Introduction

In this chapter the thesis is introduced. Section 1.1 presents the motivation for the chosen topic, section 1.2 describes the aims of the thesis and section 1.3 outlines the content.

1.1 Motivation

In recent years, the field of Artificial Intelligence (AI) has witnessed an unprecedented surge in both academic interest and public attention, largely driven by the rapid advancement and widespread diffusion of Large Language Models (LLM). These models, capable of understanding and generating human-like language, have significantly influenced how AI is perceived and utilized across various domains. Accompanying this surge is a parallel trajectory of growth in computational capabilities, which has enabled the training and deployment of increasingly complex and larger-scale Neural Networks (NNs).

However, this evolution comes with a cost. The training and operation of such large models require immense computational resources and energy consumption, raising critical concerns around sustainability, scalability, and environmental impact. These concerns have given rise to the emerging paradigm of *Green AI*, which emphasizes the development of AI systems that are not only effective but also efficient in terms of computation and energy use. *Green AI* stands in contrast to *Red AI*, which prioritizes performance improvements without regard to computational cost.

Within the broad landscape of Machine Learning (ML), Federated Learning (FL) has emerged as a compelling approach to distributed model training. FL allows a central server to train a global neural network model by leveraging the local data and computational resources of a network of distributed clients. This is achieved without directly accessing the clients' private data, thereby offering advantages in terms of privacy and data governance. Nonetheless, FL introduces significant technical challenges—most notably, the high volume of communication rounds required between clients and the server, and the substantial computational demands on each participating device.

In light of these challenges, this thesis explores the integration of deep compression techniques within the FL framework. Deep compression refers to a suite of methods designed

to reduce the size and complexity of neural networks—such as pruning, quantization, and knowledge distillation—without significantly compromising their predictive performance. By embedding such techniques into the FL pipeline, the goal is to enhance both computational efficiency and communication performance, thereby aligning with the principles of *Green AI*.

This work investigates the feasibility, implementation, and impact of combining deep compression with FL. Specifically, it examines how these techniques can be harmonized to reduce the overall system cost, improve scalability, and maintain model accuracy, all while addressing the pressing concerns of sustainability and efficiency in modern AI systems.

1.2 Aims

We aim to enhance the computational performance of the FL process by implementing a comprehensive compression pipeline, referred to as the Full Compression Pipeline (FCP). This pipeline is designed to improve both communication and computational efficiency, with the ambition of surpassing current state-of-the-art approaches. The main objectives of this work are as follows:

- **FL Simulation Framework:** Develop a simulated FL environment that enables the integration and customization of various deep compression techniques. This framework should also support rigorous performance evaluation under different settings and constraints.

- **Compression Techniques Design and Optimization:** Design and implement compression methods that are specifically adapted for the FL setting. These techniques will be optimized within the FCP to maximize their efficiency without compromising model accuracy or convergence.

- **Generalization and Scalability:** Demonstrate the generalizability of the FCP by applying it to diverse neural network architectures and datasets. This will validate the robustness and flexibility of the proposed framework across a range of practical scenarios.

1.3 Thesis structure

In order to address the previous objectives, the remainder of this thesis is organized as follows. In Chapter 2, we lay the theoretical foundations necessary to understand the core topics of this work, including artificial neural networks, federated learning, and deep

compression techniques. Chapter 3 provides an overview of existing literature related to *Green AI* and the application of deep compression methods within the FL paradigm. In Chapter 4, we describe the design choices and structure of the proposed Full Compression Pipeline (FCP) within the FL framework. Chapter 5 presents the results of the experiments conducted as part of this thesis, offering insights into the performance and applicability of the proposed methods. Finally, Chapter 6 summarizes the key findings and outlines possible directions for future work.

2 Theoretical foundations

In this chapter, we aim to establish theoretical foundations that are prerequisites for understanding the work described in this paper. First, Section 2.1 explains all relevant aspects of NNs, then Section 2.2 describes FL and finally Section 2.3 lists compression techniques related to our work.

2.1 Deep Neural Networks

2.1.1 Introduction

In the vast field of AI, artificial neural networks imposed themselves as a reference for ML processes. They are studied since a few decades, but Deep Neural Networks (DNNs) have become a real center of attention — both in research and among the general public — over the past 20 years.

The basic element of a neural network is a *perceptron unit*, also named *neuron*, introduced in 1957 by Frank Rosenblatt in the early days of AI [36]. It takes an input $(x_1, \dots, x_d) \in \mathbb{R}^d$ and processes it to give an output $z \in \mathbb{R}$. This is done in two stages, first by calculating the weighted sum b with the weight vector $\boldsymbol{\omega} = (\omega_0, \dots, \omega_d) \in \mathbb{R}^{d+1}$. Note that ω_0 is the *bias*, it is added independently of the rest of the sum:

$$b = \sum_{i=1}^{d} \omega_i x_i + \omega_0 \tag{2.1.1}$$

To make the writing easier, we can introduce $x_0 = 1$ and $\boldsymbol{x} = (x_0, x_1, \dots, x_d)$, so that the equation becomes:

$$b = \sum_{i=0}^{d} \omega_i x_i = \boldsymbol{\omega} \cdot \boldsymbol{x} \tag{2.1.2}$$

The second stage is to apply to b an *activation function* of the form $h : \mathbb{R} \to \mathbb{R}$ to finally get $\boldsymbol{z}$:

$$z = h(b) = h(\boldsymbol{\omega} \cdot \boldsymbol{x}) \tag{2.1.3}$$

The Figure 2.1 gives a symbolic representation of the perceptron unit.

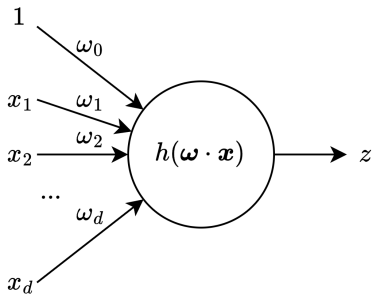


Figure 2.1: Schematic of a perceptron unit

Let's now consider $\boldsymbol{z}$ not as an integer, but as an element of $\mathbb{R}^m$. We have to adapt the weight vector to a matrix $\boldsymbol{\omega} \in \mathbb{R}^{m \times (d+1)}$. For each $j \in \{1, \ldots, m\}$, we define:

$$z_j = h(b_j) = h(\sum_{i=0}^{d} \omega_{ji} x_i) \tag{2.1.4}$$

And $\boldsymbol{z} = (z_1, \ldots, z_m)$ is the result of the perceptron's transformation on $\boldsymbol{x} = (x_1, \ldots, x_d)$. In more recent literature, such a perceptron is often called a *linear layer*. It can be represented as in the Figure 2.2.

To sum this up, the perceptron is defined by its weight matrix $\boldsymbol{\omega}$ and its activation function h. Its takes as input a vector of size d and returns as output a vector of size m. If we consider the output vector $\boldsymbol{z}$ of a perceptron as the input vector of another perceptron, we build a 2-layers perceptron. There is no theoretical limit to the depth of a *multi-layer perceptron* (MLP), but the computational power needed to train and run the model increases with the number of parameters.

We can repeat this process and build an even deeper network. If we give a number for each layer, from 1 to l, the first layer of such a neural network is called the *input layer*, and the last one is the *output layer*. All the other layers (2 to $l - 1$) are called *hidden layers*.

For example, let's consider a MLP with $l = 2$ layers. We note $\boldsymbol{x} \in \mathbb{R}^d$ the input of the MLP, and $\boldsymbol{z} \in \mathbb{R}^n$ its output. Let $\boldsymbol{y} \in \mathbb{R}^m$ be the output vector of the first layer, which is

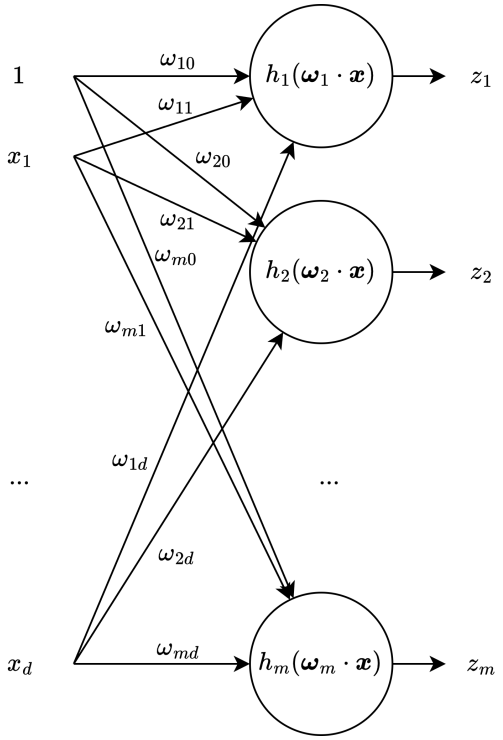


Figure 2.2: Schematic of a full perceptron

also the input vector of the second layer. Then we can write:

$$\begin{aligned} &y_j = h^1(b_j^1) \text{ for } j = 1 \text{ to } m \\ &\text{with } b_j^1 = \sum_{i=0}^{d} \omega_{ji}^1 x_i \\ \\ &z_k = h^2(b_k^2) \text{ for } k = 1 \text{ to } n \\ &\text{with } b_k^2 = \sum_{j=0}^{m} \omega_{kj}^2 y_j \end{aligned} \tag{2.1.5}$$

where $h^{(1)} : \mathbb{R} \to \mathbb{R}$, $\boldsymbol{\omega}^{(1)} \in \mathbb{R}^{d \times m}$ and $h^{(2)} : \mathbb{R} \to \mathbb{R}$, $\boldsymbol{\omega}^{(2)} \in \mathbb{R}^{m \times n}$ are respectively the activation functions and weights of the first and second layer.

The transformation of an input to an output through the NN is called *forward pass* and its algorithm is described in Algorithm 1.

Now we've described the basic architecture of this kind of NN, we want it to be successful on tasks we assign it. Training a MLP means to learn the weights $\boldsymbol{\omega}^{(i)}$ for $i = 1$ to l, so that for every input $\boldsymbol{x}$, the output $\boldsymbol{z}$ is satisfying. The next section explains what this

Algorithm 1 Forward pass

Input: Network depth, l
Input: $\boldsymbol{\omega^k}$, the weights of layer k of the model
Input: h^k, the activation function of the layer k
Input: $\boldsymbol{x}$, the input to process
1: $\boldsymbol{z}^0 \leftarrow \boldsymbol{x}$
2: **for** layer $k = 1$ to l **do**
3: $\quad \boldsymbol{z}^k \leftarrow h^k(\boldsymbol{\omega}^k \cdot \boldsymbol{z}^{k-1})$
4: **end for**
5: $\boldsymbol{y} \leftarrow \boldsymbol{z}^l$
6: **return** $\boldsymbol{y}$

means.

2.1.2 Layers and Activation Functions

For now, we've described the MLP: its layers are also called *linear layers*. However, more recent research has conducted to introduce news kinds of layers, to enhance new possibilities. In this section we will review those, as well as the different activation functions that can be used.

1. Activation functions

As explained for the MLP, the *activation function* is a function applied right before the output of a layer, that will affect the returned values. Because complex problems are often non-linear, we want to introduce non-linearities to NNs so that they can handle such complex situations. Typical examples are:

- **Rectified Linear Unit (ReLU):**
  Its mathematical formula is described in Equation (2.1.6), and represented in Figure 2.3a. All negative outputs are set to zero, and positive outputs stay unchanged.

$$\begin{aligned} \text{ReLU} \ : \ & \mathbb{R} \to \mathbb{R}^+ \\ & x \mapsto \begin{cases} x & \text{if } x > 0 \\ 0 & \text{else} \end{cases} \ = \ \max(0, x) \end{aligned} \tag{2.1.6}$$

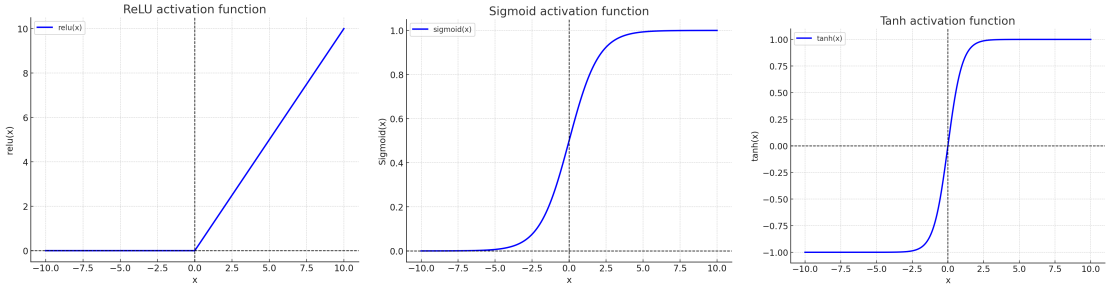
  It is widely used in the hidden layers of deep networks, allows a fast computation.

- **Sigmoid:**
  This function is non-linear too, but it bounds the outputs between 0 and 1. Its mathematical definition is described in Equation (2.1.7), and the function is represented

in Figure 2.3b

$$\begin{aligned} \text{Sigmoid} \ : \ \mathbb{R} &\to]0,1[\\ x &\mapsto \sigma(x) = \frac{1}{1+e^{-x}} \end{aligned} \tag{2.1.7}$$

The sigmoid is often used as activation function for the output layer, in the case of binary classification.



(a) ReLU (b) Sigmoid (c) Hyperbolic tangent

Figure 2.3: Different activation functions

- **Hyperbolic tangent:**
  This function is also on-linear, but it bounds the outputs between -1 and 1. Its mathematical definition is described in Equation (2.1.8), and the function is represented in Figure 2.3c

$$\begin{aligned} \text{Hyperbolic tangent} \ : \ \mathbb{R} &\to]-1,1[\\ x &\mapsto \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \end{aligned} \tag{2.1.8}$$

  It can be used in hidden layers sometimes, especially when values need to be centered around zero.

- **Other functions:**
  There exist many other activation functions, that aim to make the precedent ones better in some specific situations, such as the Leaky ReLU, or the Softmax for multi-class classification. Google also developed its own Swish [33] as alternative to ReLU, defined by $f(x) = x \cdot \sigma(x)$ where σ is the sigmoid function. However, this one affects computational complexity a lot compared to ReLU.

2. Fully Connected layer

The fully connected (FC) layer is what we named previously the 'full perceptron' (Figure 2.2). The output is computed $\boldsymbol{y} = h(\boldsymbol{\omega} \cdot \boldsymbol{x})$ where $\boldsymbol{x}$ is the input vector, $\boldsymbol{\omega}$ the weight vector and h the activation function. Its key characteristics are:

- Dense connectivity: Every neuron in one layer connects to every neuron in the next.
- High parameter count: Because of full connectivity, FC layers have a large number of parameters (weights and biases).
- Computationally expensive: Due to the high number of connections, FC layers require significant computation and memory.
- Used for feature aggregation: Often placed at the end of CNNs to flatten and combine extracted features for classification.

In the most recent popular architectures that can be found in the literature, the FC layers are often used in CNNs, to make final predictions. It will also be the case in the ResNet architecture that we'll describe below.

3. Convolutional layer

The convolutional layer is a very common architecture type for *computer vision* tasks. The idea is to slide a small filter (or kernel) over the input image or feature map. Each filter detects a specific pattern (e.g., horizontal edges, vertical edges, textures), and as represented in the Equation (2.1.9), an activation function (often ReLU) is applied. The process creates a new feature map, where detected patterns are highlighted, and multiple filters are used to detect different features.

$$\boldsymbol{y} = \mathrm{ReLU}(\boldsymbol{\omega} * \boldsymbol{x}) \tag{2.1.9}$$

Please note: here $*$ is the convolution operation.

Key characteristics of the convolutional layer are:

- Local Connectivity: Neurons in the feature map connect only to a small region of the input.
- Shared Weights: The same filter (weights) is applied across different parts of the input, reducing parameters.
- Translation Invariance: Detects patterns regardless of their position in the image.

Convolution layers can use kernel of different dimensions (1D, 2D, 3D) to match the shape of the data (respective examples: sequence; image; video or 3d medical imaging inputs). Such layers are often combined one after another, in order to detect more complex patterns.

To precisely define a convolutional layer, we need to define a few parameters:

- Number of input channels: it is typically useful for an image analysis .If the image is colored with a scale of grey, we'll consider only a single channel (often a value encoded on a byte, between 0 and 255). However, if we compute a colourful picture, we'll consider 3 channels (for RGB, the level of each color for every pixel).

- Number of output channels: it will determine the number of kernels we want to train.

- Kernel size: it is often a square or a cube (depending of the input dimension), so we usually define only the size of one edge.

- Stride: the number of units to move every time we make the kernel translate. If the stride is set to 1, the kernel will go on every possible position; if it is set to 2, the kernel will translate 2 pixels horizontally or vertically after each position.

- Padding: the number of 'outside' borders to consider around the input. It can be useful to detect patterns located at the edges of a picture.

It is important to set the stride and padding in a coherent manner according to the input size. Also, the kernel size must obviously be smaller than the image size (except if there is enough padding, but this is a very rare use case).

4. Pooling layer

The pooling layer is a type of layer in CNNs that is used to reduce the spatial dimensions (height & width) of feature maps while preserving important information. This helps to reduce computation, improve generalization, and make models more robust to small variations in input images. It is particularly useful to:

- Reduce computational cost: Fewer parameters and operations.

- Prevent overfitting: Force the network to extract only the most essential features.

- Provide translational invariance: Detect patterns regardless of small shifts or distortions.

- Enhance feature representation: Focus on dominant features while removing noise.

There are two different kinds of pooling: the max pooling (MaxPool) and the average pooling (AvgPool). Figure 2.4 explains their behavior on a simple example.

Both techniques have their pros and cons, and are used for different contexts. The Table 2.1 sums this up.

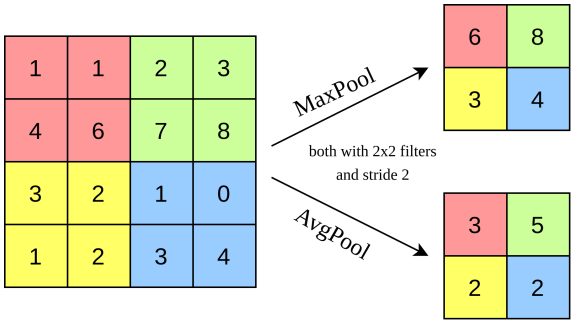


Figure 2.4: MaxPool and AvgPool

Feature	Max Pooling	Average Pooling
Keeps strongest features	Yes	No
Works well for CNNs	Yes	Yes
Used in feature extraction	Yes	Yes
Reduces noise	No	Yes
More aggressive reduction	Yes	No

Table 2.1: Comparison of Max Pooling and Average Pooling

This means that the max pooling is mainly used to detect important features, especially in computer vision to recognise edges and shapes. On the opposite, the average pooling is used if preserving general spatial information is more important.

5. Batch Normalization layer

The Batch Normalization (BatchNorm) layer is a technique used to normalize activations in a neural network. It improves training speed, stability, and generalization by reducing internal covariate shift.

A *batch* is a subset of all training samples, used to train the model. The process of fitting the model is done successively on all batches. That means that the smaller the batch, the more the number of fit passes, and the longer the training.

BatchNorm is particularly useful because for:

- Stabilizing Training: Prevents large activation changes, making training more stable.
- Faster convergence: Reduces the number of epochs needed to train deep networks.
- Regularization effect: Acts as a weak form of regularization (concept of avoiding to much variance among weights).

- Allowing higher learning rates (LR): Helps avoid exploding or vanishing gradients.

Even though it is not the only normalization technique used in NNs (normalization could be applied at other scopes, such as across features in a layer, or across spatial dimensions, or across channels in group).

2.1.3 Learning Process

In the context of this thesis, we only explain the *supervised learning*. In this framework, we dispose of a dataset consisting of p data samples $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(p)}$ alongside as many targets $\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(p)}$. For $i = 1$ to p, $\boldsymbol{y}^{(i)}$ is the true value that we want to predict with the NN, knowing $\boldsymbol{x}^{(i)}$.

To make this clearer, let's take an example: we dispose of a dataset of pictures, each one 32x32 pixels and the color is a scale of grey, which means that the input $\boldsymbol{x}$ has $32 \times 32 \times 1 = 1024$ channels. We also know that all these pictures represent a digit (from 0 to 9). Then for a picture of a '6', we want to give it to the NN and that it returns either the single value $z = 6$, or a vector $\boldsymbol{z} = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)$ for a *one-hot* representation, or any other representation that clearly states that the picture shows a 6.

In a general manner, in the context of supervised learning, the model in trained on a labeled set, adjusting its parameters to minimize the difference between its predictions and the true values (targets). The aim is to synthesize this learning process, so that the MLP gives accurate predictions on new, unseen data. This can be summarized in 3 steps:

1. **Get the labeled training dataset**

The device will fetch $q(< p) \in \mathbb{N}$ labeled data samples. Usually, the share of samples used for training represents around 80% of the total dataset. The rest is used for testing. This means that here, $\frac{q}{p} \approx 0.8$.

A 'good' dataset for machine learning research purposes is a dataset that **contains enough samples** (which is a vague definition, but it depends a lot on the context), to allow the model to generalize well and be able to prove its performance. It also obviously needs to be **adapted to the model** that we want to train, like for example having medical data (heart rate, blood pressure, symptoms, ...) and not random images, for a model that aims to predict diseases. Moreover, still in the context of supervised learning, we want the dataset to be **correctly labeled**, in order to avoid counter-productive training. Last but not least, as for FL the dataset is distributed among clients that don't communicate with each other, we can consider cases where the **dataset is independently and identically distributed (IID), or not**.

Different partitioners can simulate these behaviors, such as the latent Dirichlet allocation (LDA), which uses an hyperparameter α to part the samples among clients. For LDA, the bigger α is, the more IID is the data. In our work, we will use some well known datasets of image classification, such as the CIFAR-10 and the CIFAR-100, respectively with 10 and 100 different target classes, or ImageNet with more than 20,000 categories. The EMNIST dataset also presents a big interest for the same purpose, and can also used to test the model on different datasets. This one is an extended version of the MNIST, introduced by Yann Le Cun in 1998, consisting of about 70,000 samples of handwritten digits. The EMNIST implements letters in addition.

2. **Train the model**

Training the model means that we want to adjust the initial given weights to fit better the training data. It is done applying the *stochastic gradient descent* (SGD) method. The gradient descent consists of trying to minimize a loss function $\mathcal{L}$ by iteratively updating the parameters $\boldsymbol{\omega}$ of the NN, according to their derivative $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\omega}}$. The stochastic version of the gradient descent consists on doing each iteration on a sub-domain of the global training set, called *batch*. Usually, during learning phase, we add to $\mathcal{L}$ a regularization term, called *weight-decay*, which applies a penalty $\mu\|\boldsymbol{\omega}\|_2$ to the parameters' magnitude.

3. **Test the model**

To test the model, we apply the NN to the $p-q$ remaining samples, and calculate predetermined metrics (for example the final/average size of the model). A very important one is the accuracy: for each $i \in (q+1, \dots, p)$, we consider $\boldsymbol{x}^{(i)}$ the test data samples, $\boldsymbol{z}^{(i)}$ the associated output values calculated by the NN, and $\boldsymbol{y}^{(i)}$ the true target value associated with each sample. The accuracy is defined as the share of correct predictions made by the network. Put in mathematical terms, we can define the function

$$\begin{aligned} \mathbb{1}_{\boldsymbol{x}} :\ & \mathbb{R}^m \times \mathbb{R}^m \to \{0,1\} \\ & (\boldsymbol{y}, \boldsymbol{z}) \mapsto \begin{cases} 1 & \text{if } \forall k \in (1, \dots, m),\ y_k = z_k \\ 0 & \text{else} \end{cases} \end{aligned} \tag{2.1.10}$$

so that the accuracy is easily written as in the equation (2.1.11)

$$\text{accuracy} = \frac{1}{p-q} \sum_{i=q+1}^{p} \mathbb{1}_{\boldsymbol{x}^{(i)}}(\boldsymbol{y}^{(i)}, \boldsymbol{z}^{(i)}) \tag{2.1.11}$$

For example, accuracy values of popular models trained on different datasets are often more than 80%. Indeed on the ImageNet dataset, AlexNet performed in 2012 at a 16.4% top-5 error rate, then in 2014 the VGG16 pushed it down to 7.3%, and in 2015 the ResNet-50 and Inception v3 had respectively 3.6% and 3.5% top-5 error rates. On the CIFAR-10, VGG6 can reach 85% top-1 accuracy, when VGG16 or ResNet-18 achieve around 92%.

Note: the top-5 error rate means that the correct target is not among the 5 outcomes determined as 'most probable' by the NN. It is relevant to evaluate classification workers.

These metrics give insights on the quality of the training, typically if the model is *overfitting* the training data and not being accurate on new samples. The metrics can be used to accept or reject the proposal of model.

2.1.4 Specific CNN Architectures

Theoretically, we can build any neural network given input and output sizes, and then train it and evaluate how it performs. However, like we've seen previously in the description of the main types of layer existing, they often bear a strategy in order to solve a specific task/issue. That means that when building a neural network, we can often wisely choose which layer to use where. Thus, in this section we'll describe a few common architectures that will grasp our interest later on, and that have already proved their performance in the scientific literature.

AlexNet

AlexNet, introduced by Krizhevsky et al. in 2012 [21], marked a breakthrough in deep learning for computer vision by winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), lowering the top-5 error rate to 16.4% when the next best approach had a 26.2% top-5 error rate. It demonstrated the power of deep CNNs in large-scale image classification, significantly outperforming traditional methods. The architecture consists of eight layers, including five convolutional layers followed by three fully connected layers, utilizing ReLU activations, max pooling, and dropout for regularization. The Figure 2.5 visually describes this architecture.

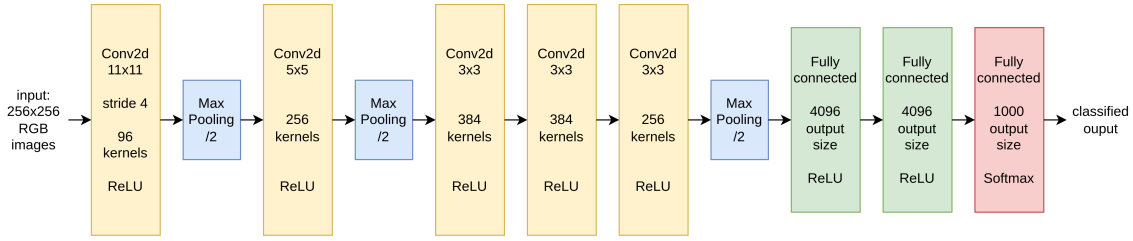


Figure 2.5: AlexNet original architecture

The developers of this network tried to remove any convolutional layer, but each time the performance of the network was affected. Notably, AlexNet leveraged graphical processing unit (GPU) acceleration to efficiently train on massive datasets, setting the foundation for modern deep learning architectures.

Other similar CNN architectures have been since then developed, such as the VGGNet in 2014/2015 with 11 to 19 layers (VGG16 is the most widely used), which improved ILSVRC top-5 error rate to 6.7%. It is important to note that the training of deeper NNs is made possible by the improvement of computation power, especially with the GPUs.

Residual Networks (ResNet)

ResNets are a type of deep CNN designed to tackle the vanishing gradient problem, which occurs when training very deep networks. They were introduced by Microsoft Research in 2015 [15] and have since become one of the most widely used architectures in deep learning, especially for image classification and recognition tasks.

The idea of ResNet is to introduce residual connections (or skip connections), which allow the network to learn residual mappings instead of trying to learn the full transformation directly. This means that instead of each layer learning a full representation, it only needs to learn the difference (residual) from the input. The key component is the residual block, represented in Figure 2.6, where $\boldsymbol{x}$ is the input, $F(\boldsymbol{x})$ the transformation applied by convolutional layers, $\boldsymbol{y} = \boldsymbol{x} + F(\boldsymbol{x})$ the final output.

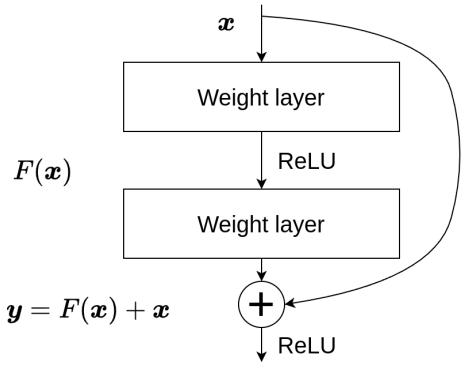


Figure 2.6: Residual block

Several ResNet variants exist, each with different depths and applications. The Table 2.2 gives an overview of the most popular ones. Later on, in the Chapter 5, we will use the ResNet-18. Its original architecture is described in Figure 2.7. It consists of 17 convolutional layers, including shortcuts, followed by a fully connected layer with Softmax activation function. These 18 layers give this model its name. In some more recent revised architectures, some layers have been modified, and some pooling layers have been added.

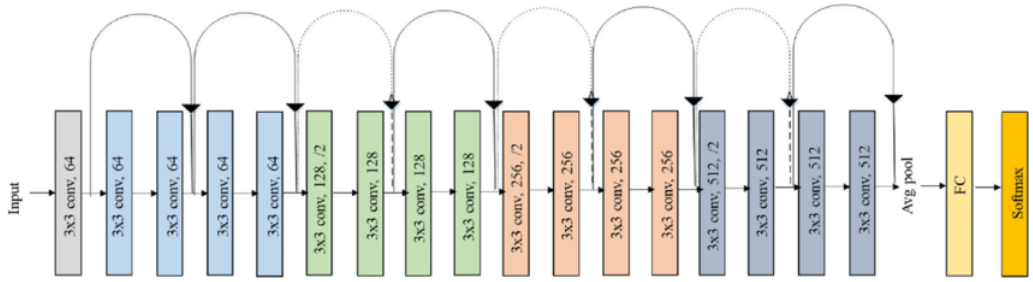


Figure 2.7: Architecture of the ResNet-18, from [34]

Model	Depth	Parameters	Top-1 Accuracy on ImageNet	Use Case
ResNet-18	18	∼11M	∼69.8%	Lightweight applications, real-time processing.
ResNet-34	34	∼21M	∼73.3%	Balanced trade-off between speed and accuracy.
ResNet-50	50	∼25M	∼76.2%	Commonly used for transfer learning in real-world applications.
ResNet-101	101	∼44M	∼77.4%	More complex tasks requiring deeper networks.
ResNet-152	152	∼60M	∼78.3%	High-performance applications with sufficient computational resources.

Table 2.2: ResNet Models Overview and Performance

2.2 Federated Learning

2.2.1 Introduction

Introduced in 2016 by MacMahan et al. [26], FL provides a groundbreaking approach to ML. In the era of AI where we need always more precise, diverse, private, and most of all numerous data, and taking into account that the number of devices connected to internet is unceasingly growing, the idea emerged to train ML models with multiple edge devices, rather than centralizing the process on single servers.

By nature, FL tackles issues of privacy and security of the data. Its setup in the context

of this thesis will be the following, as depicted in Figure 2.8. We dispose of a server,

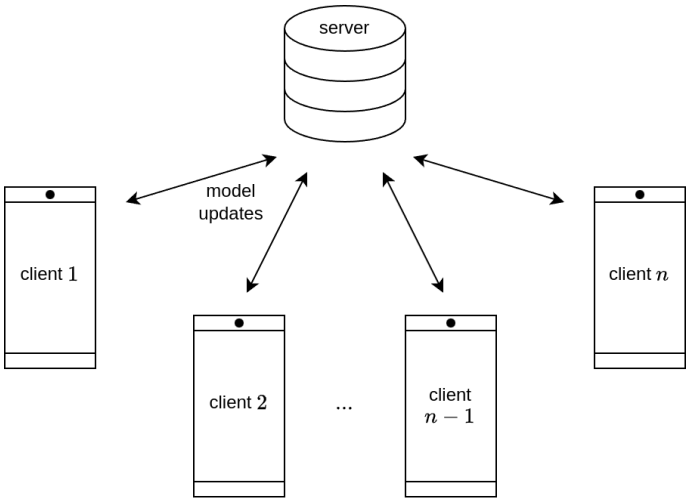


Figure 2.8: Representation of a FL setup

which 'wants' to train a model, a NN. The server 'knows' a certain amount of clients, who all individually dispose of some data useful for training the model. The process of FL learning is the following: for the first *round*, the server sends the untrained model to some clients (usually not all). These clients all individually train the model, and send back to the server their respective updates. The server *aggregates* all the received updates (how to is explained in Section 2.2.2), in order to create a global model. Then the same starts again for the second *round*, where a new random sample of clients is selected, who all train the new model and return its update, and so on until the server is happy with its trained NN. Figure 2.9 shows an example of a training over three rounds. To evaluate when convergence is reached, the server can ask the clients for accuracy scores.

The design of a FL setup must take into account many parameters, such as the computational power of the client devices (also named *edge devices*), the quality (incl. quantity) of their dataset, the capacity of communication channels, the number of rounds we want to train on, the number of epochs that client should locally apply at each round. Other hyperparameters of classic ML (e.g. learning rate, optimization goal function) are also to be taken into account.

Literature [31] cites three main architectures for FL: vertical, horizontal, and transfer federated learning. To make the explanation clearer, we give the following example: a company want to train its NN on the data of a village inhabitants. The company doesn't own the data, but some shops do (customers accepted to give them some personal data, but didn't accept it to be shared). All three situations:

- **Horizontal FL:** all clients share the same feature space, but different sample spaces. In our example, the bakery and the library both record the name, age, address, phone

number, ... of their customers, but they have different customers.

- **Vertical FL:** all clients share different feature space, but the same sample space. In our example, the library and the bank share the same customers, but don't record the same data: the library knows their preferred genre, when the bank owns data about their credit card and fiscal information.

- **Transfer FL:** all clients share different feature space, and different sample space. This is more of a real-life situation: all shops don't record the same information about their customers, who themselves go to only a share of all shops.

In the context of this thesis, we will only consider the case of horizontal FL.

2.2.2 Usual Aggregation Strategies

One of the biggest issues to resolve when executing a FL process is the question of *aggregating* the different updates sent by the clients. Because it directly affects the model at each round, the *aggregation strategy* occupies a crucial role in the process of FL.

When introducing FL, McMahan suggested to easily average the clients' updates. Let C be the set of clients, and note n_c the size of the dataset (number of samples) for each client $c \in C$. At the round i, the server selects the set $S \subset C$ of clients that'll train the model. In the FedAvg strategy, this subset S is randomly selected. For each client $s \in S$, we note $\omega_s^{(i)}$ the model update it provides to the server after training at round i. Then the aggregated result is:

$$\omega_{\text{global}}^{(i)} = \sum_{s \in S} \frac{n_s}{n} \omega_s^{(i)} \quad \text{where} \quad n = \sum_{s \in S} n_s \tag{2.2.1}$$

Even though its simplistic character, the FedAvg strategy proved itself to be really efficient.

Other methods try to address some issues of the FedAvg approach. For example, FedProx [23] introduces a regularization (proximal) term $\frac{\mu}{2} \left\| w_k^t - w_{\text{glob}}^t \right\|_2^2$ (where μ is the penalty constant of the proximal term) to the learning process of clients. It ensures that they don't learn too much for themselves instead of allowing the global learning.

Survey [31] also cites other methods such as FedNova, Scaffold, MOON, Zeno or PerFedAvg, but in the context of this thesis we'll only consider FedAvg.

2.2.3 Challenges

Federated Learning presents numerous challenges that span across multiple dimensions, including efficiency, security, fairness, and system scalability. Since its introduction in 2017 [26], researchers have been addressing key limitations that hinder its widespread adoption [18]. Several reviews [22, 46] provide a comprehensive state-of-the-art analysis of these challenges.

One major challenge is improving the efficiency and effectiveness of FL systems. Communication constraints, heterogeneity in data distributions, and resource limitations across participating devices impact model convergence and performance [22]. Optimizing aggregation strategies and designing more robust algorithms remain open research problems.

Security and privacy are also critical concerns in FL. Since raw data remains decentralized, adversaries may exploit vulnerabilities through inference attacks, model poisoning, or privacy leakage risks [18]. Techniques such as secure multi-party computation, differential privacy, and homomorphic encryption are being explored to mitigate these risks.

Another challenge is ensuring fairness and addressing sources of bias in federated models. Unequal representation of different user groups or device capabilities can lead to biased outcomes, affecting model generalization and fairness in real-world applications [46]. Strategies for mitigating bias include personalized learning approaches and fairness-aware optimization techniques.

Lastly, system-related challenges, such as scalability, heterogeneity in device participation, and resource constraints, remain open problems. Efficient scheduling mechanisms, adaptive learning protocols, and federated optimization techniques are actively being developed to address these issues [22, 26].

Addressing these challenges is crucial for the successful deployment of FL in real-world applications. Continued research efforts are essential to enhance the robustness, security, and fairness of federated systems while maintaining efficiency and scalability.

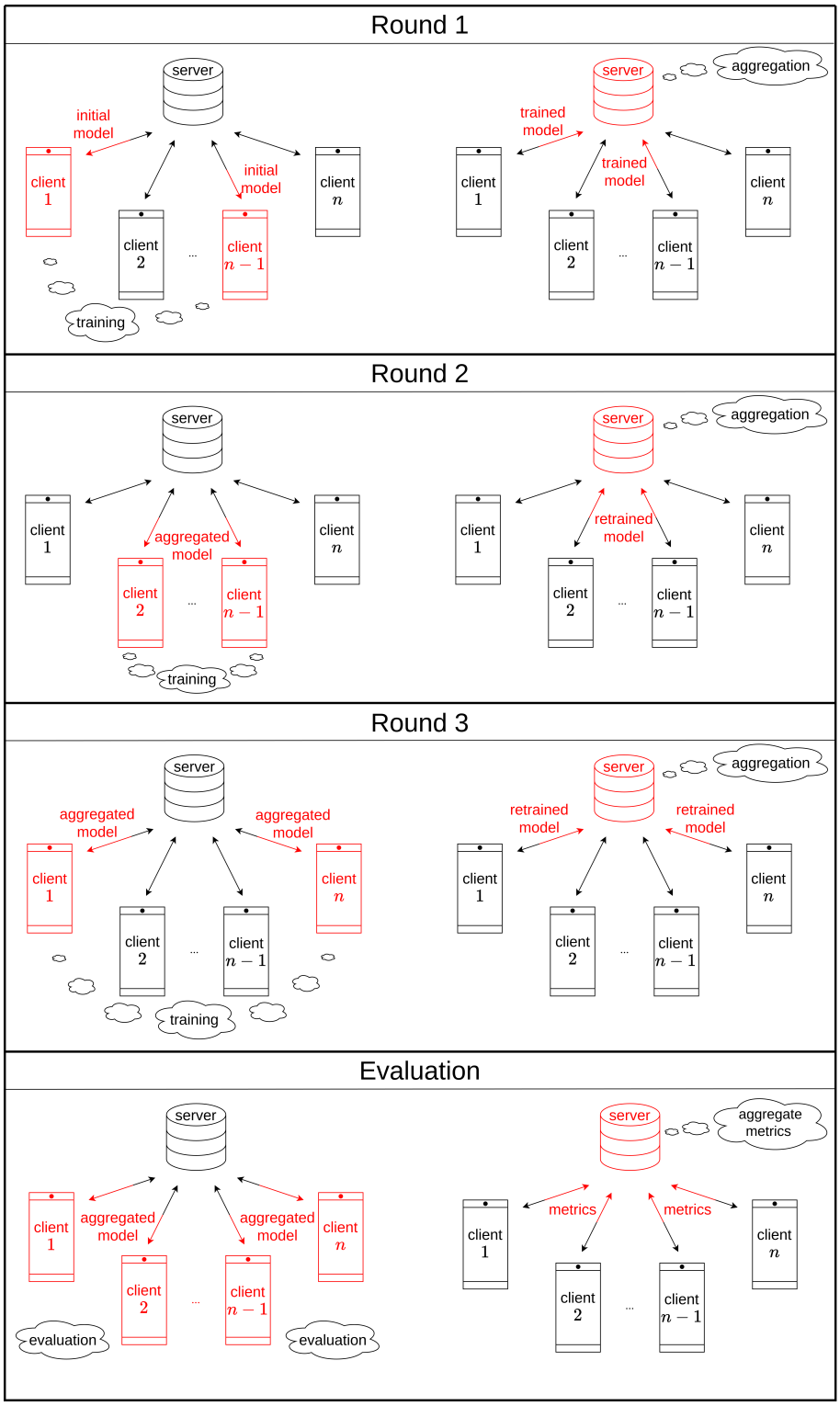


Figure 2.9: Example of FL learning process over 3 rounds

2.3 Deep Compression Techniques

DNNs have achieved remarkable success in various machine learning applications, but their deployment on resource-constrained devices remains a significant challenge due to high computational and memory requirements. To address this issue, Han et al. introduced *Deep Compression* in 2016 [14], a technique that reduces the storage and computational costs of deep networks without compromising accuracy.

Deep Compression is a three-stage pipeline consisting of *pruning*, *quantization*, and *Huffman coding*. The pruning step removes a certain amount of connections, significantly reducing the number of parameters while preserving model performance. Quantization further compresses the network by reducing the precision of weights. Finally, Huffman coding is applied to further reduce storage costs by exploiting weight distribution characteristics.

By combining these techniques, Deep Compression achieves up to 49× model size reduction and significant improvements in energy efficiency, enabling the deployment of deep networks on edge devices such as mobile phones and embedded systems. This approach has inspired further research in model compression, hardware-aware deep learning, and efficient inference techniques, shaping the future of resource-efficient AI.

2.3.1 Pruning

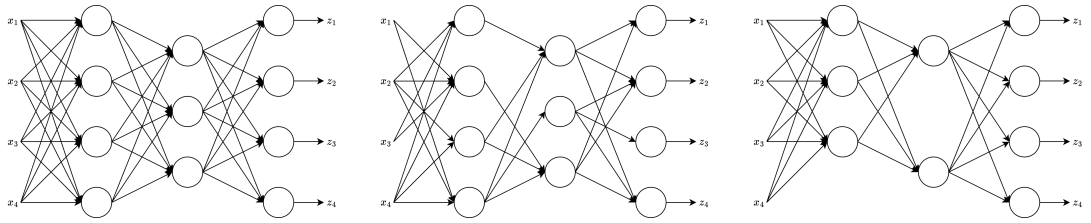
Pruning is a model compression technique that reduces the number of parameters in deep neural networks by removing redundant or insignificant connections. By iteratively eliminating weights and fine-tuning the network, pruning significantly reduces storage requirements and accelerates inference, making deep learning models more suitable for deployment on resource-constrained devices.

As discussed by Rachwan et al. [32], there exist different approaches for pruning:

- **Weight Pruning:** Removes small-magnitude weights in NN.
  Also named *unstructured pruning*, this technique allows to eliminate a precise amount of weights, usually $x\%$ of them, therefore we have a total control on the result compression of the model. Proposed by Han et al. [14], pruning aims to enhance computational efficiency and reduce memory usage while maintaining model accuracy. We will be able to evaluate its performance later on in this paper, as we use this technique in our experiments. Figure 2.10b gives an example of how this could look like when pruning 30% of all weights (12 out of 40 here) in a MLP.

- **Structured Pruning:** Removes groups of weights.

This technique also enables efficient compression, however this time whole weights structures (neurons) are removed in the final result. Even though the philosophy is different, the goal stays the same: reduce memory usage while maintaining model accuracy. Figure 2.10c gives an example of how this could look like when pruning 2 out of 11 neurons in a MLP.



(a) Example network (b) Unstructured (30% rate) (c) Structured (2/11 neurons)

Figure 2.10: Different types of pruning on an example network

The main difference between structured and unstructured pruning is how we handle the representation of their sparsity in memory.

2.3.2 Quantization

Quantization is a technique used to reduce the memory footprint and computational cost of DNNs by representing model parameters with lower precision numerical formats. This is crucial for deploying deep learning models on edge devices with limited hardware resources [14, 13].

Quantization can be categorized into several approaches based on how weights and activations are mapped from high-precision (e.g., 32-bit floating point) to low-precision formats (e.g., 8-bit integers).

- **Uniform Quantization:** In uniform quantization, the real-valued weights are mapped to discrete values using a uniform step size. The quantized values are obtained using the following formula:

$$q(\boldsymbol{\omega}) = \text{round}(\frac{\boldsymbol{\omega} - \omega_{\min}}{\Delta}) \tag{2.3.1}$$

  where:

  - $\boldsymbol{\omega}$ is the original weight structure,
  - $\omega_{\min}$ and $\omega_{\max}$ are the minimum and maximum values in the weight range,
  - $\Delta = \frac{\omega_{\max} - \omega_{\min}}{2^b - 1}$ is the quantization step size for b-bit representation.

Uniform quantization is simple and hardware-friendly but may introduce large errors for weights with non-uniform distributions [17].

- **Non-Uniform Quantization:** uses variable step sizes, often following a logarithmic or learned distribution. This approach is beneficial for distributions where smaller weights occur more frequently, as seen in neural networks [27]. A common technique is logarithmic quantization, where weights are approximated as powers of two: $q(w) = \text{sign}(w) \times 2^{\text{round}(\log_2 |w|)}$. This allows efficient multiplication using bitwise shifts, reducing computational cost.

- **Post-Training Quantization (PTQ):** reduces precision after model training without retraining. It involves min-max scaling or histogram-based quantization to map floating-point weights to lower-bit integers [20]. While fast and efficient, PTQ may degrade model accuracy due to quantization errors.

- **Quantization-Aware Training (QAT):** simulates quantization during training, allowing the model to adapt to lower precision. During forward passes, weights and activations are quantized using a fake quantization operator, while full precision is retained during backpropagation:

$$q(\omega) = \text{round}(\omega/\Delta) \times \Delta \tag{2.3.2}$$

  where Δ is the quantization step size. QAT often yields better accuracy than PTQ but requires additional training time [11].

- **Adaptive and Learned Quantization:** Some approaches use optimization techniques to determine quantization parameters dynamically. Vector Quantization (VQ) clusters weights using **k-Means clustering**, mapping each cluster to a quantized value [12]. More advanced methods, such as differentiable quantization, learn optimal bit-widths during training [24].

2.3.3 Huffman Encoding

Huffman encoding is a lossless compression technique employed in deep neural network compression to reduce storage and memory overhead while preserving accuracy. It assigns variable-length binary codes to weight values based on their frequency, with shorter codes for more frequent values and longer codes for rarer ones.

In deep compression pipelines, Huffman encoding follows pruning (removal of redundant weights) and quantization (reduction in numerical precision), further reducing the model size. By encoding frequently occurring quantized weights with fewer bits, Huffman encoding optimizes storage, minimizes memory bandwidth requirements, and enables efficient deployment on resource-constrained devices without loss of information.

Given a set of symbols $S = \{s_1, s_2, ..., s_n\}$ with corresponding probabilities $P = \{p_1, p_2, ..., p_n\}$, the Huffman encoding algorithm proceeds as follows:

1. Construct a priority queue containing all symbols, where each symbol's priority is its probability p_i.

2. While more than one node remains in the queue:

   a) Remove the two nodes with the smallest probabilities.

   b) Merge them into a new node with probability:

   $$p_{\text{new}} = p_i + p_j$$

   c) Insert the new node back into the priority queue.

3. Assign binary codes to the symbols by traversing the final Huffman tree, with '0' assigned to the left branch and '1' to the right.

The expected length of a Huffman-coded message is given by:

$$L = \sum_{i=1}^{n} p_i \cdot l_i$$

where l_i is the length of the Huffman code for symbol s_i. The optimality of Huffman encoding is derived from minimizing this expectation.

3 Related Work

In this chapter, we will discuss the related work. Section 3.1 reminds us the energy-efficiency challenges faced when working with FL, detailing a few relevant aspects for our work. Then, Section 3.2 reviews recent work on the field of deep compression applied to federated fearning.

3.1 Green AI & Federated Learning

Green AI is a term introduced by Schwartz et al. [38], referring to the idea of an AI research that could be 'more environmentally friendly and inclusive'. It distinguishes itself from the *Red AI* which aims to develop AI research without worrying about computational costs and carbon footprint.

Models are always larger, with the aim to try to increase their performances, like the GPT-3 released by OpenAI with 175 billion parameters. For this model, the training lead to a total energy consumption of 1297 MWh, equivalent to 552 tons of CO_2 emissions (tCO_2e) [29]. Considering that this is equivalent to 10 to 25 times the lifetime emissions of a car (34.2 to 50.4 tCO_2e for an internal combustion engine vehicle, or 22.5 to 26.8 tCO_2e for an electric vehicle, during its lifetime, numbers calculated according to the IPCC 2022 report [43], and energy production in Germany in 2024 [16]), this leads to a reflection about the issue of tackling this inducted pollution.
Side note: a tCO_2e (ton of equivalent CO_2) is defined by the IPCC for any greenhouse gas, as the mass of CO_2 which would warm the earth as much as the mass of that gas [8].

A paper by Salehi and Schmeink [37] analyzes the relations between *Green ML* and *Data-centric ML*, leading to the idea of a *Data-centric green ML*. In the paper are described a few frameworks as applications of this field: active learning, knowledge transfer/sharing, dataset distillation, data augmentation, curriculum learning. Although these frameworks aren't directly related to our work, they could be combined with deep compression techniques to enhance energy-efficiency without affecting performance.

A more specific focus of this thesis is the application of green AI in federated learning, which has gained significant attention in recent years. Indeed, many papers offer strategies

to optimize energy consumption, such as [4] with an energy-aware client selection. It allows to optimize the trade-off between selecting the maximal number of clients and ensuring that those clients' batteries have sufficient energy to compute the update of the model and upload it.

3.2 Deep Compression Techniques in Federated Learning

Compressing in FL offers mainly two big advantages:

- **Reduce the model size:** this is already the aim of deep compressing in the more general context of ML. We reduce the size of the model so that is it more lightweight after training, thus forward passes can be hardware and/or software optimized. However, because of the way the FL works, compression has further interest: for each round, the NN we train must be communicated between the server and the clients. This means that each improvement for accelerating communication matters for the training phase. Indeed, for a constant power usage, the less the computation required to send and receive models, the less the energy consumption.

  For this extent, we also want to introduce the concept of *communication overhead*: it is defined at the share of time spent for communication among the total time spent for the training, and is a key point in FL [45, 41]. It can then be easily calculated per round as the time not spent for computation (training, evaluation, compressing) on the model, over the time taken by the whole round.

  The communication overhead impacts FL especially when the server receives all client updates. This is because as depicted in Figure 3.1, all client updates arrive at (almost) the same time at the server. All updates require *bandwidth*, and too many needs induce a *bottleneck* effect, which negatively impacts the communication performances (especially in time, so also in energy usage) of the training. On the contrary, broadcasting the model is less impactful, because the same weights are sent to all selected clients, who all require it only once.

  Compressing at the clients (*upstream compression*) or at the server (*downstream compression*) can be applied to reduce the model size and thus the communication cost [40].

- **Enhance computation abilities:** thanks to particularities of compression techniques used, the model can be used more efficiently. For sparse (pruned) models, some representation allow to compute outputs faster. It is the same for quantization, where both software and hardware acceleration can be implemented.

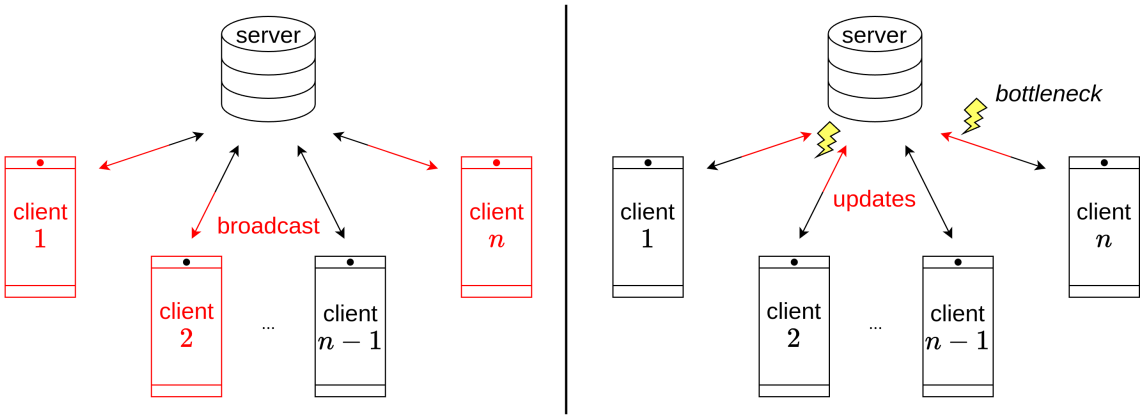


Figure 3.1: Bottleneck illustration at the server entry point

However, it is important to keep in mind that the whole processing pipeline also induces more computational pressure [47], especially on edge devices. Finding the best trade-off between the inherent computing power required for compressing and its induced reductions and optimizations, is a major research topic in FL.

The deep compression techniques described in Section 2.3 were originally designed for a classic ML case [14]. However, the compression pipeline can be adapted to FL, as we'll see in the next three sections.

3.2.1 Pruning in Federated Learning

The implementation of pruning has to be adapted to FL. Indeed, some aspects of this compression technique have to be refined to adapt it better to the distributed case. Particularly:

- **Bi- or Unilaterality:** as of the design of communication in FL, the model updates are successively downloaded and uploaded between the server and the clients. In each direction, it is needed to decide if the model is compressed — here pruned — before communication or not. It is reported in [25] that in the literature, the compression route is mostly either bidirectional or on the upstream, almost never on the downstream only. This is because during broadcast (downstream route), the size of the model — and so the volume of communication — is less important than in the other direction, where the bottleneck is a problem. It it then relevant to implement the compression techniques in priority at the clients.

- **Types of pruning:** as seen in Section 2.3.1, we distinguish structured and unstructured pruning. It is important to note that both aren't incompatible, as explains

[44] which implements a 'hybrid pruning' which basically consists of applying structured pruning to filters of convolutional layers, and unstructured pruning to fully connected (linear) layers. However, the most popular method is the 'magnitude' pruning, other name for unstructured, which is the one introduced by [14], sometimes followed by a fine-tuning on the remaining weights [5]. Strategies vary between definitely removing all pruned weights, for the aggregation and further on for all the training [44], or just aggregate the client updates without taking into account which weights/structures were pruned. The latter is much easier to use in practice, as the model structure remains the same at every point in the code [35].

- **Representation for communication:** the pruning process induces a sparsity in the weights of the model. To that extent, some technical solutions exist to primarily optimize computation, and further reduce the representation size [42]. The issue was also addressed in [14], which uses the Figure 3.2 to illustrate their idea: instead of storing all weights, save the value of non-zero weights and the difference of index with the previous non-zero weights. To further enhance this method, they propose to store the index difference on a limited amount of bit (3 in the example), and insert filter null weights where the maximum index difference is reached, so that the sparsity can be consistently recorded.

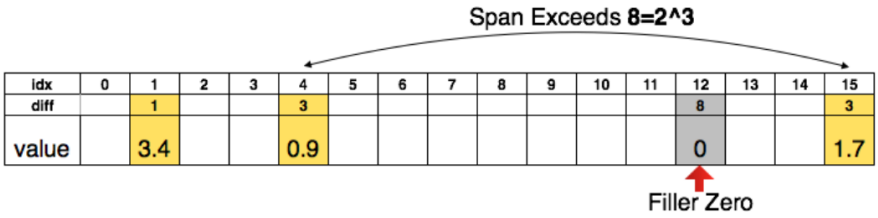


Figure 3.2: Representing the matrix sparsity with relative index. Padding filter to prevent overflow. From Han *et al.* [14]

As we'll see in the following section, the compression can be improved a step further by implementing quantization.

3.2.2 Quantization in Federated Learning

In a similar fashion as for pruning, some aspects of quantization differ or need to be specified in the context of FL:

- **Bi- or Unilaterality:** for the same reasons as for pruning, it needs to be decided if the quantization is applied before the upstream and/or downstream communication. Both upstream and downstream quantization exist in the literature [25].

- **Layer- or Model-wise:** we have to decide if we apply the quantization layer per layer, or only once and for all the model. Note that this isn't an issue for quantizations via reduced-precision representation [35]. As the magnitude of weights can often much vary from one layer to another, it is more usual in the literature to apply a layer-wise quantization [48].

- **Representation for communication:** we can once again distinguish two cases: the reduced precision and the k-Means method for fitting a reduced number of centroids. In the first case, if the technical implementation of FL we use allows to reduce the number of bits for representing and transmitting a float value, it's done; else it needs to be implemented. In the second case, we introduce the concept of codebook: given the repetition of same values among the weights, we replace them with a key identifier, and write alongside a dictionary (*codebook*) that maps the keys to their true values [14]. This is depicted in Figure 3.3, and allows to reduce the memory footprint of the network. The big disadvantage is the induced computation, both for implementing it, and then decode the message to rebuild the model after communication.

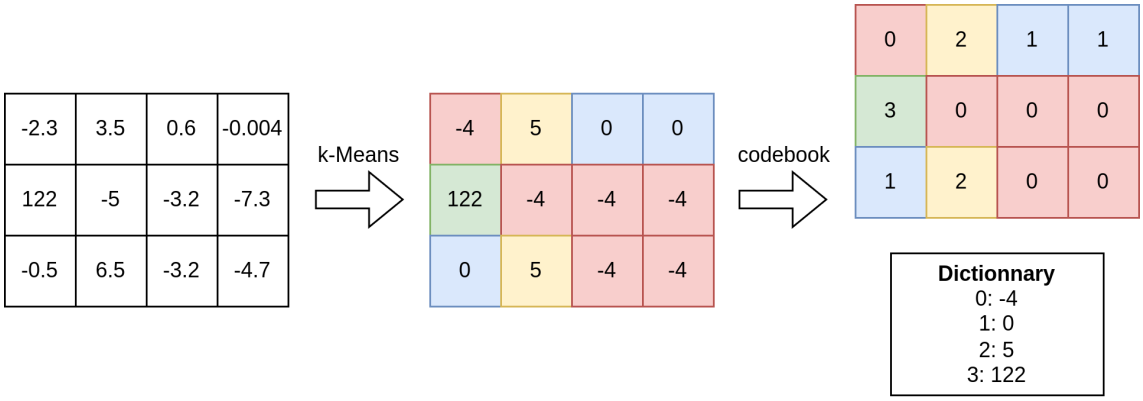


Figure 3.3: Quantization process for optimized memory usage

Now that we've found out important aspects of both pruning and quantization in FL, we can explain how to further enhance these.

3.2.3 Combination of compression techniques in Federated Learning

On the one hand, pruning and quantization can be both used, and combined with Huffman encoding [14]. The latter, described in Section 2.3.3, allows to further reduce the size of model representation, using entropy properties to encode most frequent values with a smaller memory footprint. However, this induces once again a supplementary computation cost, both before and after communication. Depending on the cost of communication, this can be a winning of losing trade-off.

On the other hand, in the literature the combination of such techniques is often not totally exploited, as different techniques tend to be successively applied [25, 35, 48], when we could implement them in a much more efficient manner, meaningfully dividing the number of loop iteration, and thus saving energy and time. This is a main axis of novelty in this thesis.

4 Proposed Framework and Assumptions

In this chapter, we aim to describe the technical aspects and choices that will conduct us to the experiments. The Section 4.1 explains how works our FL framework. Then Section 4.2 precisely describes the implementation of the deep compression techniques previously presented in Chapter 3, and finally Section 4.3 explains some further enhancements.

4.1 Federated Learning Framework

First, the context of FL — key point of this thesis — needs to be specified. For this purpose, we will use the Flower framework [6]. Described by its authors as 'research friendly', we will adapt it to our case.

4.1.1 Architecture

We need to implement a FL job. That means that we need to define a server and a certain number of clients, whose role we also need to define. Implementing this in its entirety would induce a huge workload, and is not the purpose of this thesis, that's why we make use of the Flower framework. We also choose to work with PyTorch [30].

The framework is designed to support customization while maintaining stability of the processes. This is explained because a server job is executed in background and calls predefined functions to make the federated process run. Those functions, described in the sequence diagram in Figure 4.1, can be enhanced to implement our deep compression techniques, and further features we need in our research.

We can distinguish three categories of customizable objects in the framework:

1. **Strategy**
   The strategy is a Python class that is referenced when launching a run. We can use pre-defined classic strategies, such as FedAvg [26], but we can also create our own one. That is the solution we adopt, extending FedAvg by overriding some of its

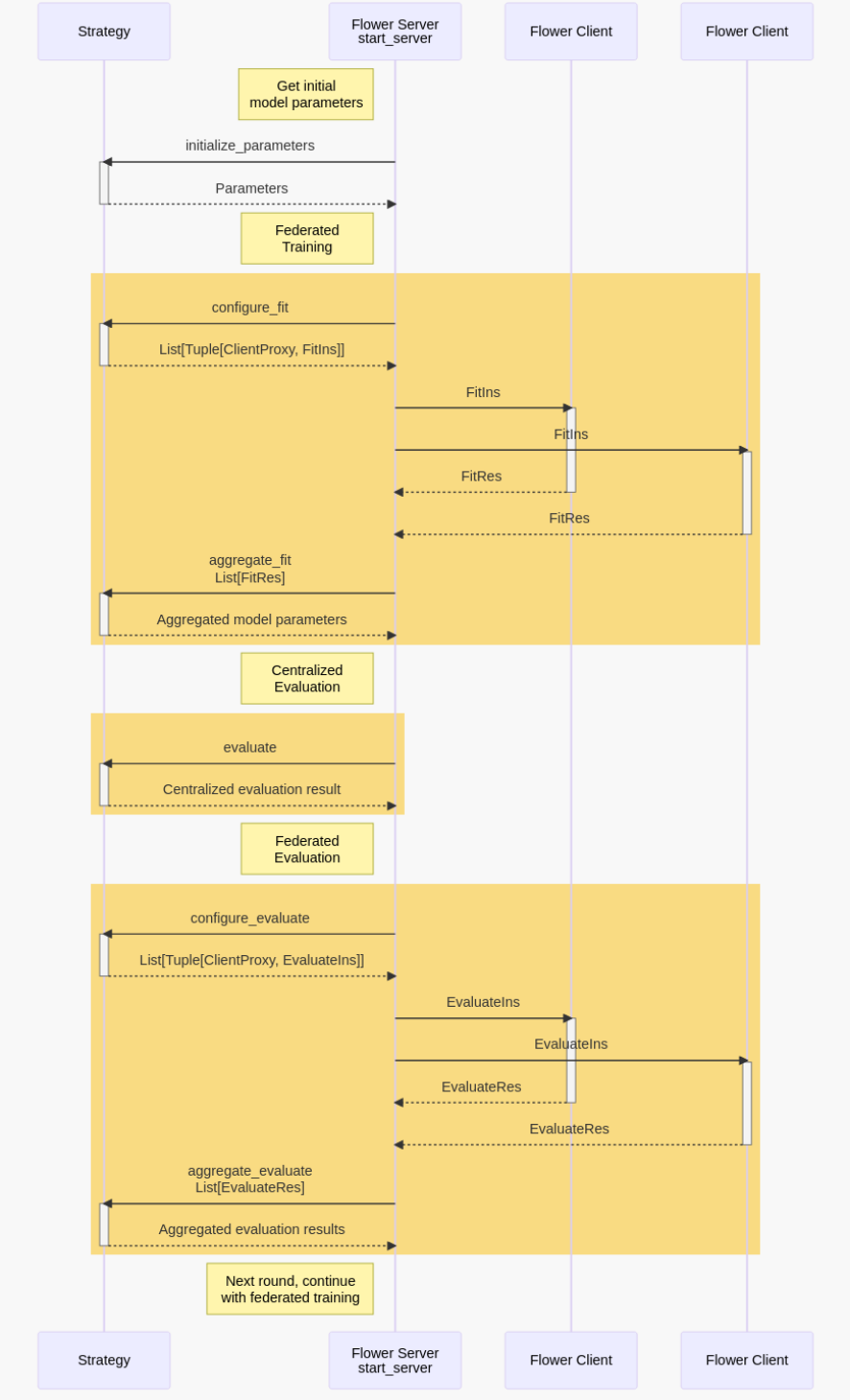


Figure 4.1: Flower strategy baseline [2]

methods.
For each FL round, the strategy is responsible of the two following tasks: select the clients for training, and initialize their call with parameters and configuration; then check validity of the results and aggregate them.
Additionally here in the framework, and for purpose of research, we evaluate several metrics (see next section) at each round to keep track of the performance evolution throughout the learning process. In a normal case of training a NN via FL, we would only need to evaluate metrics (e.g. accuracy) at the end of the whole process, to assert the performance of the trained NN before deployment.

2. **Clients**
The clients are Python classes that simulate the behavior of an edge device. The native framework implements two methods : `train` and `evaluate`, which respectively train model with given weights on a given labeled training dataset, and evaluate it on a labeled testing dataset. Then both methods return their results to the server.
Moreover, in the context of our work we implement deep compression techniques after training and before returning the model to the server. We also keep control on the training algorithm definition (e.g. SGD and its parameters such as learning rate, batch size or momentum) so that customization and different optimizations can be performed.

3. **Hyperparameters**
The framework architecture includes a `pyproject.toml` configuration file which allows not only to set parameters for the simulations, but also to define the project settings (diverse information such as version, author and requirements).
Parameters can be classified in three categories:

- **FL configuration:** the number of clients, the share of clients we use for training at each round, the number of rounds, the architecture of the model (e.g. ResNet-18), the dataset (e.g. CIFAR-10), the partition among clients.
For the latter, Flower initially implements a uniform partitioner. However, we want to be able to set the degree of IID of the distribution of data among clients. In this sense, we replace the former partitioner by a Dirichlet partitioner, as proposed in [35]. As explained before, it distributes the data accordingly to a concentration parameter $\alpha > 0$, and the bigger the α, the more IID the data becomes.

- **Clients configuration:** parameters for learning, like the number of local epochs, the batch size or for the SGD: learning rate, momentum. The fine tuning of these parameters is a key point to reach a maximal energy-efficiency during the whole process of FL.

- **Compression:** the pruning rate, the number of bits used for quantization, and also direction of compression (upstream or downstream). We choose to isolate

all the parameters (e.g. allow client-side pruning and server-side quantization only) to be able to best evaluate the pros and cons of the deep compression techniques in our work.

All of these parametrizations allow us to customize our experimental framework. Moreover, we need to keep track of metrics to evaluate the performances of each run. This is what the next section is about.

4.1.2 Evaluation

Accuracy

The Flower framework allows us to track the **accuracy** (defined earlier in Section 2.1.3) or the related **loss**. This is a good start to track if the global FL run is learning, i.e. have an increasing accuracy and a decreasing loss, however it doesn't allow us to fully evaluate the pertinence of our work on deep compression techniques.

Communication overhead

Therefore, we want to consider a supplementary metric: the **size of the data** we communicate. There are multiple way of measuring it, but we'll focus on two solutions. The first one is the one adopted by [35], using the PyTorch method `torch.save(model, 'directory')`. We can then read the size of the save in a file explorer, and shrink it further via manual compression with a ZIP tool.

However, this is not really representative of how the framework handles data communication. From this point of view, the Huffman encoding provides a really good improvement for transparency. Indeed, the way we implement it (see later in Section 4.2), the clients store directly the compressed data sequences that will be read by the server. Thanks to it, we know exactly the data volume transmitted.

Moreover, we also want to introduce time metrics. As depicted in the Figure 4.2, we measure many different temporal points, so that we can extract:

- **Training time (t_{train}):** time spent on the training only of the model. This time may vary with the size of the model. We decide here not to modify the way the training works, because of how PyTorch recommends to implement it. Indeed, it calls multiple layers of functions applied to the model object, not directly on its weights. We can also note that when possible, the training is executed on the GPU, which makes the performances better.

However, an important aspect to remember is that when deploying FL in real life, the clients are different devices with limited computation power. This also means that on a specific case, some hardware-aware acceleration may be implemented. Now we put this thought aside, because it is out of the scope of this thesis.

- **Compression time ($t_{compress}$):** time spent on compression of the model. This time is directly dependant of the parameters configured for each run. Whether we apply pruning or not, and at which rate, and then depending on the level of precision we use for the quantization, and lastly if we encode the weights, influences this metric.
  Further below (section 4.2) we'll explain more about the way we implement those deep compression techniques, and different optimizations we propose to enhance them.
  Each client measures the training and compression times and returns them to the server alongside training loss. Then the server also aggregates these metrics the same way weight updates are aggregated.

- **Round time (t_{round}):** time required to complete an entire round. Even though it is calculated independently from the two previous metrics, this duration depends on time taken by clients to update and compress the model.

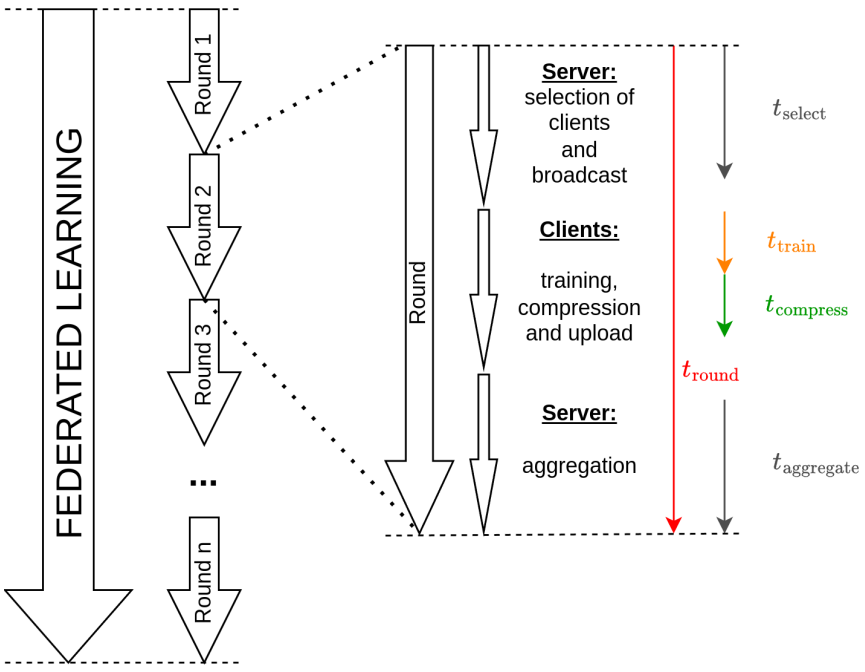


Figure 4.2: Time metrics proposition

We also keep track of the selection time t_{select} and aggregation time $t_{aggregate}$ whereby the server is actively working. By subtracting all previously defined time metrics for the

round time, the residual duration can be interpreted as communication time t_{comm}. It is important to note that, since the time taken on the client side may vary between clients and is aggregated by the mean, the communication time is also an average.

We define the **communication overhead** (in terms of time) as the ratio of communication time to the total round time, i.e.,

$$\begin{aligned} \text{communication overhead} &= \frac{t_{comm}}{t_{round}} \\ &= \frac{t_{round} - t_{computation}}{t_{round}} \end{aligned} \tag{4.1.1}$$

$$\begin{aligned} \text{where } t_{computation} = \; & t_{select} \\ & + t_{train} \\ & + t_{compress} \\ & + t_{aggregate} \end{aligned} \tag{4.1.2}$$

The aim of this thesis is to be energy-efficient in FL: that means that we want to reduce both compression and communication costs. However, by adding a compression stage we induce more computation at the clients side, and decrease the size of the model which should improve communication load. Put together, we hope to see the communication overhead decrease.

We sadly don't have knowledge of the way Flower handles the communication of client updates from them to the server, or the broadcast of the aggregated model. It implies that the communication time as previously described must be interpreted cautiously. Therefore, as previously discussed, the Huffman encoding provides a good effect. Knowing exactly the size of the transmitted data, if we specify a bandwidth and a delay (taken from a real-life situation for example), we can calculate exactly the communication time per client, and the associated communication overhead.

In order to track the metrics across training and under different settings, we opt for the technical solution of W&B (`wandb`) [7], which provides a Python API to record metrics in real time for runs. For every experiment, we create a new project and store all runs. W&B also provides a web graphical interface with plots of our metrics, and the possibility to compare individually those that interest us.

Additionally, the API also tracks some system usage stats in background, such a GPU and CPU power usage, memory allocated, or even temperature. These metrics can be useful for a deeper analysis of the impact of deep compression techniques and further optimizations we implement in our work.

Convergence speed

Finally, we introduce another key point for the energy efficiency of FL training: the **convergence speed**. Indeed, looking at a curve of the top-1 accuracy across rounds in Figure 4.3 (here we took the picture from one of our experiments as a general example), we can observe two properties:

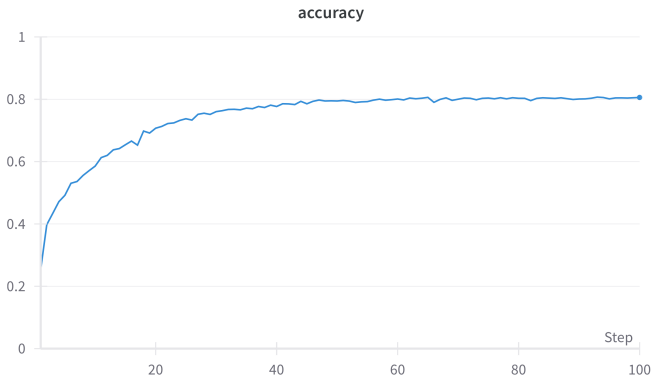


Figure 4.3: Accuracy across training rounds

- The accuracy first grows, and then stabilizes itself around its final value. Here we could maybe have stopped the training phase around the 50th round, without impacting the performance of the final model. However, dividing the number of round by two also means dividing the footprint of the training by two, in terms of time, computation resources, energy needs.

- The shape of this curve reminds us of a unit step response form a first order system, in control theory. Such a system has knows properties. Its shape, observable in Figure 4.4, can be described by the following equation:

$$f(t) = C \times (1 - e^{-t/\tau}) \tag{4.1.3}$$

  Where C is a constant corresponding to the final value, and τ is a time constant. It normally starts at zero, so for our use case we'll add a bias l, and thus adapt the C constant to be the difference between the highest and lowest values. The new equation we'd want to fit becomes then:

$$f(t) = C \times (1 - e^{-t/\tau}) + l \tag{4.1.4}$$

  Moreover, the τ constant has interesting properties, as shows Figure 4.4:

  - For $t = \tau$, the accuracy takes the value of $f(\tau) = C \times (1 - e^{-1}) + l \approx 0.63C + l$
  - For $t = 3\tau$, the accuracy takes the value of $f(3\tau) = C \times (1 - e^{-3}) + l \approx 0.95C + l$

– For $t = 5\tau$, the accuracy takes the value of $f(5\tau) = C \times (1 - e^{-5}) + l \approx 0.99C + l$

That mean that we can determine τ by directly reading on the curve the step for which 63% of the final value is reached.

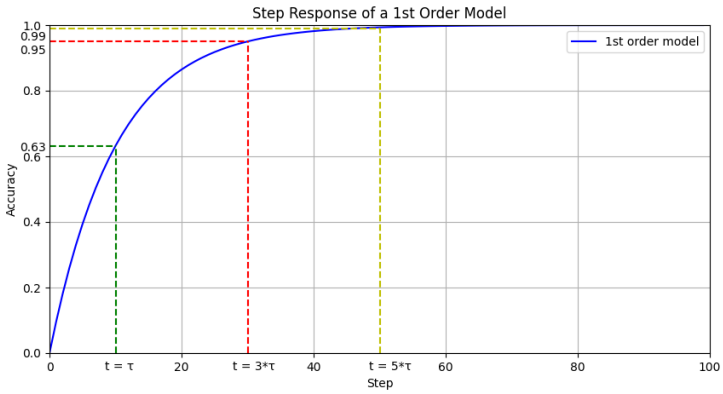


Figure 4.4: Step response of a first order system

Figure 4.5 shows how the model performs on our previous example. With a R^2 coefficient greater than 0.98, we can consider this model pertinent.

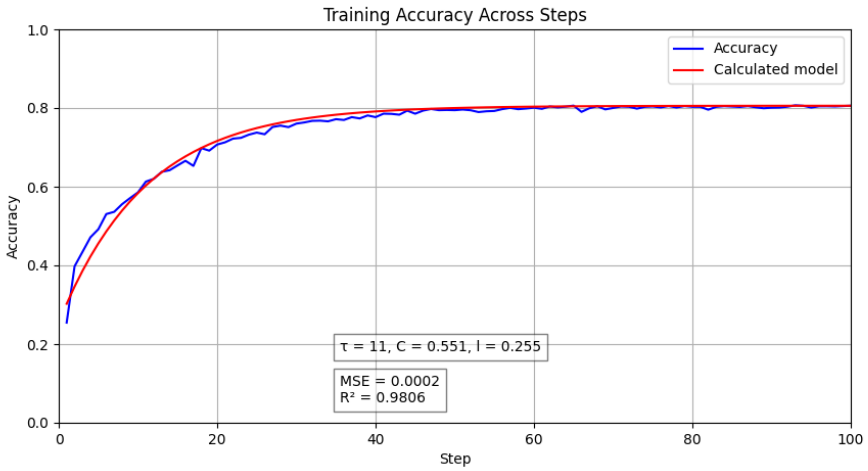


Figure 4.5: First order model fitted on a real-life FL training curve

This model can then be used to evaluate the speed of convergence, which is inverse to the size of τ. A small τ means that the FL training is reaching its final accuracy quickly, which is a property we look for.

4.2 Compression Techniques

Now that we have described the way we simulate the FL process, let's take a deeper look into the way we implement our deep compression techniques. We first discuss pruning and quantization individually in Section 4.2.1 and Section 4.2.2, respectively, then of the different subtleties of Huffman encoding and decoding procedure in Section 4.2.3, Section 4.2.4 and Section 4.2.5, and finally of a proposed compression pipeline in Section 4.3.2.

4.2.1 Pruning

In this thesis, we make the choice to use unstructured pruning only. That means that we don't consider the structured pruning, where entire structures (such as filters for convolutional layers) are pruned. Instead, we use *magnitude pruning*, which is the most discussed unstructured pruning in the literature.

The first way to implement pruning is using a PyTorch native pruning function [9]. Many options are offered, but the one that interest us is the `prune.l1_unstructured()`. It is implementing pruning on a `torch.nn.Module` object, i.e. a PyTorch entity representing a whole NN or a single layer. The use of the $l1$-norm is adequate in our case, because values pruned are one-dimensional (floating-point values), thus it simply corresponds to the absolute value.

However, we are going to implement our pruning by hand. Indeed, we want to explore the best option between layer-wise and model-wise pruning. We take inspiration of the code provided in [35], to implement the following Algorithm 2 for model-wise pruning.

Algorithm 2 Model-wise pruning

Input: $\boldsymbol{\omega}$, the weights of the model
Input: γ, the pruning rate
`sorted_weights` $\leftarrow \boldsymbol{\omega}$`.flatten().sort()`
`threshold` $\leftarrow$ value at the index $\gamma\times$`len(sorted_weights)` of the array `sorted_weights`
for each layer $\boldsymbol{\omega^i}$ in $\boldsymbol{\omega}$ **do**
  for each weight ω in $\boldsymbol{\omega^i}$ **do**
    if $\omega <$ `threshold` **then**
      $\omega \leftarrow 0$
    end if
  end for
end for
return $\boldsymbol{\omega}$, the updated weights of the model

In a similar fashion, we implement the layer-wise pruning in Algorithm 3.

Algorithm 3 Layer-wise pruning

Input: $\boldsymbol{\omega}$, the weights of the model
Input: γ, the pruning rate
  for each layer $\boldsymbol{\omega}^i$ in $\boldsymbol{\omega}$ **do**
    `sorted_weights` $\leftarrow \boldsymbol{\omega}^i$`.flatten().sort()`
    `threshold` $\leftarrow$ value at the index $\gamma \times$`len(sorted_weights)` of the array `sorted`
    for each weight ω in $\boldsymbol{\omega}^i$ **do**
      if $\omega <$ `threshold` **then**
        $\omega \leftarrow 0$
      end if
    end for
  end for
  return $\boldsymbol{\omega}$, the updated weights of the model

Both look really similar, the difference resides in the way we determine the threshold: for model-wise, we do it one time only, and then prune all weights accordingly; for layer-wise, we determine the threshold again for each layer.

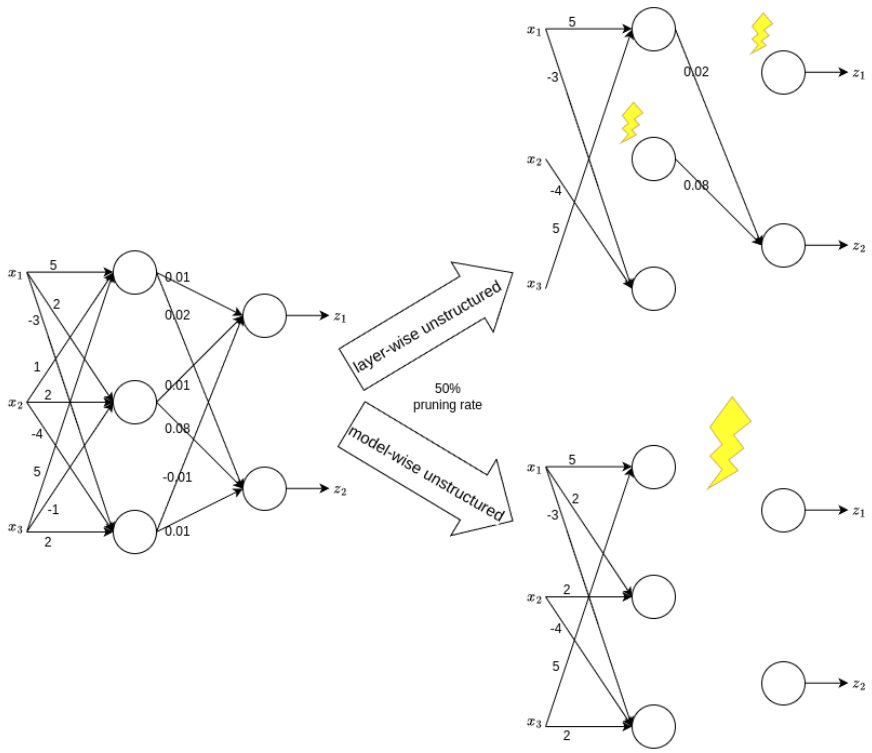


Figure 4.6: Unstructured pruning on unbalanced weights: model or layer threshold?

Even though the pruning is almost always implemented model-wise in papers, this pro-

position of doing layer-wise pruning comes from the following intuition: if on some layers, the magnitude of weights is sensibly lower than on other layers, the model-wise pruning might cut off all weights on those layers, with a sufficient pruning-rate. As illustrated in Figure 4.6, that could lead to the complete pruning of a layer, meaning that the output of the NN would not be input dependant, which is obviously not a desired behavior.
Using layer-pruning ensures that in each layer, some weights are not impacted. We'll try to validate or invalidate this intuition in the next chapter.

The attentive reader may also notice that both methods can be directly implemented using the `torch.l1_unstructured()` function, either by passing the whole model or a single layer as parameter, alongside the pruning rate. This is true, however doing it this way lacks of flexibility, when we want to also execute other operations during the loops, or also if we want to perform GPU acceleration. This is the reason why we use our own code for the purpose of pruning.

4.2.2 Quantization

The second step to reproduce and adapt all three techniques described by [14] is the quantization. Often studied in the literature, it can be implemented with many variations, some of them that we'll implement to then determine which one fits the best to our study case.

To this extent, on the one hand we implement a QAT method (quantization-aware training), i.e. on reduced precision. Following [35], we use the Brevitas framework [1] for this purpose.
The concept is really simple: instead of representing weights of the network using 32-bit floating-point numbers, we use a reduced number of bits for each number. Brevitas provides an easy solution by just rewriting similar code but importing layer-defining functions from another library. Concretely: `nn.Conv2d()` becomes `qnn.QuantConv2d()` with only a supplementary parameter, to define the number on bits to quantize on. We can note that this property is restricted to 4 or 8 bits quantization.

For this quantization, the impact is easy to evaluate: switching from a 32-bits representation to a 8- or 4-bits representation, respectively divides the size of the network by 4 or 8. However, the Brevitas framework (as for the date this thesis is written) doesn't include any saving options. That means that this gain can't be seen in a file explorer system on your computer, once the model is trained, nor in the computation performances of the code.
Side note: one way to tackle this issue would be to implement our own saving method, manually translating all layers and weights to bit chains of zeros and ones. We won't do it, because we prefer to focus on the k-Means PTQ, which provides more interesting results.

Then on the other hand, we implement also a PTQ method (post-training quantization) with k-Means method, exactly as in [14]. Our code is adapted from [19]. It takes a parameter q, which specifies the number of bits used to encode the quantized network. We can note that in a general case, the k in k-Means is equal to 2^q different clusters.

Just as for pruning, we can tune the PTQ process with many different approaches:

- **Layer- or model-wise:** we can choose to implement the quantization either for each layer separately, or all the model at once. Algorithm 4 and Algorithm 5 present both implementations.

Algorithm 4 Model-wise quantization

Input: $\boldsymbol{\omega}$, the weights of the model
Input: q, the number of bits on which to quantize
`all_weights` ← $\boldsymbol{\omega}$`.flatten()`
`kmeans` ← k-Means initialized object
`initial_spacing` ← 2^q-sized array
`kmeans` ← `kmeans.fit(all_weights,` 2^q`, initial_spacing)`
$\boldsymbol{\omega}$ ← `kmeans.cluster_centers_[kmeans.labels_].reshape(`initial shape`)`
return $\boldsymbol{\omega}$, the updated weights of the model

The difference between the two approaches is the instance on which we apply the k-Means algorithm: either the global model with all weights (and biases), or one layer at a time, with all its weights. Both approaches present advantages and disadvantages. Indeed, the model pruning allows a more efficient representation after training: only one codebook in needed to encrypt to network, whereas for layer quantization we need one codebook per layer. However according to [14], the codebook size is really little compared to the encoding of keys and indices, even after a Huffman encoding. Concerning disadvantages of the model-wise method, quantizing per layer offers a better precision and flexibility to the encoding, thus leading to better accuracy and loss performances.

Algorithm 5 Layer-wise quantization

Input: $\boldsymbol{\omega}$, the weights of the model
Input: q, the number of bits on which to quantize, initial_spacing
for each layer $\boldsymbol{\omega}^i$ in $\boldsymbol{\omega}$ **do**
`all_weights` ← $\boldsymbol{\omega}^i$`.flatten()`
`kmeans` ← k-Means initialized object
`initial_spacing` ← 2^q-sized array
`kmeans` ← `kmeans.fit(all_weights,` 2^q`, initial_spacing)`
$\boldsymbol{\omega}^i$ ← `kmeans.cluster_centers_[kmeans.labels_].reshape(`initial shape`)`
end for
return $\boldsymbol{\omega}$, the updated weights of the model

- **Initial spacing:** different strategies are possible when initializing the k-Means: the three main a using a *regular*, a *random* or a *density* spacing. All three are shown in Figure 4.7 for an arbitrary distribution of weights. We implement all three strategies in our code, and let the user choose the one to use. A specific experiment will let us conclude on which is the best.

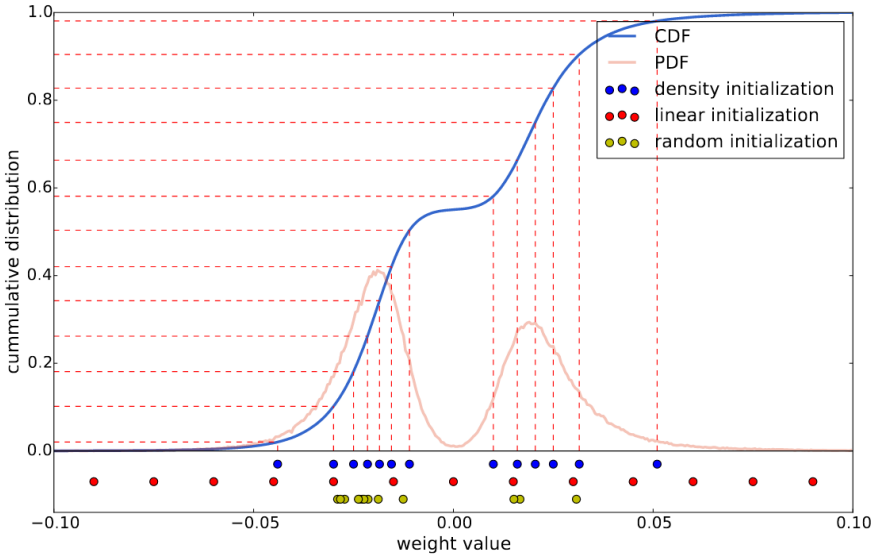


Figure 4.7: Three different spacing methods for initialization [14]

- **GPU acceleration:** the k-Means algorithm doesn't need to be implemented from scratch, as Scikit provides a version [10] that is often used in GitHub repositories along the literature. However, we enhance here our code via the use of CuML's own k-Means [3], which runs on the GPU. This allows faster computation, so gets us better performances.

The difference between QAT and PTQ, with different settings will be studied in the experiments section.

4.2.3 Huffman encoding with quantized NNs

Last but not least, we also implement a Huffman encoding/decoding in our code. The implementation is taken and adapted from [19].

First for the encoding, the logic is implemented as in Algorithm 6. The process can be broken down into several stages:

1. **Flatten the layer:** it makes computation and processing easier to have a one-

dimensional array to represent the data. When reconstructing the layer in the decoding process, the weights can be reordered like they were initially.

2. **Calculate frequency:** we evaluate for each symbol (weight value here, a floating-point number), the number of times they occur, and weight the result to get percentages. This is a key point of the whole process, because the Huffman tree will be built on this basis.

3. **Build Huffman tree:** we build the Huffman tree accordingly to the frequency map. As explained in the Section 2.3.3, most frequent weights get a shorter representation.

4. **Create a chain representing data:** for each weight of the layer, we read its encoding in the Huffman tree, and append it after the other weights. Because the Huffman code is a *prefix code*, we don't need to insert separators.

5. **Save the files:** we here save the files representing the layers of the network, with descriptive names to be able to rebuild it. This way of doing simulates a communication process. Actually in a real case of FL, we wouldn't save the files but rather send them to the server via a communication protocol.

Algorithm 6 Huffman encoding for weights

Input: $\boldsymbol{\omega}$, the weights of the model
  for each layer $\boldsymbol{\omega^i}$ in $\boldsymbol{\omega}$ **do**
    `all_weights` $\leftarrow$ $\boldsymbol{\omega^i}$`.flatten().nonzeros()`
    `frequency_map` $\leftarrow$ `[frequency(all_weights)]` ▷ Calculate a frequency map
    `huffman_tree` $\leftarrow$ `huffman_tree(frequency_map)` ▷ Build a Huffman tree
    `data` $\leftarrow$ `[]` ▷ Initialize the data encoding
    for each weight ω in $\boldsymbol{\omega^i}$ **do**
      `data.append(huffman_tree(`ω`))`
    end for
    `save(data)` or `send(data)` ▷ Save the keys values to decode the data
    `save(huffman_tree)` or `send(huffman_tree)` ▷ Save the codebook
  end for

Now to evaluate the size of the model, we just read it in our file explorer. To evaluate the size of model in [35], the adopted method is to use `torch.save()`, go to your file explorer, use a ZIP compression, and finally read it. As long as the extra step of apply the ZIP algorithm isn't applied directly in the code, the compression is only theoretical, so not a real proven performance of the provided results. Still we'll use this method when we don't use the Huffman encoding, for the sake of a baseline.

Nevertheless, another property of the Huffman code is that it is *optimal*. That means that even by applying the ZIP algorithm in the file explorer afterwards, we can't reach a

better result. That will be a useful property to evaluate the real communication cost after encoding with Huffman.

Another important remark: the more the model is quantized, the more encoding with Huffman makes sense. That is because thanks to quantization, many weights share their value, and are therefore frequent. If we encode a model whose weights all have different values, the size of the encoded model will actually be larger than the size of the original model, because we will be encoding unique keys mapping all to different values, instead of storing the values directly.

4.2.4 Huffman encoding with pruned NNs

We've just explained how to Huffman encode the data, as proposed by [14]. Moreover, when we use pruning, we can use the sparsity property to represent the data slightly differently. Scipy provides CSR/CSC matrices (compressed sparse row/column matrix) [39] that record the nonzero weight values, and indices where they occur. We can then apply our previous quantization and Huffman encoding only to the `data` field of the matrix.

Now comes the problem of representing efficiently the indices. Han *et al.* [14] also introduce the idea of representing the index difference on a fixed-length bit chain, eventually adding zeros to prevent overflow, as explained in Section 3.2.1. However, we can also think of applying another Huffman encoding on these index differences.

The idea comes from the implementation [19] of [14], and seems efficient because such differences (image of the sparsity) are likely to repeat themselves a lot in a layer, and thus can be effectively represented with an entropy coding like Huffman.

4.2.5 Huffman decoding

Now that we've generated files with Huffman encoded data and indices, we want to read them to reconstruct our NN. The idea of the process is to execute each step of the encoding in a reversed order:

For each client:

1. **Read the files:** we fetch for each client the output it produced: data, indices and Huffman trees for all layers. All are represented by a chain of 0's and 1's for each layer of the model. For the data and index differences they correspond to the encoding, whereas for the codebook they correspond to a fixed-length byte code,

that can be understood by the server. The way codebooks are encoded is out of the scope of this thesis, therefore we keep the way [19] did it.

2. **Decode the data:** from the tree, we decode the values of encoded data, to reconstruct a 1-d array. Thanks to the property of the Huffman code being a *prefix code*, the solution is unique.

3. **Decode the indices:** from the tree, we decode the values of encoded index differences, to reconstruct a 1-d array. Once again, the solution is unique. Using a cumulative sum method, we can then build the absolute indices.

4. **Build the compressed sparse representation:** Having data and indices, we can rebuild the CSR/CSC matrix representation of the layer.

5. **Extract the weights:** we rebuild the whole matrix by filling holes with zeros. Knowing the expected shape of the model, we reassign the weights in the same order as when they were flattened before encoding.

Only once:

4. **Aggregation:** now that we dispose of the reconstructed updated models for each client, we can aggregate them normally, for example with the FedAvg strategy [26].

The Algorithm 7 presents the decoding as it could be implemented.

Algorithm 7 Huffman decoding

Input: $\boldsymbol{\omega}$, the overall shape of the model
  for each layer $\boldsymbol{\omega^i}$ in $\boldsymbol{\omega}$ **do**
    `data_huffman_tree` ← `open(data_huffman_tree)` ▷ Read the tree for data
    `data_encoded` ← `open(data)` ▷ Read the encoded data
    `data` ← `decode(data_encoded, data_huffman_tree)` ▷ Decode the data

    `indices_huffman_tree` ← `open(indices_huffman_tree)` ▷ Once again
    `indices_encoded` ← `open(indices)`
    `indices` ← `decode(indices_encoded, indices_huffman_tree)`
    `indices` ← `cumulative_sum(indices)` ▷ Calculate true indices

    `compressed_matrix` ← `CSR(indices, data)` ▷ Build compressed sparse matrix
    `whole_matrix` ← `compressed_matrix.to_dense()`
    $\boldsymbol{\omega^i}$ ← `whole_matrix`
  end for
  return $\boldsymbol{\omega}$, the decoded weights of the model

4.3 Further Enhancement on Compression Techniques

Now that we have a working framework with metrics, and implemented compression techniques, what can we do more? This is the purpose of this section to answer this question.

4.3.1 Direction of compression

As seen before, the compression can be implemented either on the upstream (clients compression) or on the downstream (server compression), or in both directions.

The idea of implementing compression techniques aims to tackle the issue of *bottleneck* at the server's entry point, when all clients' updates are received at the same time. That is the reason why we apply compression first at the clients' side, so that the volume of data they have to communicate is smaller.

But on the other hand, we can also implement our compression techniques at the server: it has the same advantage for reducing the size of the network before communication. However, it induces more computation on both sides, especially at the clients that can have some limited resources, that we don't want to exploit even more. Moreover, we must not forget that pruning and quantization are destructive methods that lead to a loss of accuracy of the NN. Put all together, nothing leads us to want to compress on the downstream.

For all of these reasons, in our experiments we only apply the compression on the clients' side.

4.3.2 Full Compression Pipeline

We've explained in the previous sections how our compression techniques work and they are implemented. We can now assemble them all together to create a compression pipeline, as it is often done in the literature [14, 25]. However, as shows the Figure 4.8, the process doesn't look efficient:

- **Pruning:** we fetch all weights of the model, find the global threshold, go through each layer and prune what needs to be, and return an updated version if it wasn't done in place.

- **Quantization:** we go trough each layer, fetch the weights, use the sparse representation to isolate nonzero weights, flatten them, apply our k-Means, set back the

weights, and return an updated version if the variable wasn't update in place.

- **Huffman encoding:** we go through each layer, fetch the weights, use the sparse representation to isolate nonzero weights, flatten them, calculate index differences, and encode all of that.

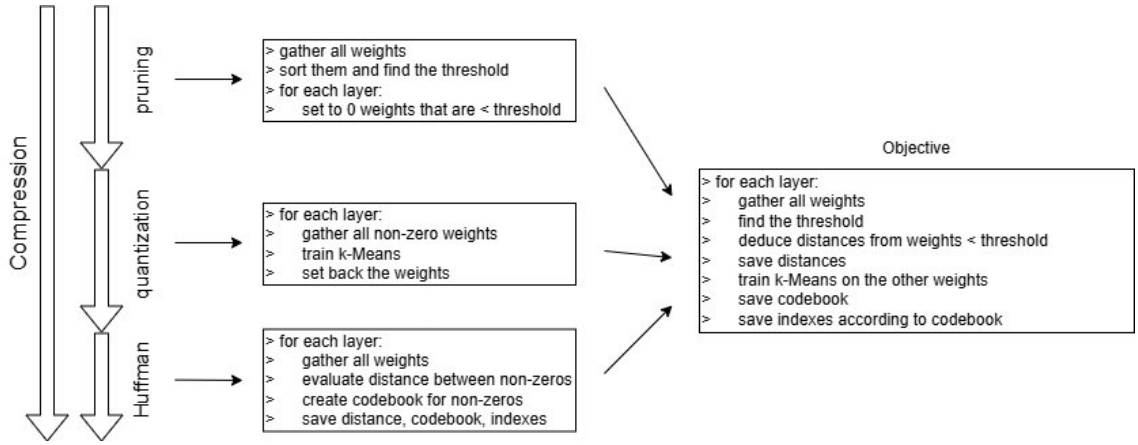


Figure 4.8: Proposed idea of a more efficient computation

This makes a lot of iterations different times on the same objects, and fetch/allocation of data. That's why we introduce the idea of executing all techniques on a single loop per layer. Then the process becomes as of in Algorithm 8.

Algorithm 8 Full compression pipeline

Input: $\boldsymbol{\omega}$, the weights of the model
Input: γ, the pruning rate
Input: k, the number of clusters for quantization
  `sorted` $\leftarrow$ $\boldsymbol{\omega}$`.flatten().sort()`
  `threshold` $\leftarrow$ value at the index $\gamma\times$`len(threshold)`
  for each layer $\boldsymbol{\omega}^i$ in $\boldsymbol{\omega}$ **do**
    `pruned_weights` $\leftarrow$ $\boldsymbol{\omega}^i$`.prune(threshold)` and flatten it
    `compressed` $\leftarrow$ sparse representation of `pruned_weights`
    `compressed.data` $\leftarrow$ `quantize(compressed.data,` k`)`
    `difference` $\leftarrow$ calculate difference from `compress.indices`
    `huffman_encode(difference)` and send to the server
    `huffman_encode(compressed.data)` and send to the server
  end for

- As the experiments prove later on, the model-wise pruning has better effects than the layer-wise one. We then need to fetch a first time all weights in order to determine the value of the threshold. The pruning itself will be executed in the main loop.

- Because we use a Huffman encoding, we don't need to specify the number of bits we want to quantize on: at the end, the representation of the weights won't have fixed-length. We rather specify a number of clusters, which offers more flexibility.

The Figure 4.9 visually presents this pipeline for a layer of a network, in eight steps.

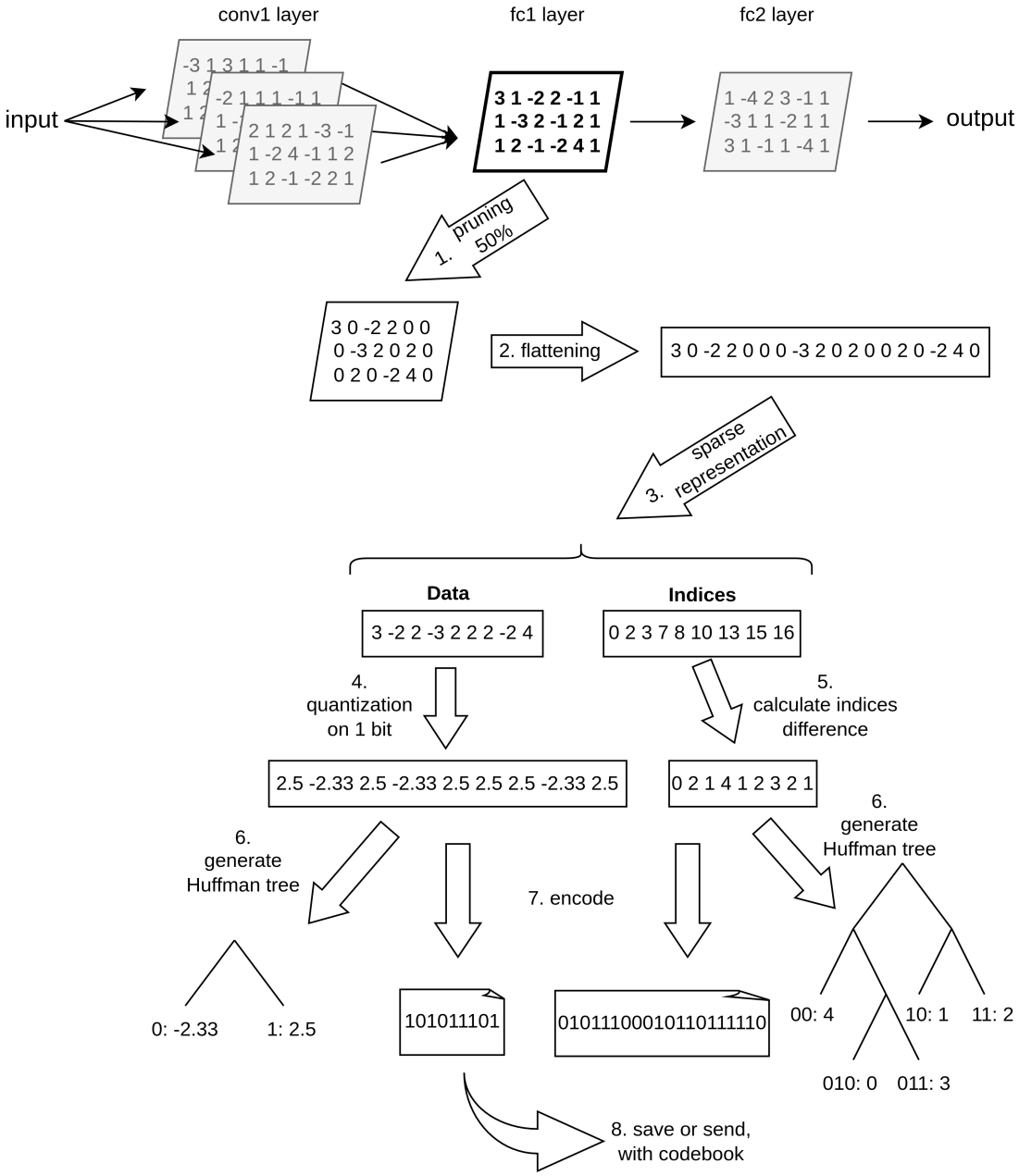


Figure 4.9: FCP for a layer on a simple NN

All this put together, Algorithm 9 sums up the integration of the FCP in our framework.

Algorithm 9 FL process using the FCP

Input: γ, the pruning rate
Input: k, the number of clusters for quantization
Input: h, the hyperparameters for learning
Input: n, the number of rounds
Input: c, the number of clients selected for each round
  for each round i in (1, ..., n) **do**
    select c random clients
    broadcast the model to selected clients
    for each selected client simultaneously **do**
      train the model knowing h
      apply the FCP knowing γ, k
      upload the encoded update to the server
    end for
    decode the updates
    aggregate the updates and replace the former model
  end for

5 Experiments

In this section, we aim to evaluate and validate different features we introduced in Chapter 4. We will first focus on determining the best setups for pruning and quantization, and then also fine-tuning other parameters inherent to FL and training of ML models. Finally, as the culmination of this thesis, we will evaluate the performance of the FCP from every angle.

5.1 General Considerations

To keep a consistent baseline, we will compare our work to a paper published by Lucas Grativol in October 2023 [35]. The author implements both pruning and quantization in a FL context, and evaluates them separately, with good results. We will then compare our results to theirs to evaluate the success of our methodology.

5.1.1 Models and Dataset

In our reference paper, the experiments are conducted on ResNets (residual neural networks). The two specific architectures that were kept are the ResNet-12, with 780k trainable parameters, and the ResNet-18, with around 11M parameters.

These are not the biggest NN of the ResNet family, but provide good results on datasets such as the CIFAR-10, with for example 91% top-1 accuracy with a ResNet-20 [15], which is close to the ResNet-18.

5.1.2 Distribution of data

In our reference paper, both cases of IID and non-IID data are addressed. For this purpose, we first want to make sure that our partitioner works well. For 10 clients, Figure 5.1 shows an IID distribution, whereas Figure 5.2 shows a non-IID one.

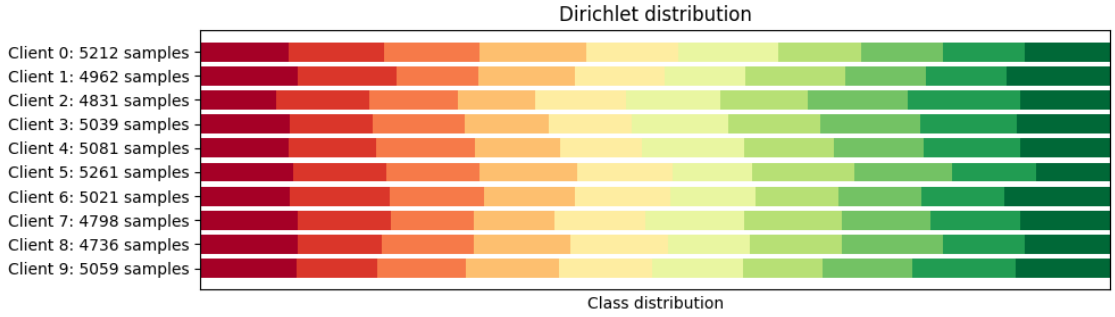


Figure 5.1: IID distribution ($\alpha = 100$) on 10 clients

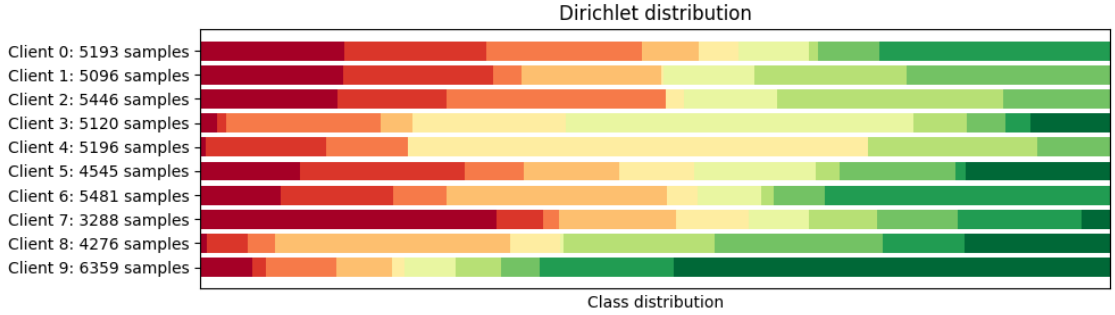


Figure 5.2: Non-IID distribution ($\alpha = 1$) on 10 clients

The difference between both is clearly visible here, with all classes approximately equally represented on the first figure, and also almost as many samples per client ; the second figure has a bigger variance for the number of samples per client.

For a further experiment we'll need a non-IID distribution on 100 clients, so Figure 5.3 shows what it looks like. The difference in number of samples per client is even bigger: here from 2 to 2478 for clients 44 and 88 respectively, in this particular example. Using such a distribution will show us how much the degree of IID-ness of data impact FL.

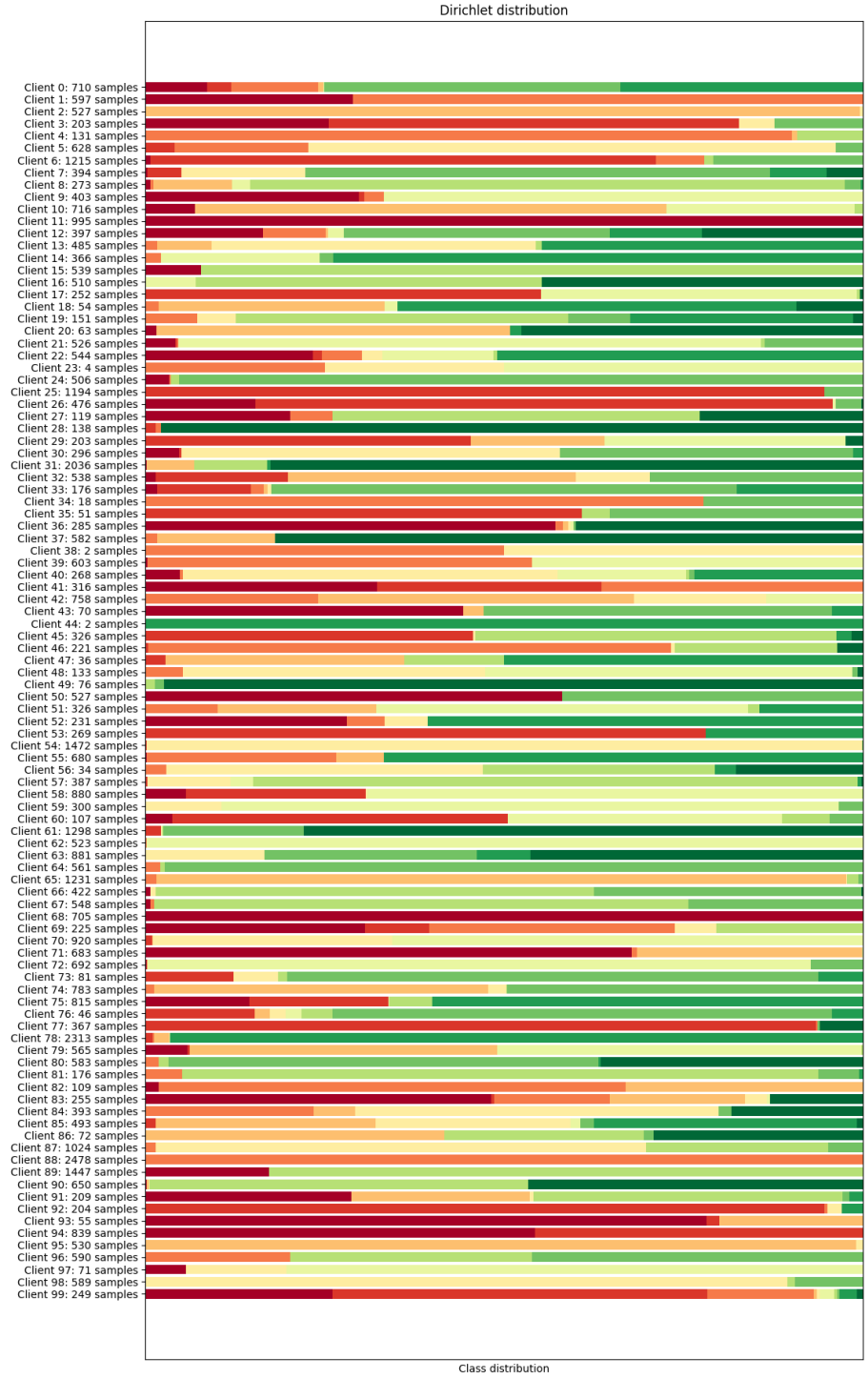


Figure 5.3: Non-IID distribution ($\alpha = 1$) on 100 clients

5.2 Evaluation of Pruning

Now that we have a clearly defined framework, we evaluate the different settings for pruning, starting with the scale at which we apply the pruning.

5.2.1 Layer-wise vs. Model-wise pruning

Following the thought in Section 4.2.1, we want to evaluate which one of layer-wise or model-wise unstructured pruning provide better results.

For this purpose, we will consider the training on a ResNet-12 on the CIFAR-10, in a setup with 10 clients and an IID distribution of the data.

We apply pruning rates from 0 to 99%, and compare the results. The Figure 5.4 shows the evolution of the accuracy across the rounds of training of FL, for runs with a pruning rate of 60%, and either model- (in green) or layer-compression (in orange). For this experiment, and all following, all values are averaged on 3 passes, unless otherwise specified.

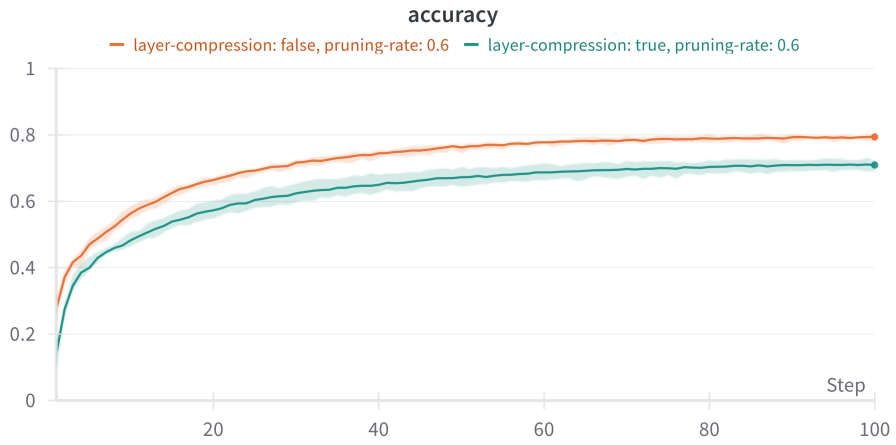
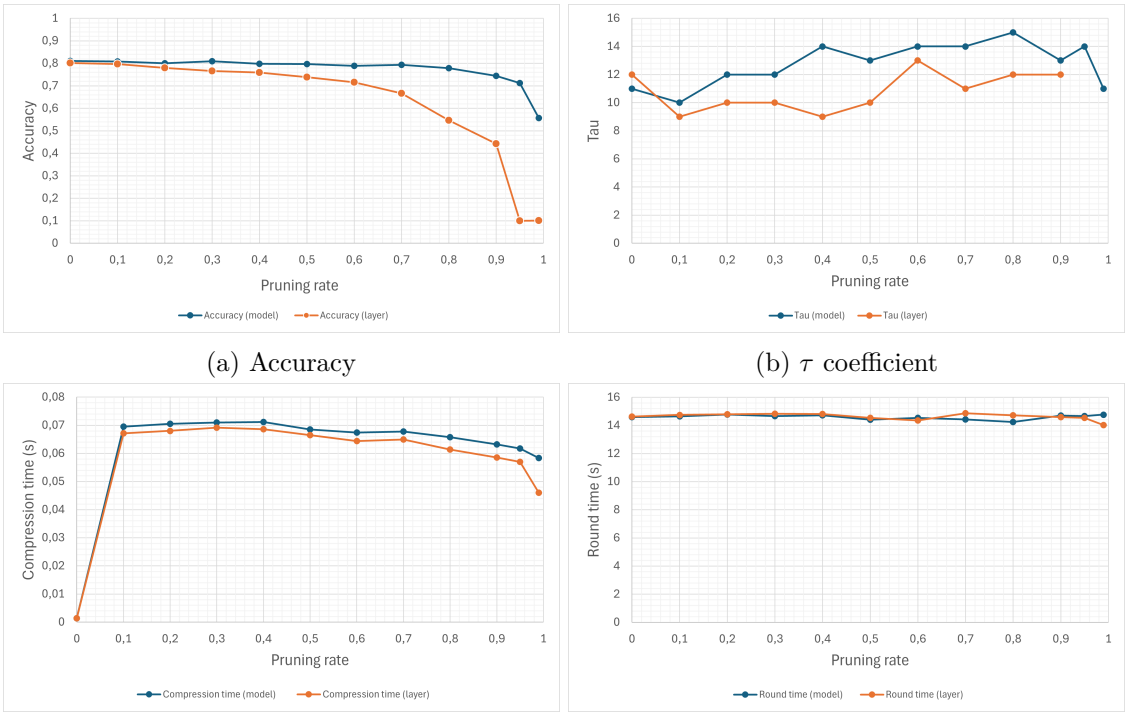


Figure 5.4: Evolution of accuracy for layer-wise and model-wise pruning

For this particular setup of 60% pruning, we see clearly that model-compression provides a sensibly better accuracy, which is a key objective in our work.

Now, let's take a look at Figure 5.5. These four graphs respectively show the final accuracy achieved, the τ coefficient indicating convergence speed, the compression time, and the round time, all plotted against the pruning rate



(a) Accuracy

(b) τ coefficient

(c) $t_{compress}$

(d) t_{round}

Figure 5.5: Evaluation of the impact of layer- or model-wise pruning

A few remarks on each of these graphs:

- **Accuracy:** For all pruning rates, the accuracy in the case of model-pruning is higher than when layer-pruning. It actually goes against our intuition of limits of the model-pruning expressed in Section 4.2.1, meaning that weights values are most probably uniformly distributed across layers, and that the risk of a layer being entirely pruned is minimal. This could then be enough to conclude, but let's take a look at the other metrics.

- **τ:** The τ coefficient, corresponding to the speed of convergence, is usually smaller — better — for the layer-wise pruning. It has to be nuanced as the difference is not huge, and moreover, what good is a faster convergence if the final accuracy is substantially worse?

- **$t_{compress}$:** The layer-wise pruning also have a slightly better training time than the model-wise pruning, but the same argument as for τ applies here.

- **t_{round}:** Surprisingly, the average time taken for a whole round (or step) of FL training is often slightly bigger with layer-wise pruning, even though both layer-wise

and model-wise are really close.

We have compared both versions regarding accuracy and convergence speed. However, in Section 4.1.2 we have also introduced the communication overhead. We don't consider it here because both pruning have the same effect on the size of the model, and thus the metric is not relevant.

In conclusion, we can say with certainty that the **model-wise pruning** provides better results than its layer-wise cousin.

5.2.2 Effect of the pruning rate

Now that we have chosen how we apply our pruning, we want to evaluate at which degree we can use it, i.e. up to which pruning rate we can go without too much affecting the performances of the model.

Grativol [35] evaluates his pruning implementation on the same setup as in the previous section, i.e. ResNet-12 on CIFAR-10 and IID data with 10 clients and 40% selection rate. He also applies it on another setting: ResNet-18 on the CIFAR-10, but this time non-IID data ($\alpha = 1$) and 100 clients, with only 10% selection rate. On this second setup, he claims a reduction of the model size to 10 MB (44.7 MB initially) with only 4.63% accuracy degradation.

We then establish the same setup, with non-IID data distribution similar to the one in Figure 5.3, and the Figure 5.6 shows the evolution of the accuracy during the training.

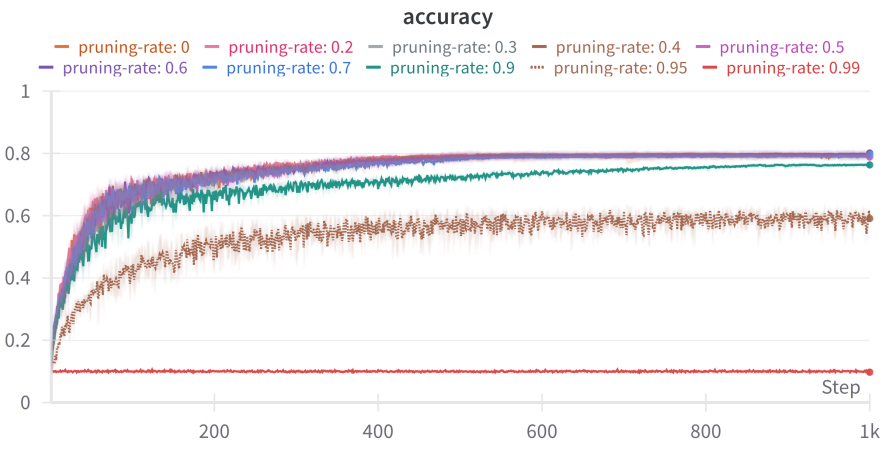


Figure 5.6: Accuracy across training

We can see that the training converges up to 95%, but for 99% pruning the accuracy is stuck at around 10%. We also notice a drop of accuracy when the pruning rate grows, just as in the previous experiment in Figure 5.5a. To make our mind clearer here, we plot in Figure 5.7a the final accuracy as a function of the pruning rate. Figure 5.7b and Figure 5.7c also present the τ coefficient and size of the final model when compressed with a ZIP algorithm.

For both experiments (ResNet-12 on IID data, ResNet-18 on non-IID data), we can observe that we can prune up to 70% with around 1% drop of accuracy, which is a desired behavior. We also see on Figure 5.7c that the size of the model decreases linearly with the pruning rate; for 70% pruning rate, we reduce of 70% the size of the model. This will help reduce the communication footprint and alleviate the bottleneck effect at the server's entry point. Moreover, on both Figure 5.5b and Figure 5.7b we observe that the τ coefficient (metric of the speed of convergence) increases slowly up to 80% pruning rate, and becomes greater after 90%. This means that even by pruning a lot (up to 80%), the FL training won't need more rounds to reach the same final accuracy as with no pruning at all.
For the ResNet-12, we see on Figure 5.5c and Figure 5.5d that the pruning rate doesn't significantly impact the time per round t_{round}, and actually the more we prune, the less we need time $t_{compress}$ for this operation.

We cited previously that Grativol [35] claims a 4.63% drop of accuracy when the size of the model is reduced to 10MB. In our setup, this size is reached for 80% pruning rate, where we read:

- **For ResNet-12:** 77.80% accuracy at 80% pruning, compared to 81.05% accuracy for the baseline (no pruning). It corresponds to a 3.25% drop.

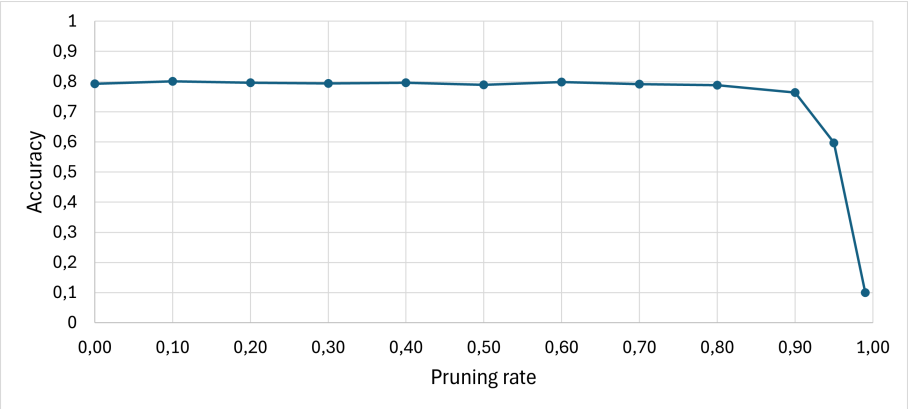
- **For ResNet-18:** 78.81% accuracy at 80% pruning, compared to 79.26% accuracy for the baseline (no pruning). It corresponds to a 0.45% drop.

We then have better results in terms of accuracy drop. However, we have to say that their baseline reaches 84.43% accuracy, thus even with 4.63% drop they have better results. The difference in performance in terms of accuracy cannot be explained, although we use the same model under the same conditions, with the same number of clients on the same dataset.
Still it is not a big deal, as we proved the efficiency of our pruning scheme.

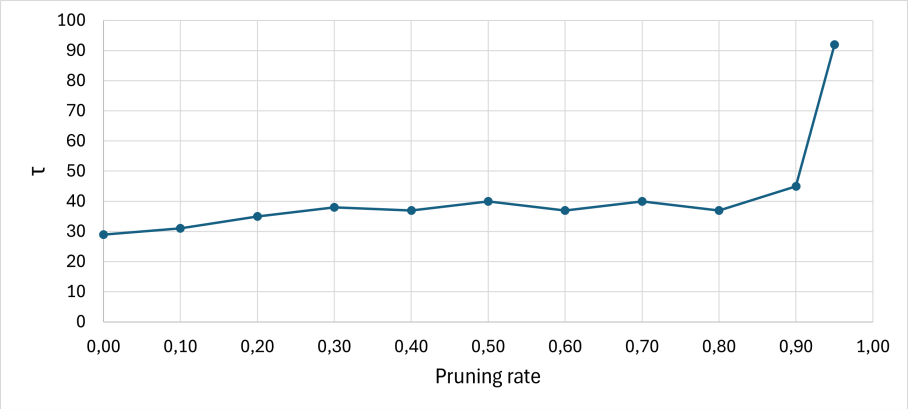
5.2.3 Conclusion on pruning

Pruning to a high rate such as 70% leads to minor accuracy drops, whereas the size of the model shrinks, widely improving the communication performances of the FL process. Even in terms of computation time, a larger pruning rate reduces the overhead induced

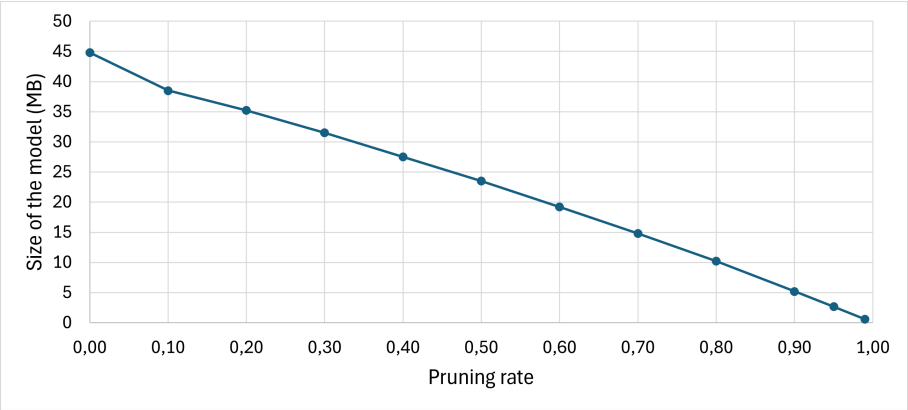
by the compression process.



(a) Accuracy



(b) τ coefficient



(c) Size of the model

Figure 5.7: Evaluation of the training of the ResNet-18 with different pruning rates

5.3 Evaluation of Quantization

After evaluating the performances of pruning in the chosen setup, we now want to choose the best quantization options, and see their impact on a FL framework.

5.3.1 QAT vs. PTQ

First, we want to compare the performances of QAT (quantization-aware training) and PTQ (post-training quantization). As explained in Section 4.2.2, Grativol [35] uses the Brevitas framework to implement a QAT. However, in the original definition of the deep compression techniques [14], a k-Means method is used, i.e. in PTQ.

Grativol experiments with quantization with Brevitas framework, but does not provide details about the model used. However the results lead us to think that the ResNet-12 was used. The training is executed on IID data, under the same conditions that for the first experiment. The paper claims that the accuracy isn't much affected by this quantization (for 8 and 4 bits), as shown in Table 5.1.

Quantization	1 epoch	10 epochs	Message size
Baseline	78.94%	78.18%	2.97 MB
8 bits	78.80%	78.58%	0.75 MB
4 bits	79.74%	77.04%	0.38 MB
1 bit	48.93%	70.89%	0.10 MB

Table 5.1: Claims for ResNet-12 on CIFAR-10 for the IID case [35]

We establish a similar setup, and want to verify these claimed results. To this extent, we consider the ResNet-12 trained on 10 clients, with the CIFAR-10 dataset in an IID scenario. We will compare our QAT Brevitas implementation on 8 and 4 bits to a baseline (without compression, already established in the previous section), and to a PTQ with k-Means, also on 8 and 4 bits (256 and 16 clusters per layer respectively).

The Table 5.2 sums up the results for the final accuracy after FL training, and the different time metrics (expressed in seconds) averaged over the whole process.

epochs	**Quantization**	**Accuracy**	t_{train}	$t_{compress}$	t_{round}	$t_{compute}$	τ
1	QAT on 4 bits	65,731%	11,82	0,06	48,15	36,25	12
10	QAT on 4 bits	61,293%	120,04	0,07	480,65	360,51	3
1	QAT on 8 bits	74,235%	11,96	0,06	49,23	37,19	9
10	QAT on 8 bits	72,017%	119,46	0,06	478,33	358,79	3

Table 5.2: Accuracy for ResNet-12 on CIFAR-10 for the IID case

Even though it gives us good information about the evolution of the accuracy of the training, we can also visualise it directly on Figure 5.8, where all four settings are shown, and the black curve represents the baseline for 1 epoch (without compression).

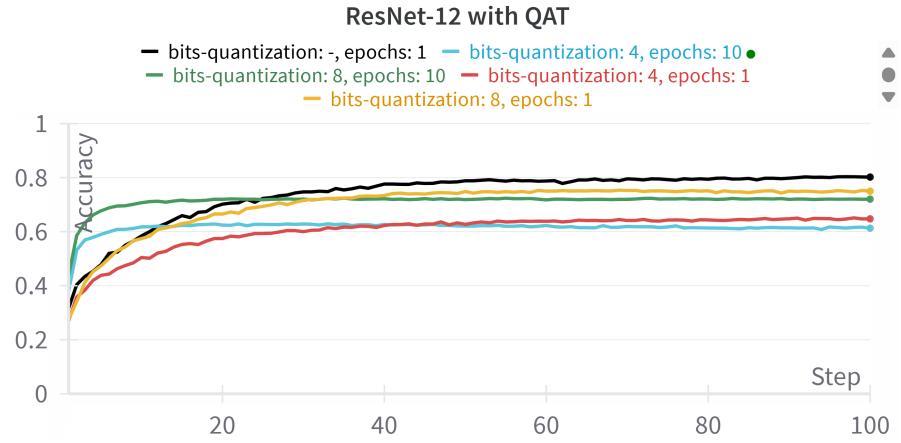


Figure 5.8: Evolution of the accuracy across the training

We can first note that the our QAT severely degrades the accuracy, which is a behavior that we want to avoid. However, our results are notably lower than the claims made by Grativol, for reasons we are unable to explain. Indeed, we use the exact same framework under the same conditions. We also don't try to quantize on 1 bit, as Grativol uses a special framework for this purpose [28], but it is of no real interest in our context.

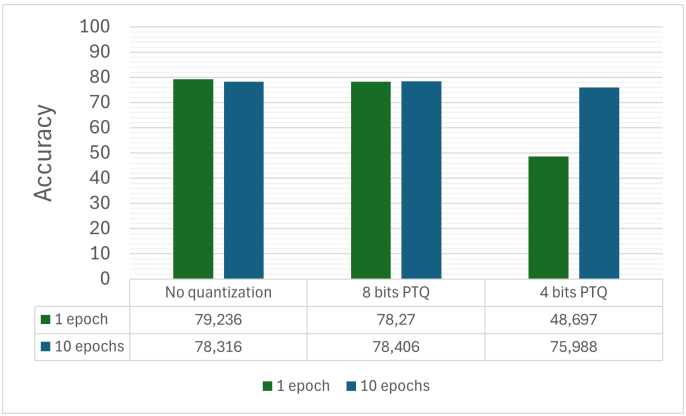
To go further, we can evaluate our implementation of PTQ with k-Means. As said before, we keep the same values for quantization (reduction to 8 or 4 bits), in order to have the fairest comparison possible.

The results are shown in Figure 5.9, for 3 different metrics.

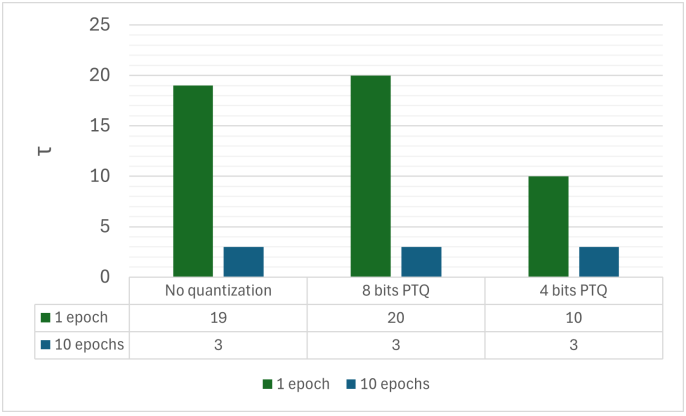
Let's compare both approaches point by point:

- **Accuracy:** The baseline (shown in Figure 5.10a on the first column) provides an accuracy close to 79%, whether for 1 or 10 epochs.
  As we see in the same figure, PTQ on 8 bits gives a drop of about 1% for 1 local epoch, or even none for 10 local epochs.
  However, as reported in Table 5.2, QAT on 8 bits leads to a accuracy or 74% and 72% for 1 and 10 local epochs respectively, so an accuracy drop of 5 to 7 percents.
  We can note that our baseline performs better than Grativol's one in terms of accuracy.

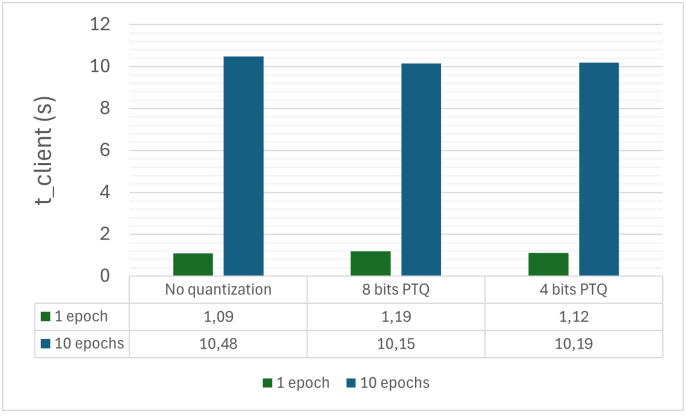
- **Convergence speed:** For each scenario, for 10 local epochs we measure $\tau = 3$ in



(a) Accuracy



(b) τ



(c) $t_{client} = t_{train} + t_{compress}$

Figure 5.9: Evaluation of PTQ with k-Means

Table 5.2 and Figure 5.9b.
However for 1 local epoch, the time constants τ are better for QAT than for PTQ

on 8 bits with 9 and 20 respectively, when on 4 bits we measure 12 and 10.
We also don't want to forget that the value of τ cannot be fully evaluated without measuring time metrics such as t_{round}, or t_{train} and $t_{compress}$ in our case.

- **Computation time:** For this metric, we measure a huge difference between PTQ and QAT. Note that the metric measured for PTQ is $t_{client} = t_{train} + t_{compress}$, because the experiment was defined and executed before refining the latest time metrics. Time metrics are also hardware dependent, which means take such evaluation should be done carefully.
  For 1 local epoch, QAT requires almost 12s, when PTQ only takes slightly more than 1s. For 10 local epochs, these times are simply multiplied by 10.
  Concretely, even if the τ for QAT on 8 bits is 2.2 times smaller than the one for PTQ, the whole FL training process is $12/2.2 \approx 5.5$ longer to reach the final accuracy.
  This extreme difference is explained by the fact that Brevitas [1] isn't implemented on the GPU at all, as PyTorch [30] is.

- **Size of the model:** We can measure this metric for PTQ, but not for QAT. Indeed, Brevitas doesn't provide any method to save the model taking benefit of its inner quantization; the values that are claimed by Grativol in Table 5.2 are only theoretical. We don't present the results of size for PTQ here, because they are only relevant when we apply a Huffman encoding afterwards, taking advantage of the repetitions of values in the layers.

To conclude on the quantization approach we choose: **PTQ with k-Means provides better results**. Brevitas' QAT presents two major issues in our work: first, it is only a proof of concept (POC) and not optimized yet, so doesn't provide satisfying performances; second, it isn't interesting for Huffman encoding because QAT doesn't necessarily create repetitions of weights in the layers.

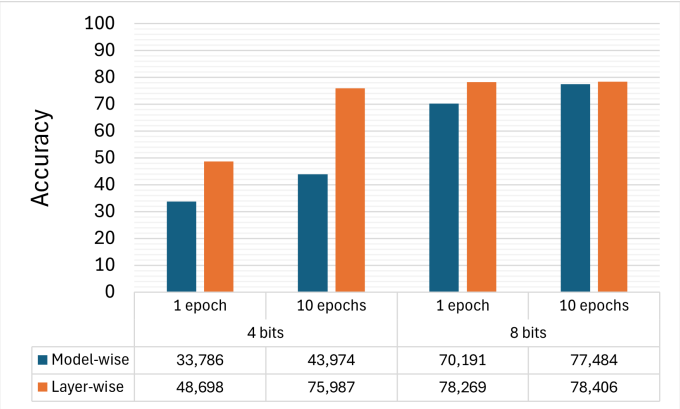
5.3.2 Layer-wise vs. Model-wise PTQ

Now that we've decided to use PTQ with k-Means, we can implement it in two different ways: either layer-wise or model-wise, as explained in Section 4.2.2.
Note that we had the same reflection for pruning earlier (Section 5.2.1), and had the wrong intuition that layer-wise pruning would be better.

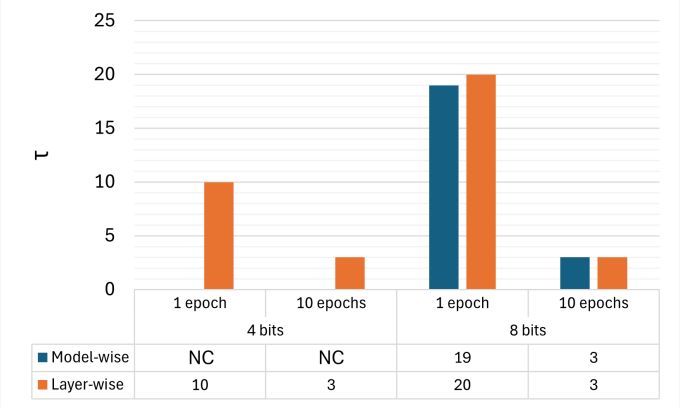
Here the motivation is the following:

- **Layer-wise quantization:** k-Means would provide a more precise quantization, on a smaller number of weights as if it were model-wise. It also means that the computation requirements would be lighter, even if repeated once for each layer.

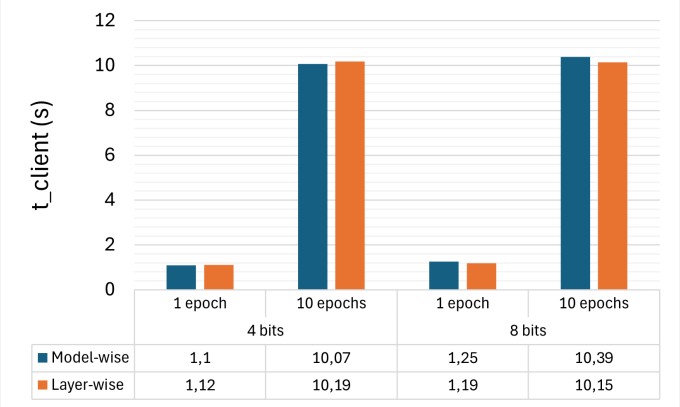
- **Model-wise quantization:** we would quantize the model only once, on a huge sized k-Means. The biggest advantage is that we need only one codebook for the whole model, so we have a lighter communication overhead.



(a) Accuracy



(b) τ coefficient for convergence speed



(c) $t_{client} = t_{train} + t_{compress}$

Figure 5.10: Comparison of layer-wise and model-wise k-Means

The Figure 5.10 provides results for both settings, measuring the accuracy, the convergence speed τ and the t_{client}. Just like for the previous experiment, we try quantize the ResNet-12 on either 4 or 8 bits, and compare with 1 or 10 local epochs.

Once again, let's compare both approaches point by point:

- **Accuracy:** For all four setups, we read a clear advantage for layer-wise quantization in terms of accuracy. Only when quantizing on 8 bits and training on 10 epochs, the model-wise quantization can have a close performance.

- **Convergence speed:** For 4-bits model-wise quantization, we couldn't calculate τ because of a lack of convergence. When quantizing on 8 bits, model-wise and layer-wise quantizations converge in almost the same number of rounds.

- **Computation time:** We had a feeling that applying the k-Means layer- or model-wise could induce a computation overhead (in one way or the other), but measurements of the time required by the clients to train and compress the model indicate that both approaches lead to similar results.

- **Size of the model:** As explained in the previous section, calculating the size of a quantized model only makes sense if we take advantage of this property, for example using a Huffman encoding. Thus for the moment we don't have measurements for this metrics.
  However as said before, model-wise quantization allows to create only one codebook. Still as reported by Han *et al.* [14], the codebook size is expected to be small compare to the storage size of encoded values, so it shouldn't make a huge difference.

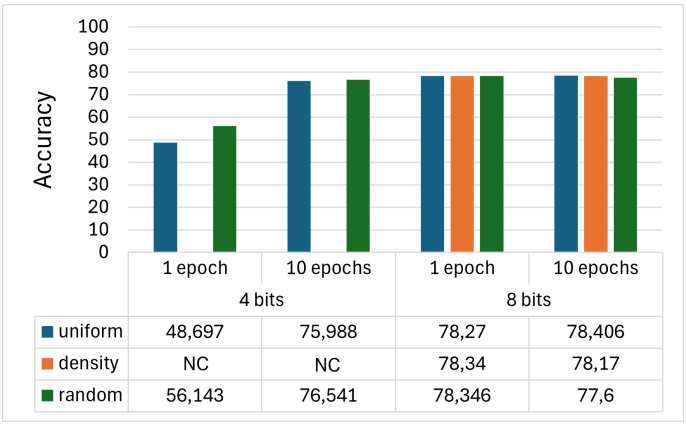
To conclude in this section, **we will be using layer-wise k-Means PTQ**, unlike for pruning where we decided to use the model-wise approach.

5.3.3 Effect of the initial spacing on layer-wise k-Means

Now, to complete the definition of PTQ, we have to decide which spacing initialization to use for the k-Means.

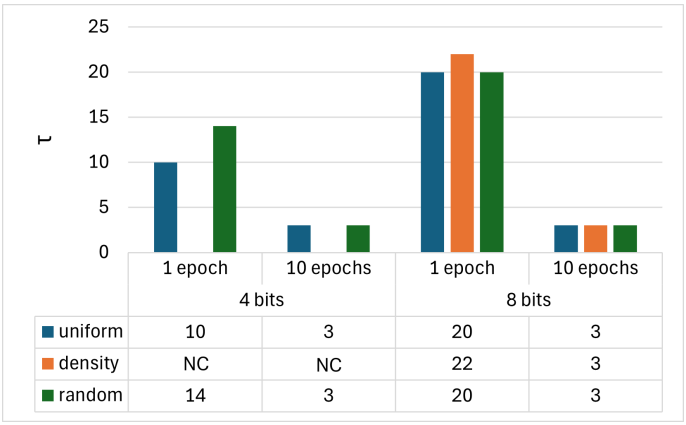
As explained in Section 4.2.2, we can initialize the k-Means either with a linear (uniform) distribution of values, or with a density-based or random distribution. A visual representation of all three approaches is shown in Figure 4.7.

We compare uniform, density-based and random spacing initialization in Figure 5.11, similar to previous sections for the accuracy, the convergence speed τ and the computation time at the clients t_{client}.



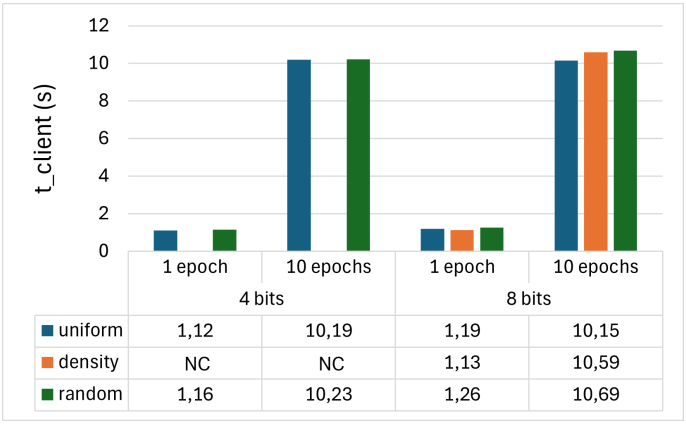
	4 bits		8 bits	
	1 epoch	10 epochs	1 epoch	10 epochs
uniform	48,697	75,988	78,27	78,406
density	NC	NC	78,34	78,17
random	56,143	76,541	78,346	77,6

(a) Accuracy



	4 bits		8 bits	
	1 epoch	10 epochs	1 epoch	10 epochs
uniform	10	3	20	3
density	NC	NC	22	3
random	14	3	20	3

(b) τ



	4 bits		8 bits	
	1 epoch	10 epochs	1 epoch	10 epochs
uniform	1,12	10,19	1,19	10,15
density	NC	NC	1,13	10,59
random	1,16	10,23	1,26	10,69

(c) $t_{client} = t_{train} + t_{compress}$

Figure 5.11: Evaluation of spacing initialization for k-Means

Before establishing the comparison, we note that for 4 bits quantization (16 centroids), the density spacing initialization leads to a crash of the simulation. However, the other ap-

proaches don't experience this issue, then we have enough material to draw conclusions.

Once again, we can evaluate all metrics one at a time:

- **Accuracy:** As said, the density approach crashes for 4 bits. For this quantization level, the random approach gives a better accuracy than the uniform one. Also using 10 local epochs provides a much better final accuracy for both approaches.
  For 8 bits quantization (256 centroids), 1 epoch provides better results overall, and all approach have a similar final accuracy. With 10 epochs, only the random initialization is slightly under.

- **Convergence speed:** Beware of the confusion! The convergence speed is measured with τ in terms of FL rounds, independently from to k-Means convergence which happens at each round. If the latter is different between all three approaches, we might interpret it from t_{client}. The difference of convergence we read here with τ is a consequence of the quality of the k-Means, which differs with the initialization space strategy.
  Considering we already eliminated the density initialization approach, uniform and random share the same performances.

- **Computation time:** This time, the uniform initialization approach is the faster of the two, but not by much.

- **Size of the model:** The initial spacing for k-Means doesn't have a direct influence on the size of the model. The only variation would be about the distribution of weights over the centroids: with Huffman encoding, an unequal distribution could have positive effect. However, we can't predict this here.

In conclusion, the uniform and random initialization approaches provide close results. We will keep the **uniform spacing initialization** because it is more reliable than the random which might sometimes provides bad results. Note that this is also the conclusion of Han *et al.* [14], who explain that uniform initialization allows to represent the outliers as well, what the density fails to do, leading to a loss of accuracy.

5.3.4 Effect of the level of quantization

Then, we want to evaluate the impact of quantization. We established that we will use PTQ with k-Means, execute it layer-wise, and use a uniform spacing initialization.

We can then take a look again at Figure 5.9, whose experiment — with ResNet-12 and 10 clients — uses exactly these settings (miracle!), and compare the different levels of quantization.

For quantization on 8 bits (256 centroids), the accuracy drops of less than 1 percent for 1 epoch, and even grows with 10 epochs. For 4 bits quantization (16 centroids), there is a significant drop for 1 epoch, which is not acceptable in our study. However with 10 epochs, the drop is less than 3 percents.

To go further, we try to train the ResNet-18 with the same PTQ on IID data and 100 clients, and only one local epoch per client at each round. The results in terms of accuracy are shown in the Figure 5.12.

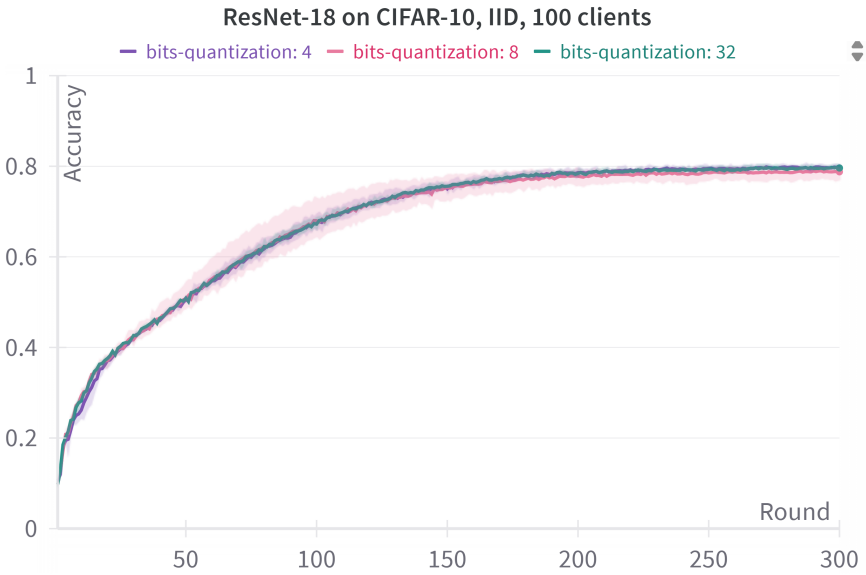


Figure 5.12: Accuracy for ResNet-18

All three curves (baseline, quantization on 8 bits, quantization on 4 bits) are really similar. That means that in this design, quantizing on 4 bits doesn't induce any significant drop of accuracy, nor latency in the τ coefficient for speed of convergence. For the training time (not shown here), it is close whatever the quantization level.

5.3.5 Conclusion on quantization

Finally, we want to conclude on quantization. We have defined the best settings for an efficient quantization: PTQ, layer-wise, uniform spacing. We have also seen that the impact of the quantization level varies with the model and conditions.

The conclusion is that just as for pruning, we established a proof of concept, but there is no real recipe to predetermine the best number of centroids to use without impacting performance.

5.4 Fine-tuning of Hyperparameters

We have validated our pruning and quantization implementations, and still aim to prove the efficiency of our FCP (full compression pipeline). However, we also want to refine some details right before, about the hyperparameters we use.

Particularly, for the learning process at the client side (which happens right before quantization), the literature doesn't agree on two values: the learning rate and the batch size. We'll try in the two following sections to find the best compromise.

Additionally, for the different experiments they conduct, [35] use either 1 or 10 local epochs. We also want to conclude which one is the more adapted to our work.

The final hyperparameter discussed in Section 4.1.1 is the momentum for SGD. We do not evaluate it in our experiments, as a value of 0.9 is widely accepted in the literature and commonly used in existing implementations.

5.4.1 Learning rate

The learning rate influences the quality of the model update returned after the SGD. A smaller learning rate offers more precision, but can also sometimes lead to non-optimal minima, and most of all, generally increases the training time t_{train}.

In [35], a learning rate of 0.01 is used, whereas the Flower framework [6] initially implements a learning rate of 0.1. We want to compare these two configurations, and conclude in our case.

To this end, we train a ResNet-12 on the CIFAR-10 with IID distribution on 10 clients, and measure the accuracy, round time, and τ coefficient. We evaluate different cases: without compression, with pruning only (30%), with quantization only (on 8 bits), or with both techniques.

Figure 5.13 presents the results for different hyperparametrizations. A few remarks: the round time is an approximation of the average round time over the whole FL process, rounded to the closest half second. The τ coefficient (tau) is as usual measured in number of whole steps. We also use a color scale to visually indicate where the best values are. Beware, this is not a linear scale, but simply a four-color indicator ranking all values!

First to determine the best learning rate, for each table we compare the first and third line, or the second and fourth line. We also take care not to confuse the left and rights sections of the tables, corresponding to different numbers of epochs.

No compression	1 local epoch			10 local epochs		
	accuracy	round time	tau (steps)	accuracy	round time	tau (steps)
lr 0.1 & bs 8	0.71730	7.5	23	0.71884	70	3
lr 0.1 & bs 32	0.78563	4	21	0.79049	32.5	3
lr 0.01 & bs 8	0.80841	7.5	12	0.78853	69	3
lr 0.01 & bs 32	0.79775	4	12	0.77643	32.5	3

(a) Results when no compression is applied

Pruning only	1 local epoch			10 local epochs		
	accuracy	round time	tau (steps)	accuracy	round time	tau (steps)
lr 0.1 & bs 8	0.75528	8.5	26	0.72140	70.5	3
lr 0.1 & bs 32	0.79699	5	19	0.78550	33.5	3
lr 0.01 & bs 8	0.80125	8	12	0.77743	70	3
lr 0.01 & bs 32	0.79456	5	14	0.77557	33	4

(b) Results when only pruning is applied

Quantization only	1 local epoch			10 local epochs		
	accuracy	round time	tau (steps)	accuracy	round time	tau (steps)
lr 0.1 & bs 8	0.74805	8	25	0.75328	68.5	4
lr 0.1 & bs 32	0.79073	4	21	0.78193	32.5	3
lr 0.01 & bs 8	0.79652	8	12	0.78256	69.5	3
lr 0.01 & bs 32	0.78776	4	14	0.77663	32.5	4

(c) Results when only quantization is applied

Pruning & Quantization	1 local epoch			10 local epochs		
	accuracy	round time	tau (steps)	accuracy	round time	tau (steps)
lr 0.1 & bs 8	0.75068	8.5	24	0.74972	70	4
lr 0.1 & bs 32	0.78993	5	18	0.78136	33	3
lr 0.01 & bs 8	0.80578	8.5	11	0.78243	70	3
lr 0.01 & bs 32	0.79516	5	14	0.77573	33	3

(d) Results when pruning and quantization are applied

Figure 5.13: Results for different hyperparametrizations

For a batch size of 8, we always get a significantly worse accuracy with the biggest learning rate. The τ is also often around two times worse when considering one single epoch. For the rest (τ for 10 epochs, round time), results are similar.
For a batch size of 32, this time the accuracy is similar whatever the learning rate, and t_{round} is unchanged. However, once again τ is significantly smaller for a 0.01 learning rate.

We can note that these observations hold almost independently of the type of compression applied, which is expected since compression happens after training and does not directly affect its process.

Based on these considerations, we choose to **set the learning rate to 0.01** for our framework, following the recommendation in [35] and diverging from the default setting used in the original Flower implementation.
Side note: all experiments presented previously (about pruning and quantization) have been run with such a learning rate.

5.4.2 Batch size

We also want to set the ideal batch size. In a similar fashion, the Flower framework initially implements a batch size of 32, while Grativol uses 8. The batch size also influences the training: changing from 32 to 8 may increase training noise, which can help regularize the model but also make learning less efficient.
Let's take a look at its influence in our context.

This time, we still compare the results of Figure 5.13, but we can also only consider the case where the learning rate is lr= 0.01.

For each case of the compression or number of epochs, the batch size set to 8 provides a better accuracy. However, it makes rounds longer, while also having a good effect on τ. We seek to evaluate the trade-off in Table 5.3, calculated as following: a ratio > 1 is a positive point for batch size 32, else it is positive for batch size 8.

Compression	t_{round} ratio 1 epoch	τ ratio 1 epoch	t_{round} ratio 10 epochs	τ ratio 10 epochs
None	1.875	1.000	2.123	1.000
Pruning	1.600	0.857	2.121	0.750
Quantization	2.000	0.857	2.138	0.750
Both	1.700	0.785	2.121	1.000

Table 5.3: Evaluation of the trade-off between t_{round} and τ convergence speed

The previous property is here illustrated: the t_{round} ratio always gives advantage to batch size 32, while the convergence speed induced by τ is sometimes better with batch size 8. Then the intuition is that total FL training time is better with a larger batch size, so we can also try to measure it.

To this extent, Table 5.4 shows the ratio of FL training time required when using batch sizes 32 and 8. These results are directly obtained by pairwise multiplication of the values from the previous table. This approach makes sense because the convergence speed (measured by the number of steps needed to reach 95% or 99% of the final accuracy) scales linearly with τ, while the round time remains nearly constant throughout training.

Then we see that in our setup, using a batch size of 8 instead of 32 leads to a larger FL

Compression	t_{train} ratio 1 epoch	t_{train} ratio 10 epochs
None	1.875	2.123
Pruning	1.371	1.591
Quantization	1.714	1.604
Both	1.336	2.121

Table 5.4: Ratio of t_{train} for batch size 8 vs. 32: always quicker with 32

training time, from $\times 1.3$ to $\times 2.1$, which induces an equal larger computation.
We also have to keep in mind the accuracy drop induced, evaluated in Table 5.5.

Compression	accuracy drop 1 epoch	accuracy drop 10 epochs
None	1.066%	1.210%
Pruning	0.669%	0.186%
Quantization	0.876%	0.593%
Both	1.062%	0.670%

Table 5.5: Accuracy drop for batch size 8 vs. 32: better with 8

We read that the accuracy drop is around 1%. Depending on the context, it may be acceptable or not. Even if in the context of this thesis we value the computation needs a lot, we also want to get the best accuracy to match the results of papers we compare to. Another key metric of our work is the size of the model, and for this the batch size has no influence. In conclusion, we will **set the batch size to 8** — just like [35] — but keep in mind that 32 might also be a relevant value to make the FL training faster.

5.4.3 Number of epochs

Last but not least, the number of epochs influences the quality of the client training during the FL process. For the different experiments, Grativol always uses both 1 and 10 local epochs at each client's side for each round, and concludes that under certain conditions, 10 epochs allow a faster convergence in terms of rounds, and also sometimes a better accuracy.

In our framework, we want to evaluate the trade-off between the larger τ induced by a smaller number of local epochs, and the larger t_{round} induced by a larger number of local epochs. Once again, we use the Figure 5.13 to compare it. We only consider the lines `lr 0.01 & bs 8`, according to our previous conclusions.

The Table 5.6 sums up the ratios under these conditions. A positive accuracy drop or a

ratio >1 indicate a favorable outcome for using 1 local epoch.

Compression	Accuracy drop	t_{round} ratio	τ ratio
None	1.988%	9.200	0.250
Pruning	2.382%	8.750	0.250
Quantization	1.396%	8.688	0.250
Both	2.335%	8.235	0.273

Table 5.6: Evaluation of the trade-off for round time and convergence speed

First, we can note that the accuracy drop is significant when using 10 epochs instead of a single one. This can be explained by the fact that at the last round, 40% of the clients update the model again (as for each round), but doing 10 epochs will induce more overfitting than 1 epoch, on a model that is already well fitted. This accuracy issue could be enough to conclude, but let's also take a look at the time trade-off in Table 5.7.

Compression	FL time ratio
None	2.300
Pruning	2.188
Quantization	2.172
Both	2.248

Table 5.7: Ratio of time taken for 1 or 10 epochs: always quicker with 1

The results are clear: they also give advantage to the use of a single epoch. Indeed, 10 epochs induce almost ten times more computation (minus the invariable compression and aggregation time), however improve the convergence speed only by about 4, not forgetting the reduced accuracy.

Here there is no doubt: **we set the number of local epochs to 1.**

5.5 Evaluation of the Full Compression Pipeline

In the previous sections, we have defined:

- The best setting for **pruning**: model-wise, unstructured
- The best setting for **quantization**: PTQ, layer-wise, uniform space initialization
- The best **hyperparameters** for FL: learning rate of 0.01, batch size of 8, 1 local epoch

The only compression technique that we didn't discuss yet is the Huffman encoding, however there aren't any parameters to set, and we already explained how we use it in Section 4.2.3.

That means that we are now ready to evaluate the performances of our FCP introduced in Section 4.3.2.

5.5.1 Experimental conditions

First, we will evaluate the FCP on a first scenario we already used in the previous experiments:

- Model: ResNet-12
- Dataset: CIFAR-10
- Number of clients: 10
- Distribution: IID with LDA ($\alpha = 100$)
- Fraction of clients selected at each round: 40%
- Number of rounds: 100

We also want to prove that the FCP can be generalized in FL, so we also define a second scenario with a different dataset:

- Model: ResNet-12
- Dataset: FEMNIST

- Number of clients: 10
- Distribution: IID with LDA ($\alpha = 100$)
- Fraction of clients selected at each round: 40%
- Number of rounds: 50

And also a third scenario to generalize even more on non-IID data:

- Model: ResNet-12
- Dataset: FEMNIST
- Number of clients: 20
- Distribution: non-IID with LDA ($\alpha = 1$)
- Fraction of clients selected at each round: 20%
- Number of rounds: 50

We will evaluate the performances of the FL training for all scenarios, considering the metrics we defined. We will train our models under different conditions, i.e. with different pruning rates and number of centroids for quantization, and also either apply the FCP or not.

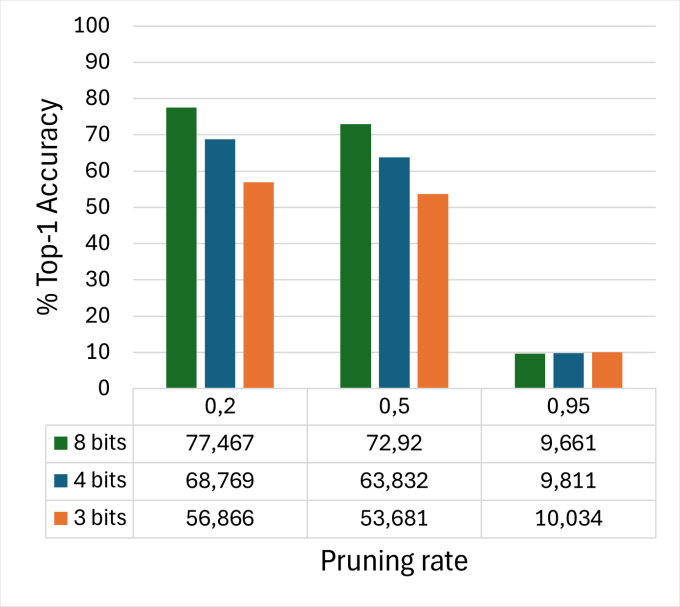
5.5.2 Evaluation of the accuracy

First, we consider for both scenarios the accuracy reached after training. Figure 5.14 and Figure 5.15 sum up the results. We also remind the baseline accuracies in Table 5.8.

Scenario	Final accuracy
1^{st}	80.642%
2^{nd}	88.954%

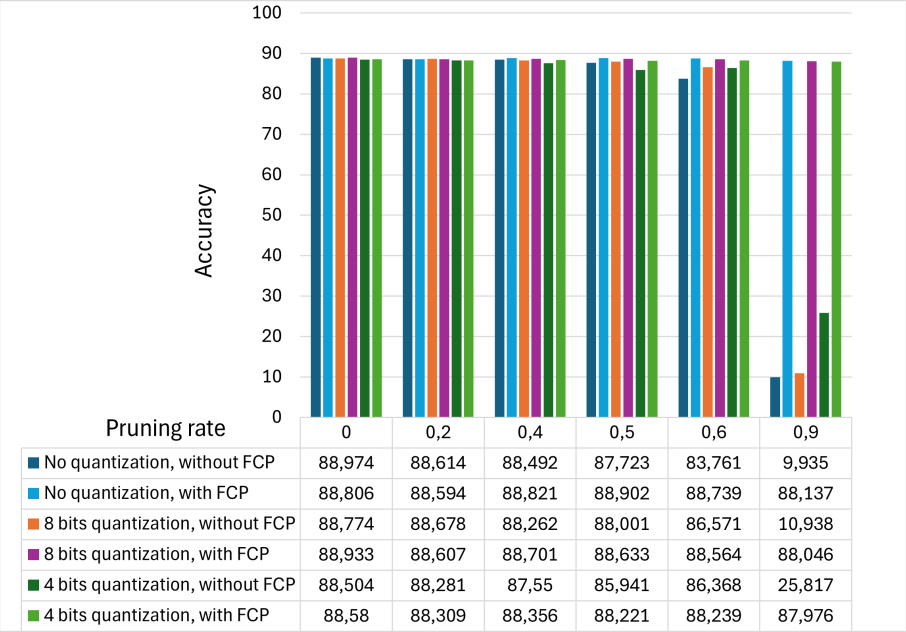
Table 5.8: Accuracy for the baselines

In the first scenario, we can observe a clear effect of the pruning rate and quantization level. With 20% pruning and 8 bits quantization (256 centroids), the final accuracy suffers from a 3% drop. However for smaller number of centroids, the accuracy drops quickly. Increasing the pruning rate from 20% to 50% leads however to an accuracy drop of 'only' 5%, but at 95% the FL training is not converging anymore.



	0,2	0,5	0,95
8 bits	77,467	72,92	9,661
4 bits	68,769	63,832	9,811
3 bits	56,866	53,681	10,034

Figure 5.14: Accuracy in the first scenario



Pruning rate	0	0,2	0,4	0,5	0,6	0,9
No quantization, without FCP	88,974	88,614	88,492	87,723	83,761	9,935
No quantization, with FCP	88,806	88,594	88,821	88,902	88,739	88,137
8 bits quantization, without FCP	88,774	88,678	88,262	88,001	86,571	10,938
8 bits quantization, with FCP	88,933	88,607	88,701	88,633	88,564	88,046
4 bits quantization, without FCP	88,504	88,281	87,55	85,941	86,368	25,817
4 bits quantization, with FCP	88,58	88,309	88,356	88,221	88,239	87,976

Figure 5.15: Accuracy in the second scenario

In the second scenario, we also evaluate the difference between the FCP and the idea of applying successively the different deep compression techniques, commonly used in the literature.

We can first note that accuracy scores are better in the scenario, this is because the FEMNIST dataset is 'easier' to learn for the ResNet-12, compared to the CIFAR-10, due to smaller images (scale of grey 28x28 instead of RGB 32x32). Even if there are more classes to decide between (62 instead of 10), a smaller model could be used with similar performances.

As a direct consequence of these considerations, we see that we can compress much more than in the first scenario, with less accuracy degradation. Curiously, the FCP provides much better results than the iterative method, even though we apply the same methods.

In this second scenario, we can use the FCP with 90% pruning rate and 4 bits quantization (16 centroids) with only 1% accuracy drop. This setting really shrinks the size of the model, and we might compress even more with similar accuracy drop.

About accuracy, we can conclude that the FCP affects it less than the classic iterative compression, and that the impact depends on the dataset and model we train our FL process on.

5.5.3 Evaluation of the speed of convergence

Now that we've evaluated the accuracy impacts of the FCP, let's take a look at the convergence speed, image of the τ coefficient calculated.

The Figure 5.16 and Figure 5.17 report the measured results in the first and second scenarios respectively.

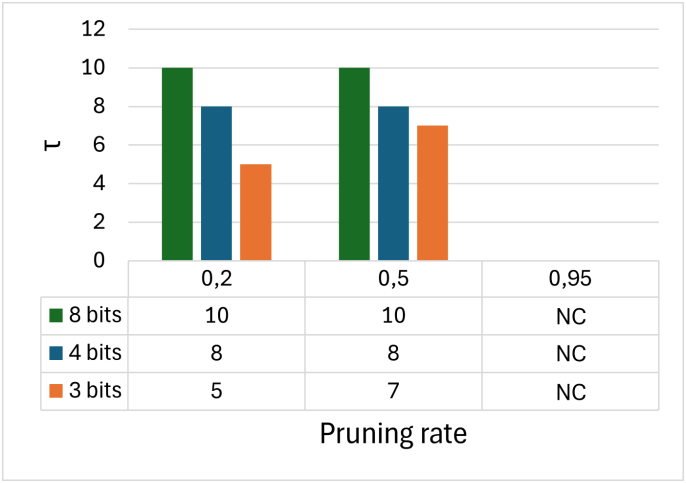
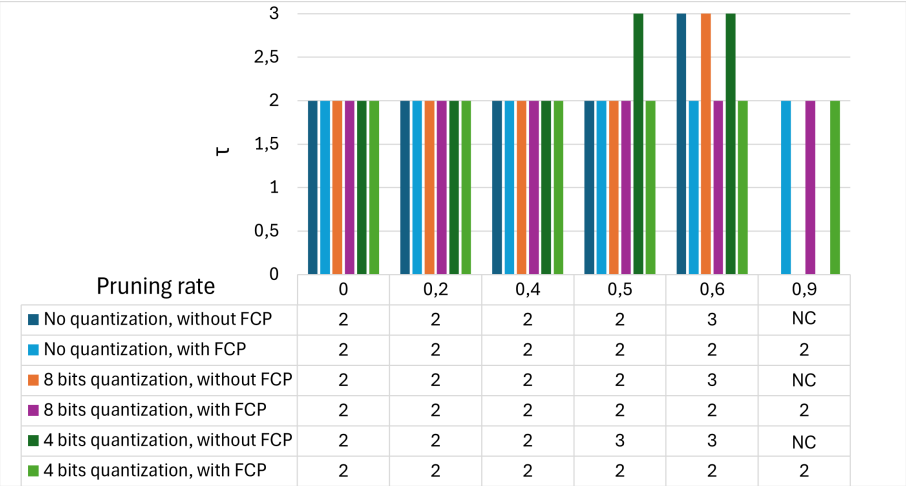


Figure 5.16: τ in the first scenario

In the first scenario, we see a clear trend: τ is (almost) invariable with the pruning rate, as long as we reach convergence, however it depends on the quantization level. Indeed, the more we quantize, the quickest the FL process converges.

For the baseline we measure $\tau = 10$, so compressing doesn't slow down the FL process.



Pruning rate	0	0,2	0,4	0,5	0,6	0,9
No quantization, without FCP	2	2	2	2	3	NC
No quantization, with FCP	2	2	2	2	2	2
8 bits quantization, without FCP	2	2	2	2	3	NC
8 bits quantization, with FCP	2	2	2	2	2	2
4 bits quantization, without FCP	2	2	2	3	3	NC
4 bits quantization, with FCP	2	2	2	2	2	2

Figure 5.17: τ in the second scenario

In the second scenario, the estimated value of τ is always of 2 or 3. This is because the ResNet-12 model learns really fast the important features of the FEMNIST dataset. Still the trend here is also that the FCP helps converging faster.

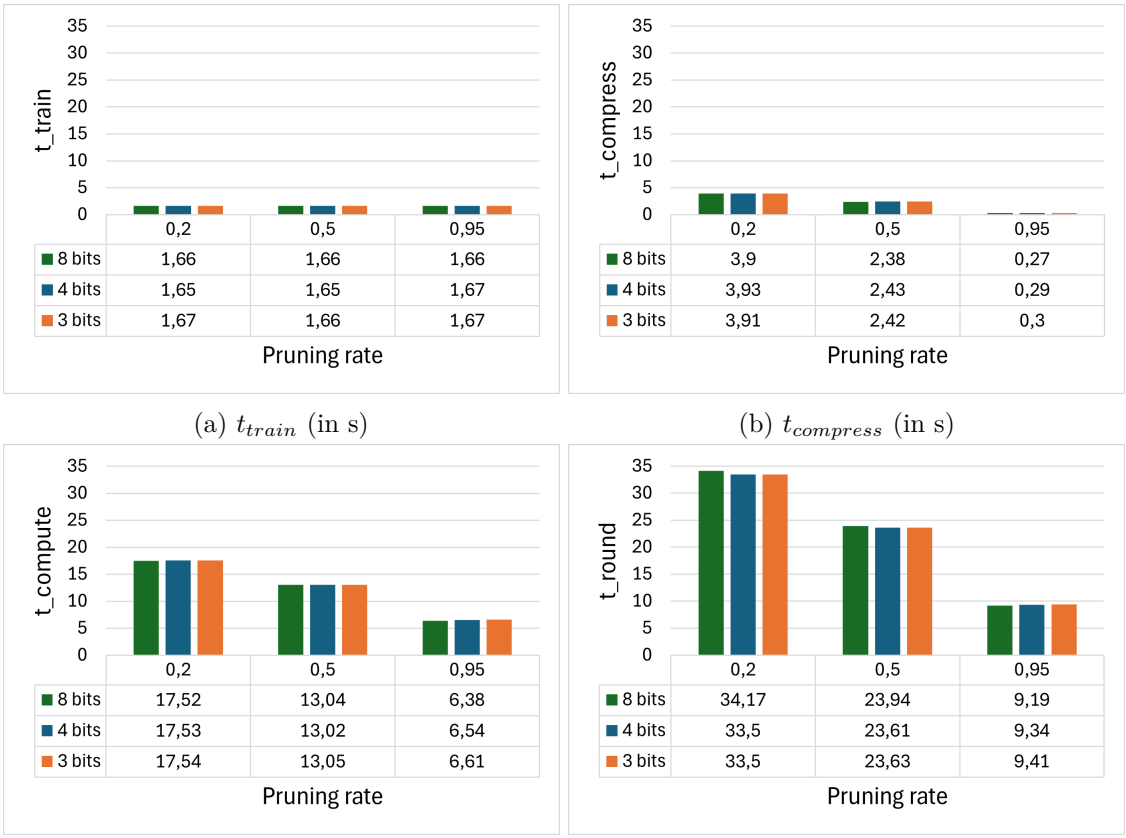
About speed of convergence, we can conclude that the FCP helps getting a positive impact, and that the level of quantization (number of centroids) is more important than the pruning rate to improve it.

5.5.4 Evaluation of the communication overhead

Lastly, we want to evaluate the communication overhead. As explained in Section 4.1.2, for this purpose we need to consider both computation requirements, and result size of the model that we want to communicate.

Evaluation of computation time metrics

Figure 5.18 and Figure 5.19 show the average time measurements across the training for both scenarios.



(a) t_{train} (in s)

(b) $t_{compress}$ (in s)

(c) $t_{compute}$ (in s)

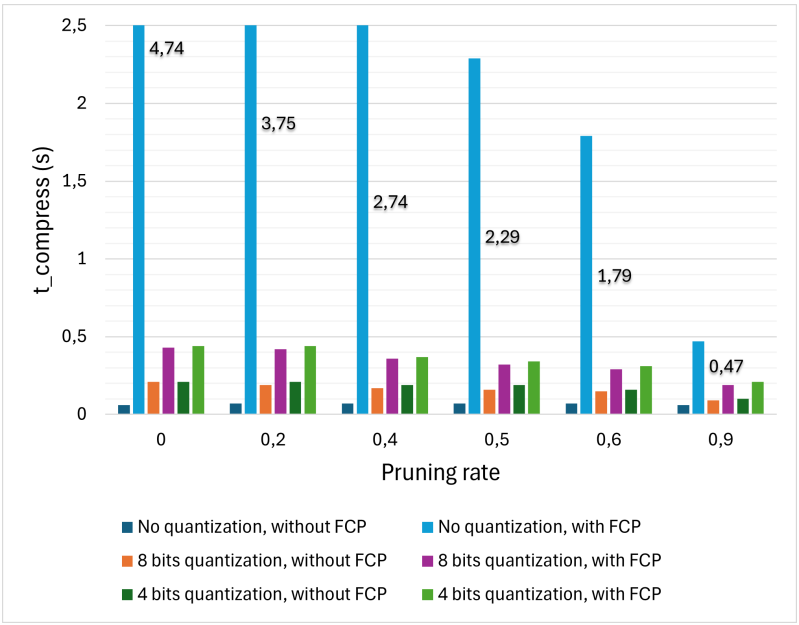
(d) t_{round} (in s)

Figure 5.18: Evaluation of time metrics in the first scenario

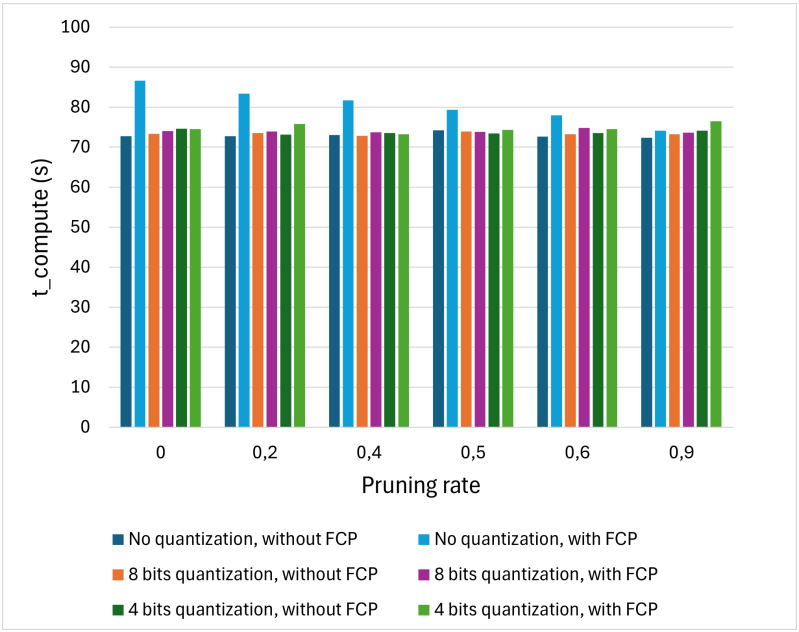
In the first scenario, we can observe that:

- t_{train} is independent of the compression used. This is expected, as we don't try to optimize this process in our work.
- $t_{compress}$ is similar for different quantization levels at a fixed pruning rate. The bigger the pruning rate is, the shorter is the time taken for compression. This makes sense because pruning and quantization are applied successively either way, and a more pruned model means that we apply the k-Means on a smaller set of weights.
- $t_{compute}$ is also dependant on the pruning rate, but not on the quantization level.
- t_{round} is following the same trend as $t_{compute}$.

In the second scenario, we only consider $t_{compress}$ and $t_{compute}$, as the latter is representative of the behavior of t_{round}, and t_{train} is invariable.



(a) $t_{compress}$



(b) $t_{compute}$

Figure 5.19: Evaluation of time metrics in the second scenario

From the results, we analyse:

- $t_{compress}$: a few remarks:

1. The FCP always induces a computation overhead, compared to the iterative method, because there is a further compression technique only in the case of the FCP: the Huffman encoding. If we compared only the time taken to prune and quantize, it would be smaller for the FCP, by construction.

2. When there is no compression, $t_{compress}$ is smaller without FCP, and much bigger with FCP. This happens because in this case, the client doesn't use time to prune or quantize, however for the FCP it still applies a Huffman encoding. As explained in Section 4.2.3, the Huffman encoding can be efficient only on pruned and quantized models, thus we see a huge overhead when applying lesser compression.

3. As a corollary, training with bigger compression (e.g. 90% pruning rate and 4 bits quantization) tend to induce less computation overhead.

- $t_{compute}$: the same tendency is visible, that the FCP induces a global computation overhead at the round scale, however the relative impact isn't as big as for $t_{compress}$. It also shrinks when compressing more.

About computation time metrics, we can conclude that the FCP have a similar impact as 'classical' compression when setting high enough pruning rates and quantization levels.

Evaluation of the size of the model

Now let's take a look at the sizes of communicated models. We evaluate these in the first scenario with the ResNet-12, but also for the ResNet-18. Table 5.9 and Table 5.10 gather the results.

Context	No compression	40% pruning rate & 8 bits quantization
torch.save()	3.1 MB	3.1 MB
torch.save() then zip	2.9 MB	1.2 MB
Huffman encode	9.9 MB	508.5 kB
Huffman then zip	9.2 MB	547.7 kB

Table 5.9: Size of the ResNet-12 under different conditions

Context	No compression	40% pruning rate & 8 bits quantization
torch.save()	44.8 MB	44.8 MB
torch.save() then zip	41.5 MB	31.5 MB
Huffman encode	151.1 MB	7.2 MB
Huffman then zip	143.2 MB	7.2 MB

Table 5.10: Size of the ResNet-18 under different conditions

As the size of the model is only dependant of its architecture and compression degree, we don't need to evaluate it in the second scenario.

These results show two points:

1. Huffman encoding produces a model smaller than the native `torch.save()` method of PyTorch [30], only when pruning and quantization are applied. Depending on the model and compression settings, the boundary at which the efficiency of the two techniques is similar is varying.

2. When using a ZIP algorithm, as [35] does, we reduce the size of models most of the time. However, it doesn't apply to Huffman encoded models for which previous compression (pruning & quantization) was applied. This is because Huffman encoding is optimal, and can't be optimized further, as long as the codebook size is small regarding data and indices encoding sizes.

Then about size of the model, we can conclude that using pruned and quantized networks helps reducing the size of the data packages that the clients communicate to the server.

Evaluation of the communication overhead

The communication overhead is directly a consequence of the two previously evaluated metrics: the computational demand of the FL training, and the size of communicated models.
As a reminder from Section 4.1.2, the communication overhead is defined as the rate of time taken to communicate the model updates of the total time of the FL process.

Taking the first scenario with a pruning rate of 40% of pruning and 8 bits of quantization: from Figure 5.18c, we evaluate an average $t_{compute}$ of 14s; from Table 5.9, we round the size of the compressed model to 500 kB, and of the uncompressed model to 3.1 MB.

Now assuming a communication bandwidth of 1 Mbps (megabits per second), we need 4s to transmit a compressed model update, and 24.8s for an uncompressed model update. Also assuming that 4 clients shall transmit their updates successively in our scenario, that means the the communication overhead is of 74.5% with the FCP, compared to 89.9% without.

With the same settings but a communication bandwidth of 5 Mbps, the communication overhead goes down to 40.2%.
In this setting without the FCP, the communication overhead would be of 84.1%.

About the communication overhead, we can conclude that the network quality influences

a lot the communication overhead, but nevertheless the compression level is also playing a big role. The FCP induces a very positive effect as long as the compressed model size is smaller than the original model.

5.5.5 Conclusion on the FCP

We have proven the efficiency of the FCP in terms of accuracy, speed of convergence, and communication overhead.

If there was one point to remember: the trade-off between the drop of accuracy and all computation advantages provided by a large compression can be optimized; however there is no general recipe for it, as the choice of model, the dataset and its distribution among clients widely influence all metrics.

The FCP is very efficient under well chosen conditions, and quality of the training can be evaluated during the first FL rounds. Indeed, by testing a few different settings (pruning rate and quantization level), the first few rounds will give hints on which will provide the best results at the end of a full training.

6 Conclusion and Perspectives

In this study, we address the critical challenges of communication overhead and energy efficiency in Federated Learning (FL). By evaluating and enhancing state-of-the-art deep compression techniques, we introduce an optimized approach: the Full Compression Pipeline (FCP). This pipeline combines pruning, quantization, and Huffman encoding, leveraging their respective strengths to accelerate computation and minimize resource usage. We demonstrate that, under well-chosen conditions, the FCP outperforms conventional compression strategies, particularly in terms of communication efficiency.

For each compression technique, we investigate the most effective configuration within the FCP. In the case of pruning, we analyze the influence of various pruning rates and scaling strategies. For quantization, we explore both the scaling factor and the selected method—Post-Training Quantization (PTQ) using k-Means clustering—evaluating its performance under different parameter settings. Furthermore, we perform hyperparameter optimization to fine-tune the FCP for maximum efficiency.

Our approach goes beyond state-of-the-art literature on deep compression in FL, effectively integrating the foundational claims of Han *et al.* [14]—specifically, the ability to significantly reduce model size without major accuracy degradation—within the FL context. This integration yields promising results in terms of training performance and system efficiency.

The FCP is evaluated using various neural network architectures and datasets. For instance, training a ResNet-12 model on the CIFAR-10 dataset across 10 clients with an average bandwidth of 5 Mbps, we demonstrate a reduction in communication overhead from 84.1% to 40.2%, with only a 5% drop in model accuracy compared to the uncompressed baseline. This results in each communication round being 1.75 times faster. Given that convergence speed remains comparable to the baseline, the FCP substantially reduces overall computation time and energy consumption.

Future research may focus on generalizing the FCP, aiming to predict optimal compression parameters to achieve the best trade-off between accuracy retention and communication/computational efficiency. Additionally, its robustness under non-IID data distributions remains an important avenue for further investigation, especially because we already proved the efficiency of one of the FCP components —pruning— under these conditions.

Bibliography

[1] Inc. Advanced Micro Devices. *Brevitas Documentation*. https://xilinx.github.io/brevitas/index.html. Accessed: 2025-04-05. 2025.

[2] Flower AI. *How to Implement Strategies*. Accessed: 2025-04-01. 2025. URL: https://flower.ai/docs/framework/how-to-implement-strategies.html.

[3] RAPIDS AI. *cuML API Reference: K-Means Clustering*. Accessed: 2025-04-07. 2025. URL: https://docs.rapids.ai/api/cuml/stable/api/#k-means-clustering.

[4] Rana Albelaihi et al. 'Green Federated Learning via Energy-Aware Client Selection'. In: *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. 2022, pp. 13–18. DOI: 10.1109/GLOBECOM48099.2022.10001569.

[5] Fazal Muhammad Ali Khan, Hatem Abou-Zeid and Syed Ali Hassan. 'Model Pruning for Efficient Over-the-Air Federated Learning in Tactical Networks'. In: 2023 IEEE International Conference on Communications Workshops (ICC Workshops). ISSN: 2694-2941. May 2023, pp. 1806–1811. DOI: 10.1109/ICCWorkshops57953.2023.10283773. (Visited on 08/10/2024).

[6] Daniel J. Beutel et al. *Flower: A Friendly Federated Learning Research Framework*. 2022. arXiv: 2007.14390 [cs.LG]. URL: https://arxiv.org/abs/2007.14390.

[7] Weights & Biases. *WandB Quickstart Guide*. Accessed: 2025-04-01. 2025. URL: https://docs.wandb.ai/quickstart/.

[8] Intergovernmental Panel on Climate Change. 'Stabilization of Atmospheric Greenhouse Gas Concentrations'. In: *Climate Change 1995: The Science of Climate Change*. Ed. by J.T. Houghton et al. Cambridge University Press, 1996. Chap. 4.17, p. 21. URL: https://archive.ipcc.ch/pdf/climate-changes-1995/ipcc-2nd-assessment/2nd-assessment-en.pdf.

[9] PyTorch Contributors. *torch.nn.utils.prune*. Accessed: 2025-04-04. 2025. URL: https://pytorch.org/docs/2.6/nn.html#module-torch.nn.utils.prune.

[10] scikit-learn developers. *sklearn.cluster.KMeans — scikit-learn Documentation*. Accessed: 2025-04-01. 2025. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[11] Steven K Esser et al. 'Learned step size quantization'. In: *International Conference on Learning Representations (ICLR)*. 2020.

[12] Yunchao Gong et al. 'Compressing deep convolutional networks using vector quantization'. In: *arXiv preprint arXiv:1412.6115* (2014).

[13] Suyog Gupta et al. 'Deep learning with limited numerical precision'. In: *International Conference on Machine Learning (ICML)*. 2015.

[14] Song Han, Huizi Mao and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV]. URL: https://arxiv.org/abs/1510.00149.

[15] Kaiming He et al. 'Deep Residual Learning for Image Recognition'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[16] Fraunhofer ISE. *Stromerzeugung in Deutschland im Jahr 2024*. 2024. URL: https://www.energy-charts.info/downloads/Stromerzeugung_2024.pdf (visited on 21/01/2025).

[17] Benoit Jacob et al. 'Quantization and training of neural networks for efficient integer-arithmetic-only inference'. In: *Computer Vision and Pattern Recognition (CVPR)*. 2018.

[18] Peter Kairouz, H. Brendan McMahan, Brendan Avent et al. 'Advances and Open Problems in Federated Learning'. In: *Foundations and Trends in Machine Learning* 14.1-2 (2021), pp. 1–210.

[19] Young Seok Tony Kim. *Deep-Compression-PyTorch: PyTorch implementation of 'Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding'*. Accessed: 2025-04-07. 2018. URL: https://github.com/mightydeveloper/Deep-Compression-PyTorch.

[20] Raghuraman Krishnamoorthi. 'Quantizing deep convolutional networks for efficient inference: A whitepaper'. In: *arXiv preprint arXiv:1806.08342* (2018).

[21] Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[22] Tian Li et al. 'Federated Learning: Challenges, Methods, and Future Directions'. In: *IEEE Signal Processing Magazine* 37.3 (2020), pp. 50–60.

[23] Tian Li et al. *Federated Optimization in Heterogeneous Networks*. 2020. arXiv: 1812.06127 [cs.LG]. URL: https://arxiv.org/abs/1812.06127.

[24] Christos Louizos et al. 'Relaxed quantization for discretized neural networks'. In: *International Conference on Learning Representations (ICLR)*. 2019.

[25] Amirhossein Malekijoo et al. *FEDZIP: A Compression Framework for Communication-Efficient Federated Learning*. 2021. arXiv: 2102.01593 [cs.LG]. URL: https://arxiv.org/abs/2102.01593.

[26] H. Brendan McMahan et al. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. 26th Jan. 2023. arXiv: 1602.05629[cs]. URL: http://arxiv.org/abs/1602.05629.

[27] Markus Nagel et al. 'White paper: A guide to post-training quantization with TensorFlow'. In: *arXiv preprint arXiv:2106.08295* (2021).

[28] Hidehiko Okada. 'Evolutionary Training of Binary Neural Networks by Differential Evolution'. In: *International Journal of Scientific Research in Computer Science and Engineering* 10.1 (2022), pp. 26–31. URL: https://ijsrcse.isroset.org/index.php/j/article/view/476.

[29] David Patterson et al. 'The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink'. In: *Computer* 55.7 (2022), pp. 18–28. DOI: 10.1109/MC.2022.3148714.

[30] PyTorch. *PyTorch Documentation*. Accessed: 2025-04-01. 2025. URL: https://pytorch.org/docs/stable/index.html.

[31] Pian Qi et al. 'Model aggregation techniques in federated learning: A comprehensive survey'. In: *Future Generation Computer Systems* 150 (2024), pp. 272–293. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.09.008. URL: https://www.sciencedirect.com/science/article/pii/S0167739X23003333.

[32] John Rachwan et al. *Winning the Lottery Ahead of Time: Efficient Early Network Pruning*. 2022. arXiv: 2206.10451 [cs.LG]. URL: https://arxiv.org/abs/2206.10451.

[33] Prajit Ramachandran, Barret Zoph and Quoc Le. 'Swish: a Self-Gated Activation Function'. In: (Oct. 2017). DOI: 10.48550/arXiv.1710.05941.

[34] Farheen Ramzan et al. 'A Deep Learning Approach for Automated Diagnosis and Multi-Class Classification of Alzheimer's Disease Stages Using Resting-State fMRI and Residual Neural Networks'. In: *Journal of Medical Systems* 44 (Dec. 2019). DOI: 10.1007/s10916-019-1475-2.

[35] Lucas Grativol Ribeiro et al. *Federated learning compression designed for lightweight communications*. 23rd Oct. 2023. arXiv: 2310.14693[cs]. URL: http://arxiv.org/abs/2310.14693 (visited on 07/11/2024).

[36] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automation*. 1957. URL: https://bpb-us-e2.wpmucdn.com/websites.umass.edu/dist/a/27637/files/2016/03/rosenblatt-1957.pdf (visited on 28/01/2025).

[37] Shirin Salehi and Anke Schmeink. 'Data-Centric Green Artificial Intelligence: A Survey'. In: *IEEE Transactions on Artificial Intelligence* 5.5 (May 2024). Conference Name: IEEE Transactions on Artificial Intelligence, pp. 1973–1989. ISSN: 2691-4581. DOI: 10.1109/TAI.2023.3315272. URL: https://ieeexplore.ieee.org/document/10251541/?arnumber=10251541 (visited on 30/09/2024).

[38] Roy Schwartz et al. 'Green AI'. In: *Communications of the ACM* 63.12 (17th Nov. 2020), pp. 54–63. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3381831. URL: https://dl.acm.org/doi/10.1145/3381831 (visited on 14/01/2025).

[39] SciPy Development Team. *scipy.sparse.csr_matrix*. Accessed: 2025-04-07. 2025. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html.

[40] Suhail Mohmad Shah and Vincent K. N. Lau. 'Model Compression for Communication Efficient Federated Learning'. In: *IEEE Transactions on Neural Networks and Learning Systems* 34.9 (2023), pp. 5937–5951. DOI: 10.1109/TNNLS.2021.3131614.

[41] Osama Shahid et al. *Communication Efficiency in Federated Learning: Achievements and Challenges*. 2021. arXiv: 2107.10996 [cs.LG]. URL: https://arxiv.org/abs/2107.10996.

[42] *Sparse arrays (scipy.sparse) — SciPy v1.15.2 Manual*. URL: https://docs.scipy.org/doc/scipy/reference/sparse.html (visited on 18/03/2025).

[43] 'Transport'. In: *Climate Change 2022 - Mitigation of Climate Change: Working Group III Contribution to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change*. Cambridge University Press, 2023, pp. 1049–1160.

[44] Saeed Vahidian, Mahdi Morafah and Bill Lin. 'Personalized Federated Learning by Structured and Unstructured Pruning under Data Heterogeneity'. In: *2021 IEEE 41st International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2021, pp. 27–34. DOI: 10.1109/ICDCSW53096.2021.00012.

[45] Luping WANG, Wei WANG and Bo LI. 'CMFL: Mitigating Communication Overhead for Federated Learning'. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 954–964. DOI: 10.1109/ICDCS.2019.00099.

[46] Qiang Yang et al. 'Federated Machine Learning: Concept and Applications'. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 10.2 (2019), pp. 1–19.

[47] Chen Zhang et al. 'A survey on federated learning'. In: *Knowledge-Based Systems* 216 (2021), p. 106775. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2021.106775. URL: https://www.sciencedirect.com/science/article/pii/S0950705121000381.

[48] Zheqi Zhu et al. 'Towards Efficient Federated Learning: Layer-Wise Pruning-Quantization Scheme and Coding Design'. In: *Entropy* 25.8 (2023). ISSN: 1099-4300. DOI: 10.3390/e25081205. URL: https://www.mdpi.com/1099-4300/25/8/1205.